University of Alberta

Shared-Memory Optimizations for Virtual Machines

by

A. Cameron Macdonell

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

Doctor of Philosophy

Department of Computing Science

Fall 2011
Edmonton, Alberta

To Nadine

Abstract

Virtual machine environments are becoming more common due to the increased performance of commodity hardware and the emergence of cloud computing for large scale applications. As the use of virtual machines continues to grow, performance critical applications will require efficient mechanisms to achieve their tasks.

We introduce *Nahanni* as a mechanism for shared memory communication in virtual machine environments. Nahanni allows virtualized applications, those running inside virtual machines, to communicate through shared memory for both data movement and synchronization when VMs are co-located on the same host machine. We describe the design, implementation, and evaluation of Nahanni as part of the QEMU/KVM virtualization platform.

We have modified existing communication layers to measure the performance benefit of Nahanni. Through microbenchmarks and applications, we demonstrate that shared memory is a useful and efficient communication mechanism in virtualized environments. Further, we discuss how the design and implementation of Nahanni enables a new class of applications, ones that use structured data, to benefit from the use of shared memory.

Acknowledgements

Although this work is published under my name, I would be remiss not to mention the key contributions and support of several people. Most notably, I must thank my supervisor Paul Lu for providing unending guidance, advice, and motivation in helping me to achieve this degree.

As well, I would like to thank the QEMU/KVM development community, in particular Avi Kivity, Anthony Liguori, Christian Borntraeger, Alexander Graf and all others that provided constructive feedback as Nahanni was developed.

Finally, thank you to my family, especially my parents and siblings for their support and most of all to my wife Nadine for her unflinching support to help me achieve this degree.

Contents

List of Tables

List of Figures

List of Acronyms

ACL Access Control List

API Application Programming Interface

ASDM Asymmetric Distributed Shared Memory

BAR Base Address Register

CEM Computational Electromagnetics

CFD Computational Fluid Dynamic

CPU Central Processing Unit

DDI GAMESS's Distributed Data Interface

DFT Density-Functional Theory

DMA Direct Memory Access

GAMESS General Atomic and Molecular Electronic Structure System

HPC High-Performance Computing

IMR Interrupt Mask Register

I/O Input/Output

IP Internet Protocol

IPC Interprocess Communication

ISA Instruction Set Architecture

ISR Interrupt Status Register

KVM Kernel-based Virtual Machine

MPI Message Passing Interface

MSI Message Signalled Interrupts

NAT Network Address Translation

NIC Network Interface Card

NUMA Non-Uniform Memory Access

OS Operating System

PCI Peripheral Component Interface

QEMU Open-source Emulation Program (despite capitalization, it is not an acronym)

SMS Shared-Memory Server

SVM AMD's Secure Virtual Machine

TBB Intel's Thread Building Blocks

TCP Transmission Control Protocol

UDS Unix Domain Socket

UMA Uniform Memory Access

VDE Virtual Distributed Ethernet

VM Virtual Machine

VMCI Virtual Machine Communication Interface

VMM Virtual Machine Monitor

VMX Intel's Virtual Machine Extensions

Chapter 1

Introduction

Hardware virtualization is the ability to allow a single hardware platform, consisting of processors (or cores), memory, and input/output (I/O) devices, to be shared concurrently between multiple operating systems (OS). On the desktop, it may be advantageous to run more than one OS at the same time. For example, a Linux user might want to run the Windows OS and a Windows application concurrently inside a virtual machine (VM), instead of using a dual-boot approach. On the server, different users might need different versions of Linux, or a combination of Linux and, say, FreeBSD. By running two concurrent OSes, both users are able to share the same physical server, despite disparate software requirements. In the realm of high-performance computing (HPC), VMs can provide encapsulation to easily deploy HPC applications that may have specific library needs while still providing good performance [18, 36, 53]. Historically, virtualization has been part of computer systems since IBM mainframes of the 1960s [47].

Today, hardware virtualization of servers is an important component of many cloud-computing platforms, especially in the context of Infrastructure-as-a-Service (IaaS) providers. Cloud computing offers remote computing resources on-demand and at a large scale that has freed individuals and organizations from acquiring their own computing infrastructure. Instead, users can simply rent the use of hardware from so-called *cloud providers* by the hour and only pay for what they use. Since a single VM can encapsulate an OS, libraries, and applications, they lessen the burden of running software on unfamiliar hardware and operating systems [43]. Therefore, VMs have become the unit of resource allocation on clouds, and VMs are convenient software encapsulation mechanisms for cloud users.

Although there are many advantages to encapsulating applications and servers inside VMs, one disadvantage of VMs is that there is an extra degree of separation between the processes within the VM (i.e., inside the *guest*) and data that lives outside the VM (i.e., on

the *host*). If the guest VM needs data that sits on the local disk of the host, then (broadly speaking) the data must either be copied into the guest, or served to the guest, in the sense of a file server. For some use cases (e.g., a long-running Web server), the necessary data could be brought inside the VM once and left there for a long time, thus amortizing any data-movement overheads. However, for other use cases, the data is often moved inside and then outside of the VM for each execution of the key application within the VM [57, 18, 36].

Specifically, scientific simulations can be encapsulated within a VM to make it easy to move the application (and its dependent libraries, tool chains, etc.) from compute node to node (e.g., [54]). But, for each simulation, the requisite input and output files must be moved, or staged, across the guest-host barrier.

As well as moving data from a host file system into a VM, data may need to be moved between processes running in different VMs in the case of a parallel computation. For example, two co-located scientific applications may need to share data in a pipeline. Moving data across the inter-VM barriers is likely to be more expensive than the host-guest barrier due to the need to cross more protection domains. Furthermore, even data-intensive uses such as map-reduce applications are being deployed using VMs (e.g., Amazon's Elastic MapReduce) [31]. Arguably, there is a trend towards using a VM as the unit of resource allocation on clouds (e.g., Amazon). For some applications on the cloud, the speed of data movement between the host and guest and between co-located guests is a potential performance bottleneck.

A well-explored idea to speed up interprocess communication (IPC) is to use shared-memory (e.g., [20, 16]) and direct memory access (DMA) techniques to minimize data copying and control transfer overheads. Within the realm of virtualization, previous versions of the VMware products supported shared memory between guests as part of the Virtual Machine Communication Interface (VMCI) [59], but that functionality has been deprecated. And, XenLoop [61] uses shared-memory ring buffers and Fido [10] for Xen provides shared-memory network and block devices. At a pragmatic level, the work in this thesis increases the availability of inter-VM shared memory on more platforms by providing an implementation of shared-memory IPC for the Linux Kernel-based Virtual Machine (KVM), to complement the previous shared-memory work on Xen. But, more importantly, at the architectural level, the work in this thesis presents an alternate and cleaner design of flat, shared memory as compared to alternate designs based on DMA concepts.

Admittedly, there is a school of thought that says that shared memory between heterogeneous processes (thus, implicitly, different Linux KVM VMs and the host) sounds

like a good idea, but is often complicated (e.g., intrusive code changes to the OS) and not necessarily faster than other IPC mechanisms (e.g., optimized stream-data IPC, such as virtio [52]). We are sympathetic to many of these critiques.

As we will show, the Nahanni system for shared memory in Linux KVM (as part of QEMU/KVM version 0.13) is a clean extension of the existing system: the paravirtualized PCI driver for the guest OS is approximately 250 lines of code, and the Nahanni patch for QEMU/KVM is approximately 800 lines of code. Furthermore, Nahanni has no functional nor performance impact on the guest VM if the paravirtualized driver is not loaded into the guest kernel, and Nahanni is a command-line option to QEMU/KVM and can be turned off completely.

Lastly, the potential performance benefits of shared-memory IPC using Nahanni can be large (between 2 and 8 times faster than the next fastest mechanism, as per microbenchmarks in Section 5.3), even when compared to current best practices, such as I/O virtualization (i.e., virtio [52]) and the 9P file system mechanisms of Linux KVM. Nahanni can also provide up to 30% performance improvement for co-located virtual applications.

1.1 Contributions

In exploring the above points through this work, we state the contributions of this works as:

1. **Unintrusive Implementation Architecture.** The design and implementation of an unintrusive shared-memory mechanism for guest-to-host and guest-to-guest IPC, namely Nahanni. Compared to the design alternatives and previous work, Nahanni is carefully crafted to involve a small number of changes (about 1,050 lines of code, none of which change the host OS kernel), and to have no performance impact on VMs that do not use the mechanism.

2. **Low-latency, High-bandwidth Performance.** A demonstration through microbenchmarks and applications that show Nahanni is the fastest mechanism for guest-to-host and guest-to-guest data movement. Using Nahanni for transferring a file into a VM from the host can be up to 8 times faster than the next fastest technique. Also using shared memory for synchronization as opposed to the network can be an order of magnitude faster. Finally, using Nahanni for inter-VM shared memory can result in up to a 30% improvement in application performance for scientific applications such as GAMESS and some benchmarks in the SPEC MPI2007 benchmarking suite.

3. **User-level Architecture: Bypass OS and bottlenecks.** The particular choice of exposing shared memory to the user-level within guests has advantages that are not available with OS-mediated stream-data communications. In particular, Nahanni can support storing structured data between VMs with applications like memcached [37, 63, 22]. While the kernel is involved in configuration, Nahanni allows normal access to shared memory without involving the kernel. Thus, avoiding kernel overheads is important in achieving the best performance and in not introducing (or aggravating) any bottlenecks in either the guest or host OSes.

As an aside, note that Linux KVM is the common term to refer to the family of Linux kernel-based VM systems, which (currently) includes ports for x86 instruction set architectures (ISA) with hardware support for VMs, IBM's S/390, and others. On the x86 ISA, which is our platform, Linux KVM is implemented as a host kernel module and requires a user-level QEMU process to create a VM. In fact, the command-line to run a Linux KVM virtual machine on x86 invokes a QEMU binary. We use the term Linux KVM, or just KVM sometimes, when referring to the hypervisor in general, and we use the term QEMU/KVM when describing the Nahanni code and modifications to the hypervisor since our modifications involve QEMU.

Although our prototype is based on Linux KVM, the combination of a small driver in the guest and localized modifications in the hypervisor make our design a candidate for other hypervisors as well. To be clear, we have not yet ported Nahanni to other VM systems, but we feel it is plausible for future work. Of course, we would look for implementation guidance from the previous work in this area, such as XenLoop [61] and VMware's Virtual Machine Communication Interface (VMCI).

Overall, Nahanni provides a compelling investigation into the uses of shared memory in virtualized environments as well as the practicality of running scientific applications within virtual machines.

Chapter 2

Overview

Nahanni is a new mechanism for sharing memory between virtual machines (VM) and, more specifically, the applications that are running inside those VMs.

Operating systems (OS) have supported sharing memory between applications as a form of interprocess communication (IPC) for decades. Shared memory is a simple and efficient mechanism for communication between cooperating process that are running on the same physical machine. The main benefit from using shared memory for IPC is that no unnecessary copies of communicated data are made as other IPC mechanisms may do [9]. Extraneous copying is a well-studied source of overhead for many (IPC) mechanisms. By being able to read and write directly to a region of memory that another application can also access directly provides a low-overhead mechanism for data transfer or synchronization.

Nahanni offers architectural and other advantages over previous shared-memory IPC approaches in VM environments. Most of the recent work with inter-VM IPC over shared memory have been based on Xen's grant table mechanism to set up shared ring buffers [64, 28, 61]. Thus, the shared memory is not visible to applications at user-level and all data that is transported through shared-memory IPC is in the form of stream data. In contrast, Nahanni's shared memory is visible to user-level code, which greatly simplifies the porting of existing libraries [27] (Section 5.6) and supports pointer-based, non-stream data structures and mechanisms [63, 27]. The one significant exception from the Xen community is Fido [10] which uses grant tables to set up a single, read-only address space among all VMs. Although Fido is capable of supporting pointer-based data structures in theory (i.e., but not demonstrated in the original paper [10]), there are security concerns associated with all participating VMs sharing a read-only address space. In contrast, Nahanni can optionally share different regions of shared memory between different subsets of the VMs, in accordance with whatever security and sharing policy is desired.

As hardware virtualization grows in popularity and functionality, especially for private [55] and public clouds [6], fast, secure, and flexible (e.g., support both pointer-based data structures and stream data) shared-memory IPC can benefit a diverse spectrum of use-cases from file staging (Section 5.3), to Web services [63], to computational science (Sections 5.5 and 5.6).

2.1 Nahanni

Nahanni is a new mechanism for sharing memory between VMs. To simplify the initial description, we will focus on a single VM, or guest, accessing memory that is shared from the host OS that the VM is running on.

Figure 2.1 shows the basic concept of Nahanni. A region of "POSIX Shared Memory on the Host" is made accessible to an application running inside a VM on that host. The guest application can write to this memory (via load and store operations) and the written data is visible (as per hardware cache coherency protocols) to host applications or VMs that are sharing the same region of host memory.

To enable shared memory functionality, the VM requires an interface to the memory via its virtualized hardware. Virtual machines are similar to real hardware in that their interface to the outside is handled by devices. A new virtual device, labelled as the "Nahanni Device" in Figure 2.1, is created that enables the guest operating system to access the shared memory on the host.

Once the guest operating system is able to access the shared memory, it will expose the memory to its "Guest User Application". If two or more guests access the same host shared-memory region, then applications within those guests will be able to communicate via that region of shared memory.

The detailed design and implementation of Nahanni will be described in Chapter 4. However, Figure 2.2, which is identical to Figure 4.20, shows the key elements of the final form of Nahanni: multiple VMs (three shown in the figure, but can be an arbitrary number) share the POSIX shared memory via a series of `mmap()` operations to the user level, with support for inter-VM interrupts via the Linux eventfd mechanism. Before moving on to the details of Nahanni's implementation, the question of why shared memory is useful for VMs will be discussed.

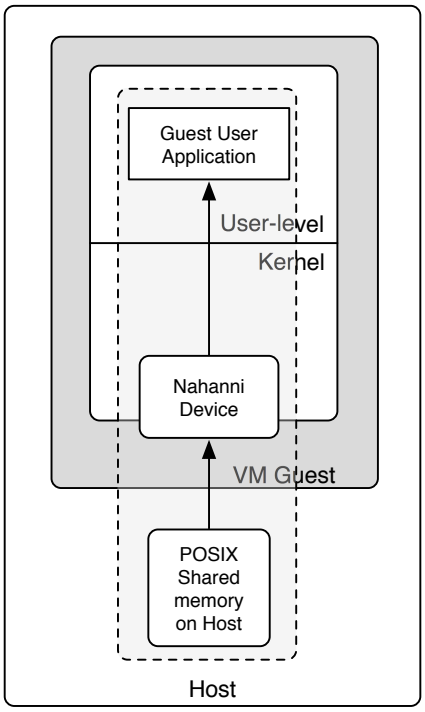


Figure 2.1: The Basic Architecture of Nahanni

Shared memory on the host is shared via the Nahanni device to a virtualized application running inside the guest VM. The shaded box indicates that the memory is shared, no copying occurs when data is written to or read from the shared memory.

2.2 Motivation and Use Cases

In recent years, VMs have become a popular technology for server consolidation, desktop virtualization, and system isolation for testing and development, to name just a few broad categories. In scientific computation, VMs are proving useful as way to package, deploy, and launch applications [54, 31, 18, 36]. These uses indicate trends of VM usage that motivates this work.

VMs as unit of resource allocation: With the emergence of public cloud providers, such as Amazon Elastic Computing Cloud (EC2) [4], Flexiant [19], and Rackspace [49], VMs have also become a unit of resource allocation and provisioning: adding or removing resources from a system involves adding or removing VMs. Although some cloud providers offer VM instances of different resource sizes, many workloads and systems will likely be designed around adding and removing entire VMs, especially if the goal is to scale across many physical hosts and/or data centres. Therefore, such a system will include many host-

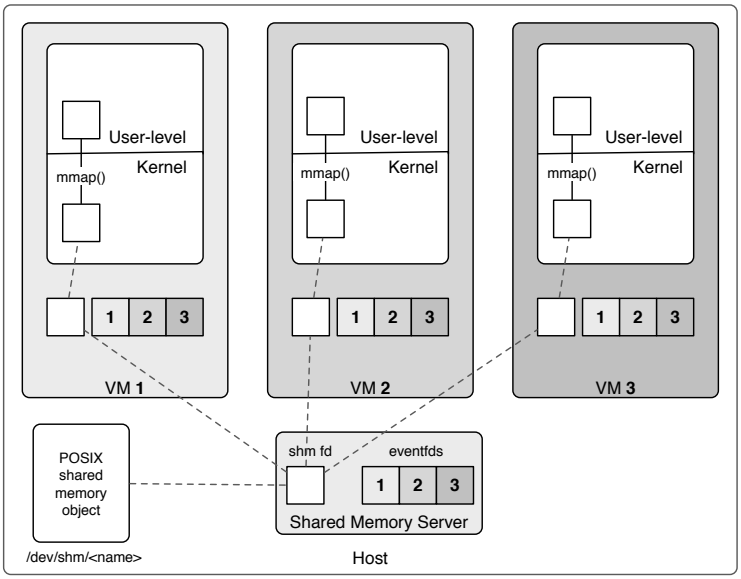


Figure 2.2: Three Virtual Machines using Nahanni

With other features that will be introduced in the coming chapters, Nahanni enables multiple VMs to share memory as illustrated above. Note that this figure is the same as Figure 4.20, but is shown here to give an idea of how multiple, co-located VMs can share memory via Nahanni.

---

to-guest boundaries as well as guest-to-guest transitions. Our work with Nahanni is an attempt to optimize these data movement scenarios.

Data stage-in, stage-out scenario: For example, suppose a scientific simulation application is encapsulated inside a VM. The simulation is mostly used for parameter sweeps, so it might be desirable to launch hundreds of VM instances to run independent simulations. Each simulation instance has to load input files pertaining to the simulation. At the end of the simulation, the output files have to be staged out. Of course, the amount of data to be staged in and out depends on the specific simulation, and it can vary from tens of megabytes (e.g., molecular dynamics) to gigabytes (e.g., seismic or geological simulations based on empirical data; visualization-oriented output).

Moving data from host-to-guest (and the symmetrical case of guest-to-host) is an important enough use case that the 9P file system was adapted and paravirtualized for Linux KVM to handle that special case [58]. As an aside, 9P was historically available in Xen, but has since been deprecated (noted by Hensbergen [58]). And, although there are other tangible benefits from a file system approach, Sections 5.3 and 5.4 show that Nahanni can be significantly faster than 9P for data staging from a pure performance point of view.

Of course, many scientific simulations may take hours of computation within the VM and dwarf any data stage-in and -out overheads in relative terms. However, it is still worthwhile to address an overhead that can be substantial in absolute terms, as both 9P and Nahanni are trying to do. After all, many a computational scientist has waited impatiently for the first simulation to load all of its data, start, and allow for a quick sanity check by the scientist before the next batch of simulations are launched.

Inter-VM communication (e.g., pipeline, parallel application): As argued by the authors of the Fido system [10], and consistent with the notion of VMs being a unit of resource allocation, complex systems of the future might consist of a set of pipelined VMs, in an analogous way to the classic Unix pipeline of separate processes.

For example, there might be a storage subsystem (e.g., implementing a custom redundancy and de-duplication policy) inside a VM, connected to a data-parallel analytics engine inside a different VM, and connected to a front-end Web interface that provides visualizations inside a VM. The analytics engine might include something as simple as a grep or search for regular expressions, a common use for a "map" phase in a map-reduce computation [14]. Or, perhaps two co-located computations need to exchange data through a computation. The data could be exchanged through shared memory instead of over a virtual network.

Again, the amount of data that is moved between pipeline stages or co-located computational tasks (and thus VM-to-VM) varies from application to application. But, if the overhead of moving the data is optimized well enough, the set of applications that are viable in such a pipeline architecture would be larger. We consider the impact of the compute-to-data ratio of various applications in Section 5.4.

2.3 Caveats

Despite our work with Nahanni, we do not claim that shared-memory IPC and the Nahanni architecture are the right choices for all scenarios. For example, if data movement and sharing overheads are not the bottleneck for a given application or workload, then existing mechanisms are likely satisfactory.

Code changes: The biggest caveat for using Nahanni is that application modifications are required to take advantage of the new mechanism. Without changes to the network stacks of the guest(s) and (possibly) the host, as is done by XenLoop and Fido, Nahanni cannot be used transparently. However, as with our experiments, code changes to use Na-

hanni can be entirely at the user-level (avoiding error-prone kernel changes) and can be hidden within user-level code libraries or behind user-level executables.

Potential Loss of Isolation: An advantage of VMs is that different VMs are isolated from each other. Except for resource contention, and in the absence of bugs that cause a host to crash, an isolated VM need not be affected nor affect another VM. Isolation, among other things, allows a VM to be migrated from host-to-host to balance load and improve resource scheduling.

Upon first consideration, Nahanni might appear to break isolation in an irreparable way. But, Nahanni is an optional feature and VMs that do not use the feature continue to be as isolated as any other VM. As well, VMs that do not turn on the Nahanni mechanism have the exact same performance as before.

Even if a set of VMs use Nahanni, the departure of one participating VM does not affect the use of Nahanni by the other VMs that are sharing memory at the level of the IPC mechanism. Of course, if the content of shared memory is left in an inconsistent state, there must be an application-specific way to recover, but that problem has been well-studied (e.g., shared-disk storage systems) and even VMs that communicate via sockets require some kind of application-specific or protocol-specific way to handle faults. In other words, isolation still remains possible for applications that do not use Nahanni. And, recovering from faults between a set of communicating applications (e.g., client-server) is an orthogonal problem to whether Nahanni is used.

Migration: Note that VM migration, in specific circumstances, works correctly with Nahanni, so even that feature is not necessarily broken. For example, a single VM using Nahanni for host-to-guest IPC can be migrated from one host to another host under Linux KVM. Of course, the nature and content of the IPC may change when the host changes, but the VM instance itself and a snapshot of the shared-memory contents can be successfully migrated and re-started. We have not yet done any performance optimizations related to migrating shared-memory IPC (e.g., memory ballooning, iterative copying) or abstractions that work for both shared memory and distributed memory [3], so it is premature to report on performance, but we have successfully tested migration for correctness. In short, Nahanni does not automatically break isolation or migration.

Chapter 3

Background and Related Work

Nahanni is designed to enhance the performance and capabilities of virtualized applications through the use of memory that is shared between virtual machines (VMs). Given these goals, Nahanni's design, implementation and evaluation are strongly tied to VMs and by extension to the broader concept of hardware virtualization. Nahanni also builds upon the large body of related work on interprocess communication (IPC), in particular IPC that is based upon shared memory.

This chapter will discuss the necessary background concepts and related work that is relevant to Nahanni. The concepts and previous work on virtualization will be discussed first, followed by IPC. Finally, some additional hardware and operating system (OS) concepts will be discussed that are relevant to Nahanni.

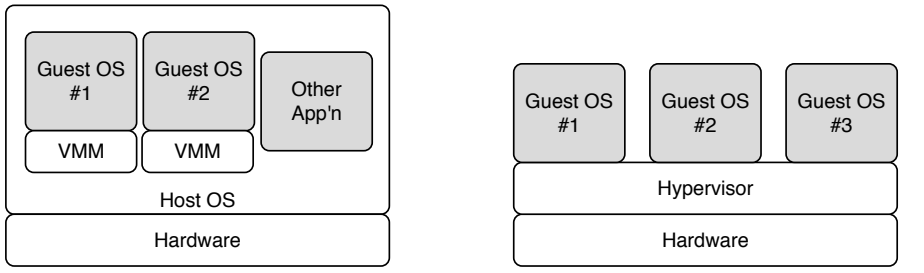
3.1 Virtualization Basics

As mentioned, virtualization and VMs are central to Nahanni. Virtualization is not a new concept and therefore there is a fair amount of previous work that relates to VMs and more specifically to inter-VM IPC which is the focus of Nahanni.

3.1.1 Hardware Virtualization

Hardware virtualization is a mechanism that allows multiple OSes and their respective applications to run simultaneously on the same hardware. Hardware virtualization has been in use for decades. In the 1960s, it was available in IBM mainframe computers [47]. The goal of hardware virtualization is to allow two or more operating systems to share the hardware such that each OS and its respective applications do not need to be modified in order to be virtualized.

There are two basic approaches to hardware virtualization which are both illustrated in



(a) Guest VMs running on top of a Host OS via VMMs

(b) Guest VMs running on a hypervisor

Figure 3.1: Two approaches to hardware virtualization

Figure 3.1. The first approach, shown in Figure 3.1 (a) consists of one OS (called the *host*) running directly on the hardware. The host OS runs software called a *virtual machine monitor* (VMM) for each virtualized OS (called *guests*). In short, guest OSes are virtualized by the VMM on top of the host OS. Like any application on the host, VMs are encapsulated within a regular OS process. Well-known VMMs that use this model are VMware Workstation, KVM and VirtualBox. The second approach, shown in Figure 3.1 (b), involves running a layer of software directly on the hardware called a *hypervisor*. The hypervisor then runs all OSes as guests (i.e., there is no host OS). The hypervisor is not a full-fledged host OS like in the first approach. A hypervisor is a custom software layer, similar to an OS in many respects, but specifically designed for running VMs. VMware ESX and Xen are two examples of virtualization solutions that follow the hypervisor approach. Both of these methods have their advantages that will be discussed in the following sections.

Despite the differences between the designs shown in Figure 3.1, in recent years the terms VMM and hypervisor have become more or less synonymous. For the remainder of this dissertation, we will use the general term *hypervisor* to refer to the software that runs the VMs in either approach shown in Figure 3.1.

3.2 The Task of the Hypervisor

The primary task of hypervisors is to multiplex or share the hardware between the virtualized guest OSes. That is, the hypervisor must allow multiple guests OSes to share the central processing units (CPU) (i.e., processors or cores), memory and hardware devices as if each OS were running exclusively on the hardware. Hypervisors multiplex a single hardware platform by presenting separate *virtual hardware* consisting of virtual CPU(s),

virtual memory and virtual devices (e.g., networks, display and disk drives) to each guest OS. Virtual hardware becomes the interface through which the guest OS can access the real hardware network and disks as needed.

In going beyond the basic requirement of multiplexing the hardware, Popek and Goldberg [47] articulated three essential characteristics of a hypervisor. Adams and Agesen [2] summarized these characteristics as:

1. **Fidelity:** Software on the hypervisor executes identically to its execution on hardware, barring timing effects.
2. **Performance:** An overwhelming majority of guest instructions are executed by the hardware without the intervention of the hypervisor.
3. **Safety:** The hypervisor manages all hardware resources.

As emphasized by the "Performance" characteristic, when running a guest OS, most instructions can and should be executed directly by the CPU. A key challenge in this regard is what are called *privileged instructions* that must be executed by the guest OS kernels.

3.2.1 Privileged Instructions

Privileged instructions are specific instructions that are part of every instruction set architecture (ISA) and are used by the OS kernel to multiplex the CPU between executing applications, and to manage the hardware. Privileged instructions can modify important CPU registers and flags that control each application. Therefore, under normal circumstances, privileged instructions should not be executed except when running the OS kernel. For example, a register that points to a process' page table can only be modified by a privileged instruction. These instructions are privileged to ensure that the OS kernel maintains control of the CPU and prevents any application from corrupting the OS kernel or other applications.

The challenge in hardware virtualization is in properly handling the privileged instructions in guest kernels because the guests run in unprivileged mode so that the host OS or hypervisor can maintain control of the hardware resources. If guest kernels did not run in unprivileged mode then multiple guest OS kernels could change the CPU state (without each other knowing) by running privileged instructions, potentially corrupting each other or the host OS.

Only the host OS has the necessary permissions to execute privileged instructions. Host OSes are notified when any application (including a guest VM) executes a privileged instruction. The notification comes from a mechanism called a *trap*. If an application such as web browser, running normally in unprivileged mode, executes a privileged instruction, a trap will occur. A trap stops the executing program and switches control to the kernel which may kill the program that attempted to execute the privileged instruction. When running a guest kernel on the CPU, any privileged instructions that are executed will similarly generate a trap to the host OS or hypervisor since guests run in unprivileged mode. Guest OSes need to execute privileged instructions in order to run, but when they are running as a guest OS they cannot be allowed to do so directly to ensure protection of the host OS and other applications and VMs.

Handling privileged instructions of guest VMs is a fundamental challenge in virtualizing operating systems. One technique historically used in hypervisors to handle privileged instructions was called *trap-and-emulate*. Hypervisors that use trap-and-emulate respond to a guest OS trying to execute a privileged instruction by having the host OS emulate the behaviour of the privileged instruction on behalf of the guest OS and allowing the guest to continue at the next instruction. By emulating the effect of the privileged instruction, say updating the page table pointer, the guest OS can continue to execute and the hypervisor maintains control (i.e., “Safety” characteristic from Popek and Goldberg [47]). While the trap-and-emulate approach is fairly straightforward, it has a drawback in that every trapped instruction pays a performance penalty as the host OS must:

1. take over execution from the guest OS,
2. emulate the instruction, and
3. resume execution of the guest OS.

Fortunately, privileged instructions are relatively rare and as long as the majority of instructions can still be executed directly by the CPU, performance will remain acceptable.

3.2.2 The x86 Architecture

Since the 1980s, the Intel 8086 architecture and its descendants have increasingly dominated the personal computing and server market. The x86 architecture, as it is generally called, presents a challenge to trap-and-emulate designs. The problem is that some privileged instructions when executing in unprivileged mode do not generate traps, but rather

the instructions are simply ignored. Therefore, the trap-and-emulate model cannot be used with x86.

Over time, *binary translation* and *paravirtualization* emerged as alternatives to trap-and-emulate. Binary translation solves the trapping problem by rewriting the binary code of the executing guest OS and replacing the privileged instructions with callouts to the hypervisor. VMware and VirtualBox are two well-established hypervisors that used binary translation. Binary translation has an advantage in that it can virtualize practically any x86-compatible operating system because privileged instructions are translated on-the-fly at runtime. Binary translation does incur overhead as the executable of the guest OS must be scanned for privileged instructions. Despite some drawbacks, binary translation was a successful method that allowed the x86 architecture to be virtualized [2].

3.2.3 Paravirtualization and Xen

An alternative to binary translation is *paravirtualization*. The Xen project [7] is the best known example of paravirtualization. Paravirtualization breaks the "Fidelity" characteristic of Popek and Goldberg because the guest OS must be modified and recompiled and therefore cannot execute "identically to its execution on hardware".

Paravirtualization requires changing the source of the guest kernel to replace any privileged instructions with callouts to the hypervisor and recompilation of the kernel. Paravirtualization has different trade-offs to binary translation. For example, paravirtualization requires different modifications for each guest OS (i.e., the modifications necessary for Linux will be different than for those of FreeBSD or Solaris). Moreover, closed-source operating systems such as Windows require the cooperation of the proprietors of the OS to make the necessary modifications. The advantages of paravirtualization include avoiding the overhead of translating guest instructions on-the-fly as occurs with binary translation, and avoiding the overhead of trapping with a trap-and-emulate approach, since the instructions requiring trapping are translated ahead of time.

The release of the Xen project in 2003 was a significant achievement in paravirtualization. Xen demonstrated that paravirtualization could achieve near-native performance on the x86 architecture [7]. Given that Xen was open-source, it also provided a cost-effective, high-performance solution to using virtualization at a large scale.

An illustration of Xen is shown in Figure 3.2. As mentioned previously, Xen follows the traditional hypervisor model (see Figure 3.1 (b)) that runs a hypervisor layer specifically designed to run guest VMs. As shown in Figure 3.2, the Xen hypervisor runs directly on

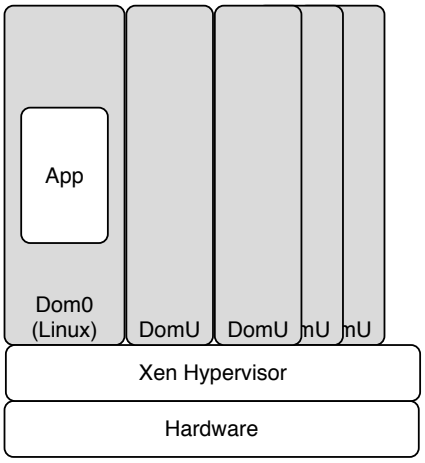


Figure 3.2: The Xen Paravirtualization Model

the hardware. The Xen hypervisor is a microkernel operating system [23] and involves a much smaller code base than a fully-featured host OS (as is needed in Figure 3.1 (a)).

When booted, the Xen hypervisor creates *domains* that run atop the hypervisor. Each domain is a separate guest OS. The most important domain is Dom0 which manages all other guest domains. Dom0 must be a Linux guest and is always running while the hypervisor is running. Dom0 runs the *Control Plane Software* that allows the creation of additional domains, so-named DomUs, that can run any operating system supported by Xen. Dom0 is also the primary conduit for DomU input/output (I/O) since all DomU I/O (e.g., network, disk) passes through Dom0.

Dom0 manages the DomU domains and provides access to devices such as the network and disk drives via virtual devices in order that the DomUs can perform I/O. Xen employs a split-driver model for the virtual I/O devices running in the DomU guests. The split-driver model involves splitting device drivers between *front-end drivers* that run in the respective DomUs and the *back-end drivers* that run in Dom0. DomU front-end drivers communicate with the back-end driver when they want to use a device such as the network card. It is the back-end drivers that multiplex a single device such as network card across all the DomUs that need access to the network. The effect of the split-driver model is that all device activities need to be communicated to Dom0 and the results be transmitted back to the front-end driver.

While the split-driver model achieves its goal of multiplexing devices between multiple domains, it has some drawbacks. In particular, split drivers have a performance over-

head [61] as all network and disk traffic must pass through Dom0. Secondly, split drivers create a security issue. Since all network and disk traffic must travel through Dom0, the Linux OS running in Dom0 is a possible source of attack that could compromise the network or disk traffic of the DomUs.

Since its release, Xen has been widely used in both the academic and industrial community due its high performance and support from major Linux distributions such as RedHat Linux.

3.2.4 Redefining Paravirtualization

After the release of Xen, the term paravirtualization broadened beyond the meaning of modifying an operating system to remove privileged instructions. Recently, the term has expanded to include not only modified operating systems but any virtual hardware or software (e.g., a bus or device) in a VM that has been modified or designed exclusively for virtual environments. A paravirtualized device does not have to have an equivalent implementation in real, physical hardware because it is designed to run only in VMs. Xen's split-driver model would be considered a paravirtualized design since its design only applies to the Xen hypervisor.

One particular paravirtualization approach that falls into this new, broader category is virtio [52]. Virtio is a device model that was specifically designed for VMs. Most hardware devices that hypervisors emulate are based on real, common hardware devices. For example, the well-known Intel e1000 ethernet network interface card (NIC) is emulated as a network device in most hypervisors for the simple reason that kernel drivers already existed for the e1000 in most OSes. However, emulating the behaviour of hardware can be inefficient. Virtio created a model to allow for simple and efficient virtual devices that could be standardized across all hypervisors (e.g., QEMU/KVM, Xen, VMware, etc). Virtio is general enough in its design to support a broad spectrum of devices including network, disk, serial and other devices. Since virtio devices were a new interface, they required new guest OS drivers to be written to support them. However, given that the different devices share a common virtio transport mechanism, less code needs to be written for new virtio devices.

To date, virtio drivers have been added to the Linux kernel and Windows drivers are also available. As will be discussed in Section 4.13.1, we had an experimental implementation of Nahanni that built upon the virtio framework.

3.2.5 Hardware-supported Virtualization in the x86 architecture

As the solutions of binary translation and paravirtualization allowed hardware virtualization on the x86 ISA, the use of virtualization grew in both the desktop and server environments. Due to the growing popularity of virtualization, the two major x86 vendors, Intel and AMD, both announced extensions to the x86 ISA to make virtualizing x86 easier, by not requiring binary translation or OS paravirtualization. Intel named their extensions *Virtual Machine eXtensions* (VMX) and AMD named their extensions *Secure Virtual Machine* (SVM). The respective extensions differed from one another in their design and are not compatible. The first Intel CPUs with VMX shipped in late 2005. The first AMD CPUs with SVM shipped in the middle of 2006. Briefly, the new instructions enable easier creation and control of virtual machines.

At the time of writing, nearly all current x86 CPUs are shipped with hardware virtualization support (some low-power mobile chips and low-cost server chips are the exceptions). The addition of hardware virtualization support made implementing x86 hypervisors much simpler by eliminating the need for binary translation or kernel paravirtualization. In 2006, within a few months of the hardware extensions being released, new hypervisors began to appear, such as Parallels [44], that relied on the hardware extensions. The Xen hypervisor also added support to make use of hardware extensions as an alternative to paravirtualization. Another project that also began in 2006 as a result of the new hardware extensions was the Kernel-based Virtual Machine project, or KVM, which is the platform for this work.

3.3 QEMU and KVM

To understand KVM, one must first understand QEMU [8]. QEMU is a computer hardware emulator that uses dynamic translation to execute a particular ISA on top of a different ISA (e.g., QEMU can emulate SPARC on top of PowerPC). QEMU supports emulation of numerous ISAs including ARM, SPARC, MIPS and many others. QEMU can be run on a number of different ISAs such as x86, SPARC, PowerPC, MIPS, etc. QEMU is a full system emulator in that it creates virtual CPU(s), RAM, and devices to execute a guest OS. The emulated OS that is run on QEMU typically runs much slower than it would on real hardware because of the overhead of emulation, specifically the translation of the guest ISA to the host ISA. QEMU is still useful despite the overheads. QEMU is commonly used for development work when real systems are rare, low-powered or difficult to develop for and debug software on. For example, QEMU is used for the smartphone development emulator

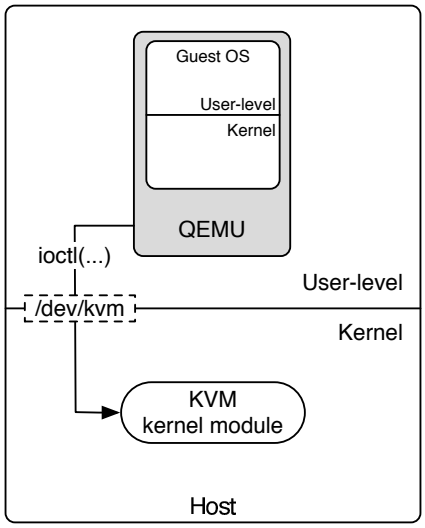


Figure 3.3: The Basic QEMU/KVM Architecture

in the Google Android project.

Aside from emulating different architectures, QEMU can also be used to execute the x86 ISA on top of x86 hosts without requiring hardware support, albeit slower than other binary translation systems such as VMware due to overheads resulting from QEMU's multi-architecture emulator design.

3.3.1 The KVM Project

The Kernel-based Virtual Machine (KVM) project [29] began in 2006 to create an efficient hypervisor based upon QEMU. As part of the KVM project, the multi-architecture emulator support in QEMU was modified to accelerate the x86-on-x86 case by using the new virtualization extensions on x86 processors. The new x86 virtualization instructions must be issued in privileged mode (i.e., the host kernel), necessitating the addition of a kernel module that is the significant contribution of the KVM project. Figure 3.3 illustrates the basic architecture of QEMU/KVM when running a single guest OS. KVM follows the hypervisor model illustrated in Figure 3.1 (a) and requires Linux as the host OS. The two-part system of QEMU at user-level and KVM in the kernel is why the term QEMU/KVM is used to describe the system as a whole, although KVM is more commonly used for brevity. As shown in Figure 3.1, the modified QEMU communicates with the KVM module via a device file, `/dev/kvm` that is created when the KVM module is loaded into the kernel. The single KVM module can support multiple VM guests.

QEMU/KVM VMs can be launched from the command-line. For example, the following command-line will start a QEMU/KVM VM with 2 CPUs (`-smp 2`), a virtual disk that will contain the OS and applications (`-hda disk.img`) and 4 GB of RAM (`-m 4G`).

```
qemu-system-x86_64 -smp 2 -hda disk.img -m 4G
```

The above is clearly a very simple but useful example. The command-line arguments to QEMU are extensive and specify a wide array of devices including network devices, graphics and USB devices.

In brief, QEMU/KVM runs a modified QEMU process at user-level that creates and manages a VM in much the same way the original QEMU system did. However, instead of emulating CPU execution, this modified QEMU relies on the KVM kernel module to run the VM natively on the CPU. The kernel module issues the virtualization instructions to achieve accelerated performance. It is the QEMU user-level process that allocates memory that will serve as the virtualized guest's RAM. QEMU also creates the virtual devices that the guest OS will use. In turn, the KVM module will setup the execution of the VM via the x86 virtualization instructions and actually trigger the VM's execution. The QEMU process and KVM module will pass control back and forth as the VM executes. With the removal of the QEMU emulation system, the majority of the guest's x86 instructions execute directly on the hardware leading to significant improvement in performance (i.e., "Performance" characteristic of Popek and Goldberg [47]).

Virtual devices, including disks, display and network devices, are still managed by the QEMU process at the user-level. When the VM accesses a device, control will pass to QEMU to perform the particular task on the virtual device, such as sending a network packet. Once this task is completed, QEMU will notify the KVM module to continue execution and the VM will be resumed.

At the time of KVM's release, hypervisors such as Xen and VMware ESX (Figure 3.1 (b)) that run directly on the hardware were the growing trend. The KVM design was motivated by the view that the task of the hypervisor is similar to the task of any operating system in that they both provide device and resource sharing and isolation between processes/VMs. Moreover, the design trade-offs of process creation, scheduling and memory management are much the same in operating systems as they are in hypervisors. Therefore, building KVM on top of Linux leveraged all the work that had been done by the Linux community to make Linux a solid, yet flexible, foundation. With KVM, the host OS that manages the guest VMs is the same Linux operating system that users and administrators are familiar with. The only addition to a standard host kernel is the KVM kernel module. KVM VMs

are regular Linux processes that share the CPU, memory and hardware devices with the other processes (guest VMs or otherwise) running on the same host OS.

Aside from virtualization, the other research topic that Nahanni explores is interprocess communication which will be discussed in the following section.

3.4 Interprocess Communication

Interprocess communication (IPC) is a fundamental topic in computing in addition to being a well-studied area of research. In general, IPC encompasses the mechanisms that allow threads and processes to exchange data and synchronize their execution. In general, there are two kinds of IPC, stream data and shared data. The distinction between the two lies in that stream data passes entirely from one process to another where as shared data is simultaneously available to multiple processes. Operating systems typically support several different IPC mechanisms including, but not limited to, pipes, sockets, shared memory and signals. The variability in mechanisms is a result of the variety of communication needs that processes and threads may have. For example, communicating processes may be geographically distant as in the case of web browsers and web servers which would require stream data as the data may pass across the Internet. Alternatively, communicating processes may run on the same physical machine in the case of high-performance parallel programs which could make use of shared-data IPC. The nature of the application(s) that requires IPC will strongly influence which mechanisms can be used and which is best suited.

The research that has investigated IPC mechanisms has explored trade-offs of different characteristics such as latency, bandwidth, security and ease of programming. Since Nahanni provides a shared memory IPC mechanism with a goal of demonstrating high-performance, we will discuss previous research that has a similar focus.

In terms of providing high performance, identifying and eliminating extraneous copying of data during IPC has been examined in several contexts. Brustoloni and Steenkiste [9] characterized the trade-offs in passing data between processes and the OS kernel. The authors emphasize the effects of buffering semantics in different IPC mechanisms on performance and point to unnecessary copying of data as being one source of overhead that leads to poor performance. The insight into eliminating copying of data served as a motivation for our work to provide a shared-memory interface between co-located guest VMs that reduces or eliminates copying of data.

Fbufs [16] are an operating system mechanism for efficient data transfer across protec-

tion domains in shared memory systems. Fbufs use page remapping and memory sharing to eliminate data copying and improve performance. Fbufs employ a concept of *originator* and *receiver* domains. An originator domain allocates a series of buffers and sets permissions so the intended receiver can map the same range of memory into its own address space. Once mapped the receiver can receive objects from the originator domain without incurring intermediate copies. Fbufs were implemented in the Mach microkernel. A mechanism like Fbufs was important for a microkernel like Mach due to the modular nature of a microkernel that may incur numerous copies as data is transferred between cooperating servers.

Beltway Buffers [13] are in-kernel mechanism for Linux that uses pre-allocated, long-lived, shared rings for data movement for all IPC mechanisms (sockets, pipes, file systems, etc) as well as networking. By keeping all data in shared rings, data copies within the kernel are reduced. Context switches are avoided by reusing the ring buffers to avoid continually allocating and de-allocating kernel memory. The concept is that once data enters the kernel, either from an application or from a device such as a disk, that data will be placed into a ring buffer, called a DBuf. Data is copied into the DBuf only once and is then accessed from there by all kernel subsystems that need to access that data. Throughput and latency are improved by eliminating copies of the data between subsystems within the kernel. Beltway buffers reside entirely in kernel space, so their use is completely transparent to applications. Despite reducing copies in moving data through the kernel, data must still be copied from applications running in user-level to ensure protection is maintained between applications and the kernel.

Gamsa *et al.* [20] explored optimizing IPC in the context of a microkernel kernel architecture in which communication between client and server processes is critical to overall performance. More specifically, they explore IPC in the context of shared-memory multiprocessors with non-uniform memory access. Their work is based upon the Protected Procedure Call (PPC) model that allow clients to execute procedure calls within the address space of the servers that the clients need to communicate with. PPCs are an alternative to message-passing systems that send stream data between client and server processes in a microkernel OS. Executing a procedure call within the server's address space eliminates the need to send and receives messages to achieve the same result. Moreover, copying of data between address spaces that requires memory-to-memory copies and locking can be eliminated to improve performance. The specific work of Gamsa *et al.* was targeted for the Hurricane operating system and did show improved PPC call overhead through various

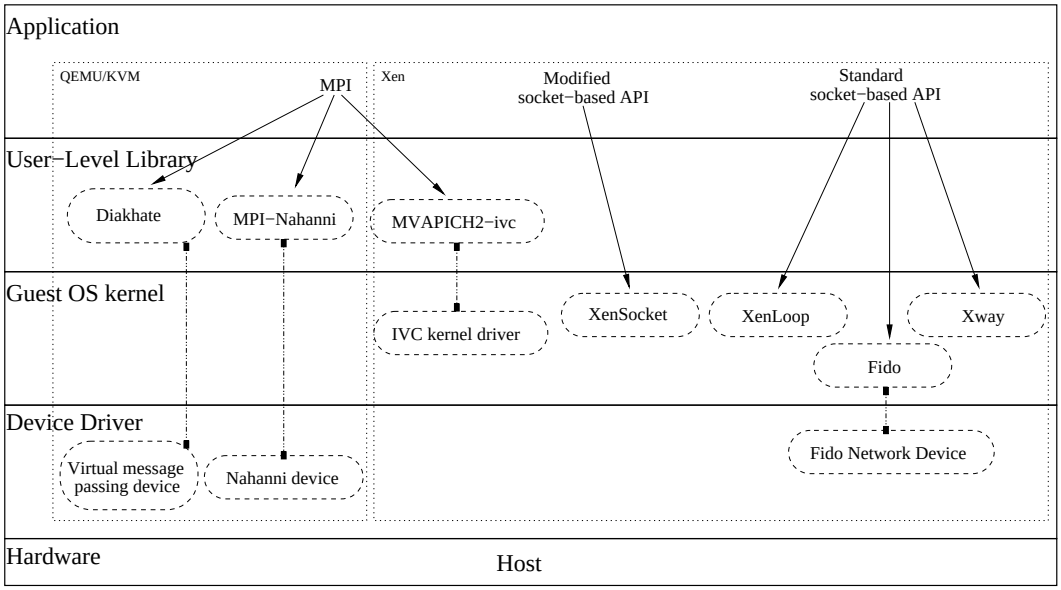


Figure 3.4: Architectural Layers of Inter-VM IPC Systems (adapted from Ke [26])

optimizations.

While the concept of PPCs are certainly foreign to the accepted wisdom of protected address spaces, the motivation for them is consistent with our goal of improving inter-VM IPC by reducing memory-to-memory copying.

Architecturally, Nahanni is a significant departure from previous shared-memory IPC approaches for non-VM environments. For example, much of the earliest work with shared-memory IPC was in the context of process-to-process communication [20], or user-level process-to-kernel communication [60]. However, with traditional, non-virtualized IPC only a single OS kernel is involved in the communication, well-known techniques such as kernel-level blocking for synchronization can be used. But, Nahanni involves multiple, independent, guest OS kernels in different VM instances, therefore the traditional synchronization mechanisms are no longer available. So far, our approach has been to use lock-free and spin-based synchronization techniques, as well as inter-VM based mechanisms such as virtio-serial [26]. We do not claim any improvements of Nahanni over the previous work with process-to-process and process-to-kernel shared-memory IPC, but are merely pointing out the architectural differences.

Now that we have highlighted IPC research related to shared memory, in the next section we will discuss previous efforts to improve IPC between VM and their associated applications.

	XenSocket	Xway	XenLoop	Fido	IVC	Diakhaté	Nahanni
OS mod	module	patch	module	patch	module	module	module
Hypervisor	xen	xen	xen	xen	xen	kvm	kvm
Hypervisor modified	no	no	no	yes	yes	yes	yes
Binary compatible	no	yes	yes	yes	no	no	no
MPI support	no	binary	binary	binary	library	library	library
Data Format	stream	stream	stream	stream	stream	stream	stream & structured
Details		TCP only		trust		2 vm limit	

Table 3.1: Comparison of Inter-VM High-Performance Communication Mechanisms

3.5 Inter-VM IPC

IPC between VMs, or inter-VM IPC, has been explored from several angles in previous research. First, we can view the related work in terms of architecture and layers of abstraction (Figure 3.4, adapted from Ke [26]). The two major architectural trends are: First, as represented by Nahanni, and MPI-Nahanni (Section 5.6) specifically, inter-VM IPC can be implemented without any changes to the guest OS kernel. Second, as represented by XenLoop [61] and other systems, the guest OS kernel can be modified with new datapaths.

On the one hand, modifying the OS kernel can potentially make the fast IPC completely transparent to applications. For example, XenLoop is binary compatible with existing applications, not even requiring a re-compilation. On the other hand, changing the OS kernel is error-prone, can introduce new overheads and bottlenecks (Section 5.6) for many applications (i.e., not just applications needing fast inter-VM IPC), and the changes must be updated as new versions of the OS kernel are released. With Nahanni and MPI-Nahanni, updating a user-level library (if necessary) is easier than OS kernel updates, and many applications access IPC via libraries such as MPICH2/MPI [26] and the memcached client library [63, 22] anyways. A small device driver is needed with Nahanni, but the driver is only used during IPC initialization (i.e., does not change any existing OS datapath for the communication itself), and the standard kernel-to-device-driver interface changes infrequently, which minimizes the update/maintenance problem.

Second, we can view the related work in terms of functionality and mechanisms. Table 3.1 summarizes the different inter-VM communication mechanisms that will be discussed below and compares them based on their supported hypervisor as well as whether modifications are required to the hypervisor, guest OS and/or guest applications to use them.

The row labels in Table 3.1 deserve some discussion to understand the differences between the compared mechanisms.

OS modification All systems discussed require changing the guest operating system to support the new transport mechanism. Depending on the extent of the changes necessary, the changes may be contained in a single kernel module. Kernel modules are advantageous in that they are compiled separately and so avoid recompiling and reinstalling the kernel. A second advantage of kernel modules is that they can be loaded at runtime without requiring a reboot. Mechanisms that are contained within a kernel module are labelled as *module* in Table 3.1. If the changes are too extensive they may require changes to multiple OS subsystems. In the later case, the changes require a patch to be applied to the kernel source, recompilation and installation of the new kernel, significantly increasing the installation effort. Mechanisms that require patching are labelled as *patch* in Table 3.1.

Hypervisor The two open-source hypervisors that have been used to investigate inter-VM communication are Xen and KVM.

Hypervisor Modified Some optimizations require modifying the hypervisor to support them while others do not. For example, Xen has supported sharing pages between domains for some time via a mechanism called the grant table. Mechanisms that make use of the grant table to enable shared-memory communication do not require modifications to the Xen hypervisor. As discussed, KVM comprises two parts: QEMU at the user-level and the KVM kernel module in the host kernel. For Nahanni, all modifications are restricted to QEMU; the KVM kernel module is not modified.

Binary Compatible When designing an optimization, a key design choice is whether applications can take advantage of the new mechanism without any changes to the applications or libraries. An application that can use a mechanism without requiring any modification is said to be *binary compatible* (since the executable binary application need not be changed). Applications that either require a new API or code modifications are not binary compatible. Typically optimizations that exist entirely in the kernel are binary compatible as user-level applications are essentially unaware of their existence.

MPI Support MPI is the arguably the most common communication library in high-performance applications. Accelerating MPI performance between VMs could benefit numerous

HPC applications. MPI is an abstraction and typically compiled as a linked library. If an optimization can be contained within an MPI library, an application would simply need to be re-linked, not recompiled, to take advantage of the optimization.

Data Format Different IPC mechanisms are designed for different use cases. Stream data, mentioned previously, is the most common communication model for IPC. Optimizations labelled as *stream* only support stream-based communication, that is basically message passing. An alternative to stream data is structured data which requires shared memory that is simultaneously accessible by multiple applications. Structured data is stored in shared memory rather than just transferred across shared memory.

Details Additional details regarding the use of particular mechanisms are mentioned.

The use of inter-VM shared memory for communicating between VMs has previously been explored. The majority of inter-VM communication research has focussed on the Xen hypervisor [7] due its having been available since 2003 and being open source. Recalling that Xen uses the term "domain" to refer to VMs, the term "inter-domain" in the following discussions is equivalent to "inter-VM". To the best of our knowledge, no other research project has explored using shared memory between host and guest applications.

XenSocket [64] is an inter-domain communication system that provides one-way communication sockets between co-located Xen domains. XenSocket uses shared memory between the guest domains to transfer the data, but does not allow direct access to the shared memory from user-level. Data is sent and received using the standard `send()` and `recv()` calls typically used in Unix sockets programming. XenSocket is not binary compatible with existing applications, that is applications must be modified, albeit minimally, to use XenSocket. One important insight that XenSocket highlights is that the page-flipping mechanism that was available in Xen was not optimal for inter-VM performance even though it reduced explicit copying. Page flipping is a mechanism for IPC that remaps pages between process address spaces (or in the case of Xen, between domains) rather than copying the data, thus saving one copy operation. The fact that XenSocket achieved improved performance over the standard page-flipping mechanism showed that the overhead of page-flipping is significant and may not provide the best solution. In the case of XenSocket, using fixed shared memory between domains achieved better performance than page flipping.

XenLoop [61] used shared memory to create a high-performance loopback mechanism to speed network communication between co-located VMs. A shared-memory region that can be accessed by both VMs is created using a Xen-specific inter-domain shared memory

facility. Inter-VM network FIFO channels are created between communicating VMs that can pass messages through the channel with minimal copying. XenLoop is abstracted in the virtual hardware of the Xen domain meaning that no changes to existing applications, libraries or the front-end network device (recall Xen has a split device driver model) are necessary to take advantage of XenLoop. A potential drawback of the XenLoop design is that it does not expose shared memory to the guest OS or applications and still incurs copying through the network stack.

Fido [10] is another shared-memory optimization for Xen. Fido relies on trust between co-located VMs. To eliminate copies, Fido allows each guest to map all other guest domains' memory with read permission. The authors claim that in an enterprise environment cooperative mapping of all guests' memory is reasonable because the likelihood of a malicious guest is extremely low in a private, company-controlled environment. By mapping the sender VM's address space into the receiver's address space, the receiver can read data directly from the sender's address space, thus eliminating a copy. Similar to XenLoop, Fido implements a network device that uses the mappings to move data efficiently and without requiring modification of applications or libraries. Fido was also used to implement a block device interface.

Xway [28] is another inter-domain communication optimization for Xen. Xway chooses to maintain binary compatibility to support legacy applications and libraries rather than expose a new API. Xway intercepts packets above the TCP networking layer and passes those destined for co-located VMs through inter-domain shared memory without going through the network layer. Xway requires modifying the Linux kernel to support intercepting packets and only accelerates TCP traffic.

IVC [24], also for Xen, chooses to not provide binary compatibility, but creates an IVC user-level library that applications must be written to use. Similar to our evaluation of Nahanni, the evaluation of IVC includes creating an IVC-aware MPI library, *mvapich2-ivc*, that MPI applications can be linked against. Creating such a library allows MPI jobs to not require modification, but just re-linking to take advantage of IVC. IVC's design allows user-level memory in one guest to be shared with other guests. As with other mechanisms, IVC uses Xen's grant table mechanism to achieve this. IVC focusses on stream-data benchmarks and does not explore structured data use cases. IVC also requires that shared memory be separately configured for each pair of peers.

As mentioned, the majority of research has focussed on Xen, but research involving the Linux/KVM project [29, 30] is increasing. KVM is a more recent project since it builds

upon hardware virtualization extensions in x86 architectures that were not available until 2005.

Diakhaté *et al.* [15] investigated an inter-VM shared-memory system for use primarily with MPI. The system uses a virtio-based device added to each guest to access the shared memory. Multiple guests for their system were created using the `fork()` system call to facilitate easy sharing of memory. Requiring all cooperating guests to be created with `fork()` eliminates the possibility of running differently configured guests. The system achieved near native performance, but experiments were restricted to a small subset of MPI commands. In principle, this mechanism could support host/guest shared memory, but it was not mentioned by the authors.

VMware had a shared memory mechanism available for desktop virtualization called VMCI [59]. VMCI allowed applications to create named shared-memory regions that are shared between guests. For reasons that are not entirely clear, VMCI was deprecated in favour of a socket mechanism. We are not aware of any research involving VMCI shared memory.

Given the variety of research that has been conducted on inter-VM communication it is worth re-iterating that to the best of our knowledge no projects have explored sharing memory to the user-level as Nahanni allows. As well, Nahanni is the only mechanism that allows host-guest sharing since the shared memory can be accessed by applications that are running directly on the host OS.

So far in this chapter we have focussed on hardware virtualization and IPC as they are the two fundamental concepts that form the basis for Nahanni. However, to give a complete picture of Nahanni's design and function there are also some concepts related to the specific implementation of Nahanni as part of QEMU and the Linux kernel that must also be discussed.

3.6 The Linux Kernel

The particular implementation of Nahanni that is presented in this work is based upon Linux at both the host and guest levels. To be clear, Nahanni's design is OS agnostic at the guest level, but for the experiments shown later Linux was used. At the host level, Nahanni builds upon KVM which is a Linux-specific system, but the design of Nahanni could be ported to other host OS/hypervisor pairings. In this section we will elaborate on the Linux-specific elements that are part of the Nahanni design.

3.6.1 Device Drivers

In Linux, like most modern operating systems, peripheral devices are supported by device drivers. Device drivers are code modules the host OS uses to interact with the device. Device drivers implement an abstraction so that the kernel can communicate easily with devices. By encapsulating the implementation details of a peripheral within a driver, and exposing a standard interface to the kernel, a device driver helps to make the OS more modular and stable.

Devices are generally separated into three broad categories: block devices, character (called "char" for short) devices and network devices. A device driver identifies itself as either a block, network or character device during configuration. As their name reflects, block devices make data from their device available via *blocks* which have a fixed and defined size. Block devices send and receive these fixed-size blocks of data to their respective devices. Given that block devices have a defined size they allow random access to the data via the driver. The most common block drivers are file systems drivers. Character devices differ from block devices in that they send and receive streams of bytes to their devices such as serial ports or console devices. Since data is streamed through the device to say a display device, and not stored, character devices typically do not support random access.

Drivers are written for a specific bus type and implement a set of functions that the kernel will use to configure and operate a device. In the case of Linux, the *probe()* and *remove()* functions are the two necessary functions provided by a device driver that configure and deconfigure the device, respectively.

Character devices also typically implement the standard POSIX file system operations such as `read()`, `write()`, `ioctl()` and so on. A device will also have a corresponding entry in the /dev file system, `/dev/foo` for example. By performing file operations on `/dev/foo` an application can control the device associated with the `/dev/foo` device file. One supported character device operation that is of particular importance to Nahanni is `mmap()` that can allow a region of device memory to be mapped to user-level. Given Nahanni's goal to provide zero-copy data sharing between VMs, being able to map the shared memory to user-level is essential to avoid the copying of data from the user-level to the kernel.

3.6.2 PCI and UIO

The Peripheral Component Interface (PCI) is a device bus that provides access to peripheral devices that are plugged into a computer's motherboard. The PCI bus is the most commonly used bus on desktop and server computers today giving the computer access to its graphics,

sound and network devices.

PCI is more than simply a bus, but defines how the OS should interact with PCI devices. More recently, a new standard called *PCI Express* has been defined to provide better performance and give more flexibility than the original PCI implementation. However, PCI still remains a common interface for many devices.

In 2009, a new device driver model was added to the Linux Kernel called UIO and its goal was to move as much device driver code for PCI devices into user-level applications instead of having more code in the kernel. The observation that motivated UIO was that there are numerous devices that have similar and straightforward behaviour. That behaviour is reading and writing to a few registers and accessing memory on the device. Both of these operations can be performed at user-level if the device registers and device memory are mapped into user-level. Drivers that make use of the UIO interface still require some code to run in a kernel module, but that code is much simpler than a traditional device driver.

The UIO design fits well with Nahanni's design goals as it allows efficient access to devices from user-level eliminating traps to kernel-level which can be a source of overhead. UIO does require a small amount of driver code that runs in the kernel. The purpose of the kernel level code is to configure interrupts and enumerate memory regions that will be mapped into user-level. Any UIO device is accessible by user-level applications via a device file named `/dev/uioN` where `N` is an integer beginning at 0. The device file is created when the driver is loaded. Since the device file is part of the file system namespace, it can be protected with the necessary file permissions to limit access to authorized applications.

As of version 2.6.37, the Linux kernel currently ships with four UIO device drivers that are used for a variety of devices.

3.7 Concluding Remarks

This chapter has provided a background in the general concepts that are important in understanding Nahanni such as devices, IPC and the Linux kernel. Moreover, we have discussed the previous research that relates to Nahanni to express what is novel in the design and implementation of Nahanni. As much as possible we have taken the lessons from previous research in designing Nahanni. In the next chapter, we will provide a detailed description of the design and implementation of Nahanni.

Chapter 4

Design and Implementation

To this point, Nahanni has been discussed at a high level in terms of its function and goals: Nahanni is a user-level, shared-memory interface for virtual machines (VMs) that supports both stream data and structured data. In previous chapters, we have motivated some of the use cases of Nahanni and discussed the previous research and concepts that relate to it. It is now fitting to discuss the design and implementation of Nahanni at a detailed level and describe precisely how Nahanni allows multiple VMs to share memory.

4.1 Design of Nahanni

In the design of Nahanni, each piece was deliberately and carefully chosen. Figure 4.1 illustrates the three major elements implementing Nahanni:

1. **A POSIX shared-memory region on the host.** Nahanni uses host resources, namely POSIX shared memory as it exists in Linux, as the basis for sharing memory. No modification is required to the host OS or any of its existing kernel modules, including KVM.

2. **A modified QEMU that supports a new Nahanni device.** Nahanni requires adding a new virtual device, named "ivshmem" for inter-VM shared memory, to QEMU/KVM. Section 4.4 describes in detail the changes that were necessary to the user-level QEMU to support the device. These changes have been merged into the QEMU release as of version 0.13 from August 2010 [48, 33, 35]. Note that ivshmem is the virtual device implementation of Nahanni and the two terms will be used interchangeably in this chapter.

3. **A Nahanni guest kernel driver.** A new Linux kernel driver was created that can communicate with the ivshmem device. The kernel driver creates the interface to

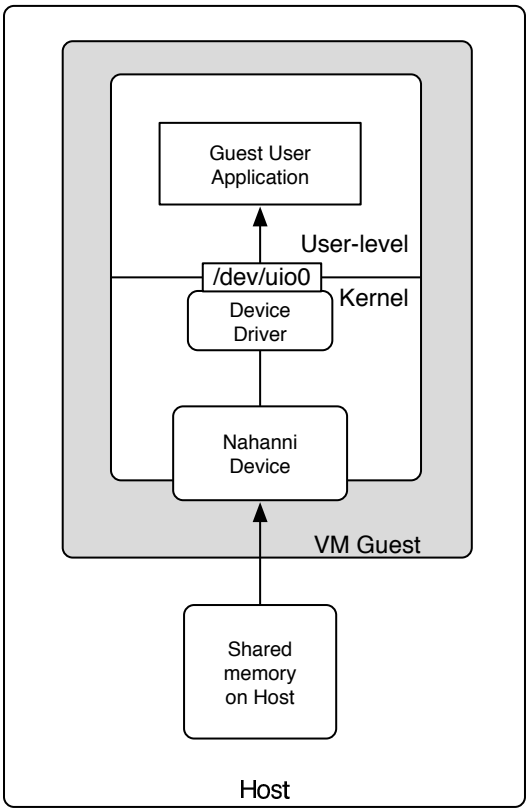


Figure 4.1: Nahanni's Software Stack

the guest user-level to give applications within the VM direct access to the shared memory. The device driver is available from a public source repository [40].

Although different approaches were possible, these three elements that implement Nahanni reflect our design goals in creating a shared-memory mechanism. These goals include providing an shared-memory mechanism that provides utility to those applications that use it while having no negative impact on applications that do not use it. With the above three components, VMs that do wish to share memory can do so without onerous installation effort. Furthermore, memory can be shared between host and guest applications or between applications in different guest VMs.

While the final implementation that we will describe may appear to be straightforward and obvious, in fact there were other compelling alternatives with strong advocates from the within QEMU/KVM development community. Some of the proposed alternatives may have provided equivalent functionality but are inferior from the point of view of orthogonality (e.g., optional Nahanni device), flexibility (e.g., UIO compatibility) and simplicity/adapt-

ability (e.g., raw shared memory).

In the next section, we will layout the alternatives and elaborate on the choices that were made.

4.2 Design Alternatives

Although the final form of Nahanni is straightforward, there were reasonable but more complicated designs that were considered, implemented, and eventually rejected. While a detailed discussion of related work was given in Chapter 3, here we sketch out some of the particular design alternatives not chosen for Nahanni to better understand the choices that were made.

First, Nahanni supports arbitrary structured data in shared memory instead of just stream data. Unlike the XenSocket [64], XenLoop [61], and IVC [24] mechanisms based on the Xen hypervisor, Nahanni is targeting a broader array of applications than just those that use stream data mechanisms, such as sockets. If Nahanni were only focussed on stream data, then the design could have abstracted Nahanni shared memory as an in-kernel socket interface, similar to XenSocket. However, such a design would not allow structured data and would require a robust programming interface that may restrict other uses. Instead, we have targeted low-latency, structured-data use cases as well as stream-based applications [63]. Structured-data use cases are those that *store* data in shared memory, as opposed to stream-based use cases that simply *transfer* data across shared memory. The Fido system [10] for Xen targets both stream data and block (aka structured) data, but, unlike Nahanni, Fido does not currently handle storing data directly in shared memory or synchronizing via shared memory.

Storing data and synchronizing in shared memory can allow applications that rely on structured data, such databases or in-memory caches, to take advantage of Nahanni. In particular, we have explored the use of Nahanni to extend the well-known memcached [37] to cache key-value pairs in Nahanni memory, and then allow multiple VMs to access the cache. As well, in Section 5.4, we show that Nahanni is up to 8 times faster than other mechanisms for stream data, with the appropriate code changes.

Second, Nahanni uses the peripheral component interface (PCI) standard for peripheral devices. Supporting shared-memory IPC mechanisms in QEMU/KVM has other design alternatives. Specifically, one could build a shared-memory mechanism within the virtio subsystem, which negates the need to use the PCI bus. In fact, on the insistence of the

QEMU/KVM open-source community, an earlier implementation of Nahanni was based on virtio (see Section 4.13.1). However, after a great deal of work, we (and the community) ultimately decided that the number of changes required to virtio to support Nahanni was too large. Therefore, we abandoned the virtio implementation in favour of the current PCI-device-based approach to Nahanni. Designing Nahanni as a PCI device required fewer code changes to QEMU and more easily supports use by other OSes that support PCI such as Windows. Using PCI also allowed the guest driver to be based upon the UIO driver framework that allows applications greater control over their use of the Nahanni device.

The failed experiment with a virtio-based implementation of Nahanni is a classic example of how a community's first instinct can be wrong. One the one hand, virtio was (and still is) considered to be the future of high-speed data-transfer mechanisms within QEMU/KVM. That is why virtio and the related concept of vhost are key points of comparison in Chapter 5. And, it seemed, any new mechanism that might improve data transfers should (incorrectly) be implemented within that framework. On the other hand, virtio is based on direct memory access (DMA) or transfer-engine-like semantics, which is fundamentally different from the classic shared-memory semantics of Nahanni. Furthermore, given a region of Nahanni shared memory, it can be used as part of an efficient data-transfer strategy (Section 5.3.2). But, it is not possible to efficiently share data if a DMA engine is all that is available. Our lessons will be presented to the QEMU/KVM community [34].

Third, Nahanni exposes shared memory up to the user-level within VMs. Reading and writing to Nahanni memory requires no intervention from the guest kernel or hypervisor. This user-level design avoids crossing two protection barriers: from guest user-level to guest kernel and from guest VM to host when accessing the shared memory. VM exits are expensive because they result in switching control to the host OS, a sort of heavyweight context switch. The alternative to the user-level design would hide Nahanni memory within the guest kernel or within the virtual hardware as an inter-VM interconnect similar to XenLoop. If we had chosen to hide shared memory below the guest kernel level, then guest kernel traps would be necessary for each communication. Storing data and using synchronization primitives directly in shared memory without trapping into the kernel can have a significant performance impact. In Section 5.5, we discuss the GAMESS computational chemistry application which uses shared memory for data sharing and for synchronization via semaphores (see Figure 5.5). Without shared memory, GAMESS can use stream data mechanism (i.e., sockets and the Message-Passing Interface (MPI)), but we show how using Nahanni results in up to a 30% improvement in performance.

Exposing shared memory to the user-level requires the application to be modified to use it, however we are interested in exploring those modifications and the trade-offs involved. As a research project, Nahanni is designed to allow users to explore both stream-based and structured-data applications that may benefit from shared memory.

Now that we have explained our design decisions at greater depth, we will provide a detailed explanation of three main components that comprise Nahanni from Figure 4.1.

4.3 Component 1: POSIX Shared Memory

Linux, like most operating systems, supports sharing memory between processes as a method of IPC. There are two well-known interfaces that support shared memory in Linux: POSIX and System V (SysV). Since Nahanni uses POSIX shared memory, only the POSIX interface will be described in detail, however aside from the initial setup, POSIX and SysV shared memory can provide the same shared-memory semantics. Our discussion will also focus on the C library interface for POSIX shared memory since QEMU/KVM is implemented in C.

A Linux process can create, modify and destroy shared memory in much the same way that it can disk files. We will use the phrase *shared-memory object* to refer to memory that is shared to distinguish them from memory-mapped files which can also be used for sharing. While shared-memory objects are accessible via the file system namespace, they do not use any stable storage for backing, so they differ in that way from traditional memory-mapped files. For example, shared-memory objects are not persistent across reboots whereas traditional memory-mapped files are.

The C library interface for accessing POSIX shared memory is similar to the interface for accessing disk files. There is a difference in syntax however. POSIX shared-memory objects are accessed via `shm_open()` and `shm_close()` system calls rather than `open()` and `close()` calls that are used for disk files.

Figure 4.2 shows the code that was added to QEMU to open the shared-memory object using the `shm_open()` function on lines 3 and 10. While this code happens to be taken from Nahanni, there is nothing Nahanni-specific about it. Code fragments from other programs that use POSIX shared memory would be similar.

Another difference between shared memory and disk files is that `read()` and `write()` system calls cannot be used with shared memory. Shared-memory objects are to be mapped into the process' address space using the `mmap()` system call and accessed like an array or

```
1       /* try opening with O_EXCL and if it succeeds zero the memory
2        * by truncating to 0 */
3       if ((fd = shm_open(s->shmobj, O_CREAT|O_RDWR|O_EXCL,
4                   S_IRWXU|S_IRWXG|S_IRWXO)) > 0) {
5         /* truncate file to length PCI device's memory */
6          if (ftruncate(fd, s->ivshmem_size) != 0) {
7             fprintf(stderr, "ivshmem: could not truncate shared file\n
                  ");
8          }
9
10      } else if ((fd = shm_open(s->shmobj, O_CREAT|O_RDWR,
11                  S_IRWXU|S_IRWXG|S_IRWXO)) < 0) {
12         fprintf(stderr, "ivshmem: could not open shared file\n");
13         exit(-1);
14
15      }
```

Figure 4.2: Nahanni: Opening of the shared-memory object in QEMU

pointer-based structure (i.e. with load and store operations).

Within QEMU, after the POSIX memory object has been opened with `shm_open()`, an `mmap()` system call is used to map the memory object into the QEMU process' address space:

```
ptr = mmap(NULL, s->ivshmem_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
    0);
```

To help understand the above function call, we will explain the arguments. The first argument, `NULL`, indicates that the memory can be mapped to any available address range within the process' address space. The second argument, `s->ivshmem_size`, passes the size of the memory to be mapped. The next two arguments to `mmap()` specify protection modes (`PROT_READ|PROT_WRITE`) and configuration (`MAP_SHARED`) for the memory and are similar to most mmap calls in general. The second last argument, `fd`, is a file descriptor returned from the `shm_open()` call that indicates the memory object to be mapped. The final parameter is the offset into the shared memory object to map from. Nahanni always maps from the beginning of the region, so this parameter is 0.

The above `mmap()` call shows the essential mechanism of how a QEMU process gains access to the shared-memory region on the host. If two or more QEMU processes map the same shared-memory object (i.e., using the same POSIX shared-memory object in the file system namespace passed to mmap via the `fd` argument), those processes can communicate and share data via loads and stores to the shared-memory region.

This section has explained in detail how POSIX shared memory, as it exists in Linux,

is accessed to create a shared-memory region on the host by the QEMU application. In the next section, the additional changes made to QEMU will be described to illustrate how QEMU, via the ivshmem device, allows a guest OS to access the POSIX shared-memory region from the host.

4.4 Component 2: A Modified QEMU

The modifications that were necessary to QEMU comprised adding a new virtual device to the QEMU/KVM virtual hardware support. No modifications are necessary to the KVM kernel module and so changes are restricted to the user-level QEMU application.

QEMU/KVM, like all virtualization solutions, provides a VM that can execute an unmodified OS. Real computers perform input and output (I/O) via devices such as network cards, disk drives and video displays that are controlled by kernel drivers for those devices. Virtual hardware must perform I/O in the same way, via virtualized devices implemented to standard interfaces (e.g., PCI), to support unmodified OSes. So in designing a shared-memory interface between VMs, it is natural to implement it using a standard device interface, such as PCI, that QEMU/KVM supports.

A major goal of this work was to have an impact on the broader QEMU/KVM community and having our changes merged into the QEMU/KVM project was key to achieving this goal. Since our changes add a new virtual device, they pertain specifically to QEMU, the user-level portion of QEMU/KVM that manages the hardware and memory of the VM (see Chapter 3).

The implementation of Nahanni with respect to the code changes necessary to QEMU can be best understood as three parts:

1. a new virtual PCI device: *ivshmem*
2. a new memory-allocation method for QEMU
3. a new command-line option for ivshmem

All three mechanisms require modifying the code base of QEMU. As mentioned, the modifications total 800 new lines of code to the user-level QEMU application.

4.4.1 `ivshmem`: The Nahanni PCI device

Nahanni is implemented as a new device in QEMU called ivshmem. As mentioned, given the similarity to graphics memory behaviour, an existing virtual graphics card in QEMU

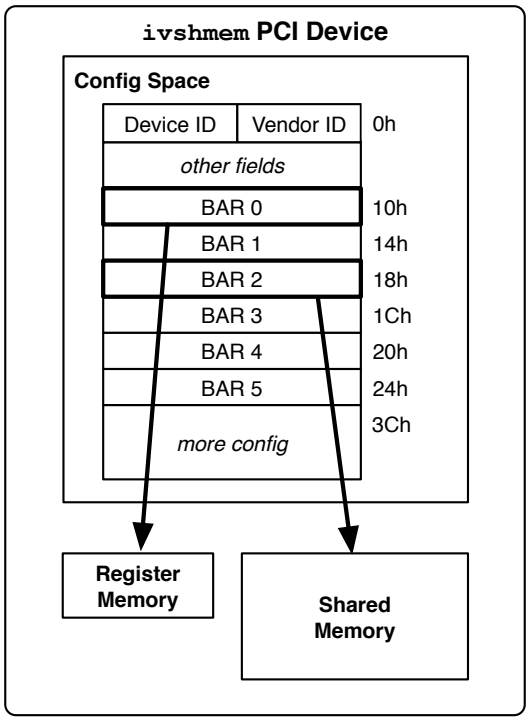


Figure 4.3: Nahanni: The `ivshmem` PCI device layout

The configuration space and 2 memory regions are shown. The base address registers (BARS) within the configuration space are used to access the memory regions. BAR0 points to register memory region while BAR2 points to the shared memory region.

---

was used as a guideline for creating the Nahanni device. A virtual PCI device was chosen as the mechanism for accessing the shared memory region on the host. Figure 4.3 illustrates the important components of the ivshmem PCI device that will be explained in this section, namely the configuration section, register memory and shared memory of the device. PCI supports accessing large regions of memory on devices such as graphics cards that can have gigabytes of video RAM. As well, PCI is supported by nearly all commodity operating systems and so could allow Nahanni to be used by other guest OSes such as Windows or FreeBSD.

The design of QEMU is intentionally modular. All new devices in QEMU must follow the *qdev* interface that defines a standard interface for peripheral devices that QEMU emulates. Figure 4.4 shows the qdev definition of the ivshmem device. The definition consists of a `PCIDeviceInfo` C structure as shown in Figure 4.4. This structure serves as a definition of the device that QEMU will use to create and control the device within the virtual

```
1  static PCIDeviceInfo ivshmem_info = {
2    .qdev.name = "ivshmem",
3    .qdev.size = sizeof(IVShmemState),
4    .qdev.reset = ivshmem_reset,
5    .init = pci_ivshmem_init,
6    .exit = pci_ivshmem_uninit,
7    .qdev.props = (Property[]) {
8       DEFINE_PROP_CHR("chardev", IVShmemState, server_chr),
9       DEFINE_PROP_STRING("size", IVShmemState, sizearg),
10      DEFINE_PROP_UINT32("vectors", IVShmemState, vectors, 1),
11      DEFINE_PROP_BIT("ioeventfd", IVShmemState, features, IVSHMEM_
            IOEVENTFD, false),
12      DEFINE_PROP_BIT("msi", IVShmemState, features, IVSHMEM_MSI, true)
            ,
13      DEFINE_PROP_STRING("shm", IVShmemState, shmobj),
14      DEFINE_PROP_STRING("role", IVShmemState, role),
15      DEFINE_PROP_END_OF_LIST(),
16   }
17 };
```

Figure 4.4: Nahanni: The qdev PCI device structure for ivshmem in QEMU

---

hardware including adding the necessary command-line parameters.

The structure defines the name (line 2), the size of state required (line 3) and function pointers to control the device at a high-level for reset (line 4), initialization (line 5) and deallocation (line 6). The `.qdev.props` field (line 7) defines the command-line options for ivshmem which will be discussed in Section 4.4.3 and Section 4.6.

The initialization of the ivshmem device is performed by the function `pci_ivshmem_init()`. When a QEMU VM (with or without KVM) is booted, all devices will have their initialization functions invoked. All persistent state for the device is maintained in a single data structure, the device structure. The device structure contains all the state the device needs to maintain to perform its functions. For example, a network device would need to create buffers to handle the sending and receiving of data. The size of the device structure must be declared in the `.qdev.size` field (line 3) as the qdev system will allocate the needed space for the device (the initialization function need not allocate it explicitly).

Though the creation of the ivshmem device is QEMU-specific in terms of the mechanisms described above, the device that is created must follow the PCI standard in its layout and behaviour. It is now fitting to discuss how the ivshmem device shares the POSIX memory region it mapped into QEMU (see the previous Section 4.3) with a guest OS through the PCI device interface.

Nahanni PCI config space

All PCI devices, virtual or real, have a 256-byte configuration space that is physically part of the device. The configuration space is labelled **Config Space** in Figure 4.3. When a device is connected to the PCI bus, the configuration space layout allows the OS to determine the type of the device, the vendor that made it, and all details about its operation. By reading the vendor and device IDs from the configuration space, the operating system can load the correct driver for the device if the OS has it available. If no driver is found for that particular device, the device cannot be used.

The PCI configuration space is divided into 27 fields that contain various details about the device. The first two fields of the configuration space are 16-bits each and specify the device and vendor ID, respectively. The remaining fields of the configuration space specify features of the device including interrupt behaviour and memory regions the device may have.

QEMU virtual devices have a configuration space that is implemented as a 256-byte array. The initialization function for a QEMU device must configure the PCI configuration space correctly so the guest OS can read it. The QEMU code base provides a convenient set of macros for setting the various fields of the config space. For example, the following macro sets the 16-bit vendor ID for the ivshmem device. RedHat, Inc. [51] which owns KVM, allowed its vendor ID to be used for the ivshmem device.

```
pci_config_set_vendor_id(pci_conf, PCI_VENDOR_ID_REDHAT_QUMRANET);
```

An important group of fields in the PCI configuration space in regards to the ivshmem device are the base address registers (BARs). BARs point to the regions of memory on PCI devices that are involved in the device's function (e.g. graphics memory on a graphics card). A device can support up to six BARs.

On the ivshmem device, up to three bars are used: BAR0 is always used and points to register memory for the device; BAR1 is optionally used if message-signalled interrupts (MSI) are used; BAR2 is always used and points to the shared-memory region. The Nahanni device driver reads the three BARs to configure device access in the guest for the regions. In general, a PCI driver for a particular device must know which BARs are used by that device and whether they are registers to control the device's behaviour or simply a memory region (as in a graphics card or ivshmem).

Nahanni Device Memory

The Nahanni PCI device contains two memory regions that are referenced by BAR0 and BAR2 in the ivshmem PCI configuration space. The ivshmem device contains a small region of register memory labelled as **Register Memory** (BAR0 in Figure 4.3). The register memory is not shared between guests. The use of the register memory will be discussed in Section 4.6.5. The POSIX shared-memory region is labelled as **Shared Memory** (BAR2 in Figure 4.3). Of course, the shared-memory region is shared between guests.

The above description summarizes the layout of the ivshmem device in accordance with QEMU's qdev device model. In the next section, we describe the modifications that were necessary to the QEMU memory-management code to support the use of a mapped POSIX shared-memory object for the shared memory region of the ivshmem device.

4.4.2 Mapped Memory Allocation

Virtual devices are allocated as part of the virtual hardware of a QEMU process within the QEMU process' address space. Since QEMU VMs are normal Linux processes, the QEMU process has to explicitly allocate memory for the virtual RAM of the VM and for any device memory (such as graphics memory). Since QEMU is just a regular Linux process, it allocates memory using the standard memory allocation methods such as `malloc()`. QEMU keeps track of memory using structures called RAMblocks (despite the name, the memory can be used for device memory, not just RAM). For example, the allocation wrapper function within QEMU, `qemu_ram_alloc()`, is passed an argument indicating the size of memory to be allocated and returns a pointer to the memory. The allocated memory is tracked in the RAMblocks structure and can be used by the QEMU hypervisor for system RAM or device memory.

As shown in Section 4.3, the standard Linux mechanisms for sharing memory, namely POSIX shared-memory objects and the `mmap()` system call, can allocate a region of memory and make it accessible within a process' address space. In order for QEMU to use this memory for device memory (i.e, ivshmem) a method was needed to add the mapped region of memory to the RAMblocks. Therefore, a new function named `qemu_ram_alloc_from_ptr()` was added to the QEMU memory allocation functions. The new function allowed a QEMU device's initialization to pass a pointer to an already allocated region of memory and have that memory added to the RAMblocks for proper management. In the case of the ivshmem device, a region of shared memory would be mapped into the QEMU process using mmap and then added to the memory of the running system. Figure 4.5 shows the lines

```
1 ptr = mmap(0, s->ivshmem_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
2
3 s->ivshmem_offset = qemu_ram_alloc_from_ptr(&s->dev.qdev,
4                            "ivshmem.bar2", s->ivshmem_size, ptr);
```

Figure 4.5: Adding memory accessed via `mmap()` to QEMU's memory allocation

The dynamic memory allocated by the `mmap()` call (line 1) is added to QEMU's memory allocation (line 3) via the new `qemu_ram_alloc_from_ptr()` function. This new function is used to add mmapped POSIX shared memory that serves as the shared-memory region between VMs.

of code that map the memory (line 1) and then add that memory to QEMU's RAMblock memory-management system.

The memory region is given a helpful name ("ivshmem.bar2") by passing a string to the function (lines 3 and 4). Naming the region allows for easier management and debugging. The allocated region is reserved for use by the ivshmem device within QEMU. When the call to `qemu_ram_alloc_from_ptr()` returns successfully, the memory can be used by the ivshmem device to provide the shared-memory region for the device.

4.4.3 New Command-line Option

As shown in Section 3.3, QEMU/KVM VMs are launched via the command-line. Nahanni's ivshmem device, like any device in QEMU/KVM, is added to a virtual machine via a command-line argument, such as:

```
-device ivshmem,shm=<name>,size=<size in MB>
```

The above command-line switch would simply be added to the larger QEMU/KVM command-line as shown in Figure 4.6.

```
qemu-system-x86_64 -smp 4 -device ivshmem,shm=nahanni_shm,size=4096 -
    hda karmic.img -net nic,macaddr=00:0c:29:f0:bc:30,vlan=0,model=
    virtio,netdev=bar -netdev tap,ifname=vhosttap0,downscript=no,id=
    bar,vhost=on -m 8g
```

Figure 4.6: An invocation of the QEMU hypervisor with an ivshmem device attached

When the VM specified in Figure 4.6 boots, an ivshmem PCI device will be created and connected to the virtual PCI bus in the guest. In the example command-line above, two parameters to ivshmem are specified: `size` and `shm`. These parameters define the shared-memory region to be used for the ivshmem device:

1. **size** defines the size of the shared-memory object. The size must be a power of two, a restriction of PCI memory regions.

2. **shm** specifies the POSIX shared-memory object to use as the shared memory. The object will be created if it does not exist and truncated to the specified size. If the object already exists, QEMU will not resize it but will ensure the size of the object matches the size given with the 'size' parameter.

There are other parameters that pertain to a more advanced Nahanni configuration that will be described in Section 4.6.

4.5 Component 3: Guest OS Device Driver

The third and final component that is necessary for Nahanni is a device driver. Since the ivshmem device conforms to the PCI standard for device, the device driver will make use of Linux's in-kernel PCI driver interface to configure the device for use. The driver that will be described is for the Linux OS since we target Linux as our guest OS for our evaluation. We chose to use the Linux UIO device driver framework for the ivshmem device to simplify the implementation and minimize the amount of code that runs at kernel level.

It should be mentioned that the Nahanni ivshmem device inside QEMU is both guest driver and guest OS agnostic. A non-UIO Linux driver could be written to control the ivshmem device. In fact, the first guest driver implemented was not based on UIO. Also, the QEMU-based ivshmem device can be supported by guest OSes other than Linux by writing a driver for the desired OS (e.g., Windows). We are aware of at least one Nahanni user who has written a driver for ivshmem for an operating system other than Linux.

Section 4.8 will describe how the ivshmem device is accessed from user-level by applications. Here, we will discuss the driver and how it configures the device.

Device drivers are the software interface that allow the kernel and applications to make use of hardware devices. The simplicity of Nahanni's design allows the driver to be straightforward as well. In short, the driver configures the ivshmem device and requests the guest kernel to map the device memory into the kernel's address space. The driver then enables

```
1    info->mem[1].addr = pci_resource_start(dev, 2);
2    if (!info->mem[1].addr)
3       goto out_unmap;
4    info->mem[1].internal_addr = pci_ioremap_bar(dev, 2);
5    if (!info->mem[1].internal_addr)
6       goto out_unmap;
7
8    info->mem[1].size = pci_resource_len(dev, 2);
9    info->mem[1].memtype = UIO_MEM_PHYS;
```

Figure 4.7: Nahanni: kernel driver initialize for ivshmem device memory

The ivshmem shared-memory region is configured in the UIO driver code. Three PCI configuration functions are called on lines 1, 4 and 8 to configure the ivshmem shared-memory region so that the memory can be accessed from user-level.

the necessary functionality to allow applications to map the memory region from kernel space into user-level.

The code section shown in Figure 4.7 is part of the ivshmem UIO driver. While Figure 4.7 does not show the complete driver, it shows the essential part of the kernel driver that configures the Nahanni shared-memory region. The ivshmem driver calls three PCI kernel functions: `pci_resource_start()`, `pci_ioremap_bar()` and `pci_resource_len()`. In particular, `pci_ioremap_bar()` requests that the kernel map the device memory on BAR2 (the shared-memory region) into the guest kernel's virtual address space.

Once the guest kernel UIO driver for the ivshmem device has executed, the device is configured and ready for use from the user level, including user-level libraries and applications. So, while the configuration may seem overly simple, all that is required is to enumerate the memory regions in this way so that applications can access them.

In general, a Linux device driver typically makes its respective device accessible via a file added to the `/dev` file system and ivshmem is no different. When the device driver is run, a device file will be created under the `/dev` directory that corresponds to the device. The device files for UIO devices all begin with "uio" followed by an integer. The first UIO device to be initialized will be associated to the device file `/dev/uio0`. The second will be associated with the device file `/dev/uio1` and so on.

Applications inside the guest can access the ivshmem device by performing system calls on the device file (under the `/dev` directory) associated with the device. UIO differs from most device drivers in that it is designed to minimize the amount of code that runs in the

kernel, although some code must run in the kernel for a UIO device driver such as the code shown in Figure 4.7.

While a detailed explanation of how applications use shared memory is left for Section 4.8, we give a brief example here.

As mentioned in Section 3.6.2, the UIO model allows mapping of the device-memory regions into user-level by applications to access the memory without requiring further system calls. When accessing a UIO device, the call to `mmap()` looks like an `mmap()` call to map a memory-mapped disk or POSIX shared-memory object (see Section 4.3):

```
map_region = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 1 *
    getpagesize()));
```

A UIO device-memory region is mapped into user-level using the regular `mmap()` system call. One small caveat to using the `mmap()` system call is that UIO overloads the sixth parameter of the `mmap()` call, which is typically reserved for an offset. The offset parameter normally indicates an offset into a file that is to be mapped. For example, mapping 1,024 bytes from a file at offset 512 would map the byte range from 512 up to 1,536. With UIO the semantics are different. With UIO devices, the offset parameter specifies the memory region to map, by passing a multiple of the OS page size. For example, in the example call shown above, UIO memory region 1 would we mapped.

Recall that Figure 4.7 shows the code that initialized memory region 1 (`info->mem[1]`). By passing `1 * getpagesize()` as the offset argument, the kernel is instructed to map the memory region indexed at 1 into the address space of the calling application. Note that the memory region indexed at 0 will be discussed later in Section 4.6.5.

4.5.1 Brief Summary

With the above three components: a shared-memory mechanism, the new ivshmem virtual PCI device and a guest OS device driver, it is now possible to share memory between host and guest applications as well as between guest applications running in different VMs. Figure 4.8 shows how POSIX shared memory is exposed in two VMs running on the same host machine. Shared memory is enabled by having the two VMs map the *same* POSIX memory object on the host. The shaded box is meant to indicate that the memory is shared up to the user-level in both guests and no copying of data occurs when the shared memory is accessed.

In the next section, we will augment the basic configuration just described to increase the functionality of Nahanni by adding a shared-memory server that will support a novel in-

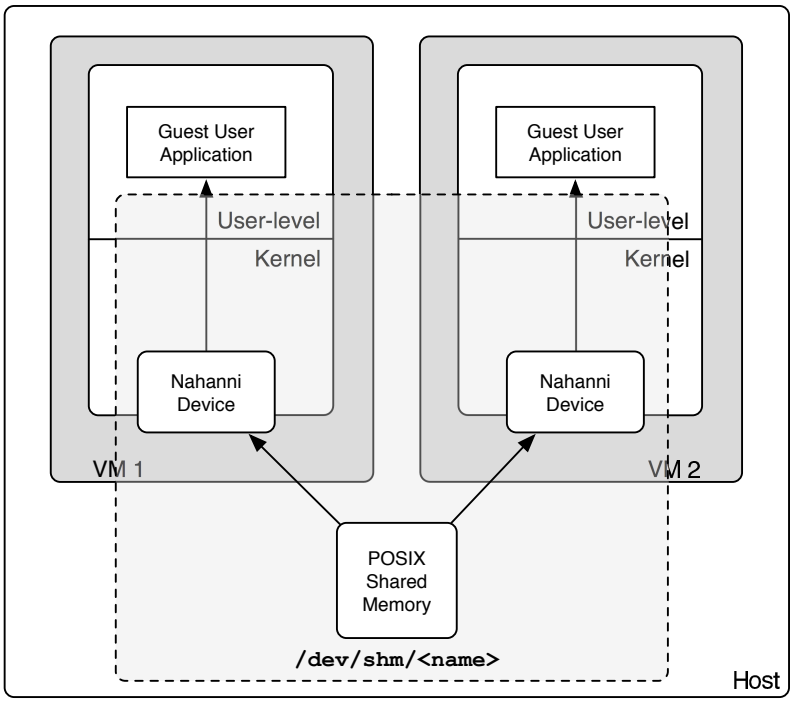


Figure 4.8: Two VMs sharing the same POSIX memory region

The sharing of the same POSIX memory region allows two VMs to communicate efficiently. While only two VMs are illustrated, Nahanni does not place a limit on the number of VMs that can share a single shared memory object.

---

terrupt mechanism to increase the range of applications that can take advantage of Nahanni.

4.6 Inter-VM Notifications

The primary goal of Nahanni is to provide shared memory between the host, guest VMs, and their associated applications. While certain applications may find sharing memory and using load-store operations between VMs useful on its own, inevitably other applications will require notification mechanisms to work in conjunction with shared memory. Notification is important if, for example, the shared memory was used between producer and consumer processes running in separate, co-located VMs. A producer process could notify the consumer, running in a different VM, that new data is available to be consumed by sending a notification.

In addition to basic notifications, synchronization mechanisms such as barriers, semaphores and mutexes may be necessary depending on the particular use-case of the

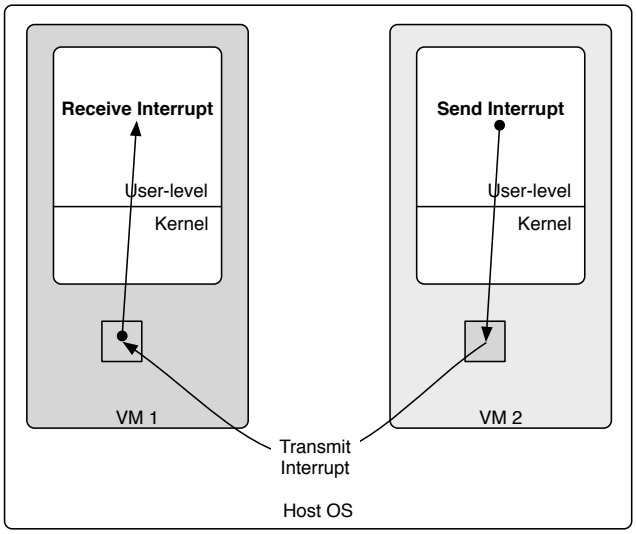


Figure 4.9: An interrupt being sent from one guest to another

The sending guest triggers an interrupt that is delivered to the user-level of a receiving guest

shared memory. With the functionality described so far, Nahanni supports spinlocks that monitor and modify a value stored in shared memory. Therefore, synchronization primitives built upon spinlocks are inherently possible via Nahanni shared memory. However, spinlocks have drawbacks in certain scenarios that developers may want to avoid. In particular, when under contention, spinlocks can lead to wasted CPU cycles as the waiting process (or processes) continually poll the variable in shared memory. Polling for long intervals is generally undesirable and certain applications will want an alternative to spinlocks.

The general alternative to polling is blocking. Blocking mechanisms allow processes to cease to execute, or block, while waiting for some condition to occur to avoid the wasted cycles that polling would incur. Mechanisms that support blocking semantics require notification that the event they are waiting upon has occurred so that any blocking process can be unblocked. Such notifications are generally referred to as *interrupts*.

Figure 4.9 illustrates the basic functionality of an inter-VM interrupt. An interrupt is triggered by a particular application, then the interrupt is transmitted and received by an application running in a different guest. Given the utility of such a mechanism, the next step in Nahanni's design and implementation was to add an interrupt mechanism that would allow applications that are using Nahanni for sharing memory to signal each other as illustrated in Figure 4.9. Adding such a signalling mechanism would allow for richer

notification and synchronization mechanisms for Nahanni applications.

Providing interrupts will require four individual mechanisms to able to:

1. uniquely identify different guest VMs,
2. send a message from inside a guest VM,
3. transport a message from one VM to another, and
4. notify a guest that a message has arrived.

For each of the above mechanisms, we will seek to build upon the design decisions that have already been made, namely the PCI standard and the UIO driver interface. Moreover, in keeping with the QEMU/KVM design philosophy, we will use existing functionality present in Linux as much as possible rather than re-inventing the wheel.

Considering the above four required mechanisms, we first decided that a centralized server was the correct approach to (1) facilitate VM identification and (2) implement message transport. While a distributed mechanism is possible (e.g., as future work), it would be much more complicated than a centralized one. Since Nahanni is designed for a single host, scalability issues are limited to the number of VMs that a single host could run, which we posit to be on the order of dozens and unlikely to grow beyond a few thousand in the near future. This centralized server that will coordinate interrupts between QEMU/KVM VMs is simply called the Shared-Memory Server (SMS).

4.6.1 The Shared-Memory Server

The Shared-Memory Server (SMS) is a stand-alone host application external to QEMU that will manage the sharing of resources for inter-VM communication between QEMU guests. The SMS exists mainly to provide some convenience and to meet our current needs. Further refinements and optimizations to the SMS would be the subject of future work.

The SMS provides a single point of access for all resources related to Nahanni. By design, the SMS will handle distributing communication endpoints to all guests as well as providing access to the POSIX shared-memory object itself. Having all resources controlled by the SMS reduces the chance of misconfigurations. For example, if guests only contacted the SMS for interrupt endpoints, then guests may be able to send interrupts but may (incorrectly) connect to different shared-memory regions via the `-shm` parameter. Having the SMS handle all aspects of Nahanni reduces the likelihood of configuration errors. Therefore, the SMS centralizes access to shared memory and the interrupt mechanisms.

Since the SMS is a stand-alone process, the QEMU VMs must connect to it using some form of IPC. We decided on a Unix Domain Socket (UDS) which is a common IPC mechanism supported in UNIX-like OSes including Linux. UDSs are created with a path name in the file system namespace which will allow for easier configuration of the guests. QEMU also supports connecting to file descriptor mechanisms via QEMU character devices (called *chardevs* in QEMU), so the support within QEMU is already present to communicate with a process like the SMS over a UDS.

Figure 4.10 shows the launching of a particular SMS. The command-line to start the SMS must specify the size (-m) and name (-n) of the POSIX shared-memory object as well as the UDS the SMS listens on (-p). The particular server whose invocation is shown in Figure 4.10 will share a 256 MB POSIX memory region named *dynamo* (the name is arbitrary). The SMS will listen on the socket via the path `/tmp/ivshmem_socket` for QEMU processes to connect to it.

```
ivshmem_server -m 256 -p /tmp/ivshmem_socket -n dynamo
```

Figure 4.10: Launching Nahanni Shared-Memory Server (SMS)

To allow a QEMU process connect to an SMS, the QEMU command-line options for Nahanni add an additional `-chardev` parameter that specifies the UDS to connect to. In particular, to connect to the SMS launched as in Figure 4.10, a QEMU process would add the new chardev parameter coupled with the `-device ivshmem` parameter (seen previously) as follows:

```
-chardev socket,path=/tmp/ivshmem_socket,id=foo -device ivshmem,
    chardev=foo,size=256
```

The chardev parameter is given an identifier name, via `id=foo`, that the ivshmem device parameter associates with (via `chardev=foo`). The existing `-device ivshmem` parameter is changed to specify a `chardev` to communicate with rather than the name of the shared-memory object (specified by `shm=<name>` as shown in Section 4.4.3). Figure 4.11 shows the complete QEMU command-line that will connect a VM to the SMS launched as shown in Figure 4.10.

Previous to the SMS, guest VMs shared memory by specifying the same shared-memory object and size with the `-shm <name>` parameter. When using an SMS, guest VMs must communicate to the same SMS in order to share memory. The SMS will provide access to

```
qemu-system-x86_64 -smp 4 -chardev socket,path=/tmp/ivshmem_socket,id=
    foo -device ivshmem,chardev=foo,size=256 -hda karmic.img -net nic,
    macaddr=00:0c:29:f0:bc:30,vlan=0,model=virtio,netdev=bar -netdev
    tap,ifname=vhosttap0,downscript=no,id=bar,vhost=on -m 8g
```

Figure 4.11: Launching QEMU to communicate with an SMS

the POSIX shared-memory object. The `-device parameter` still requires the size of the shared-memory object be given with the `size` parameter as a sanity check for the size of the shared memory. When the QEMU processes receives the shared memory file descriptor from the SMS it will confirm the size is correct.

Before discussing the SMS in further detail, the next two sections will introduce the first two items in the list in Section 4.6: (1) the VM identification method and (2) the interrupt transport mechanism, as they are fundamental to understand the behaviour of the SMS.

4.6.2 Identifying Guest VMs

To be able to send messages between VMs, applications will need a way to identify the recipient VM to which they intend to send a message. We require a mechanism to identify VMs with an identifier. The identifier will need to scale to a reasonable number of VMs. A VM must also be able to determine its own identifier as well the identifiers of other VMs in order to send messages to those other VMs.

As mentioned, choosing to support interrupts between multiple guest VMs motivated the centralized control via the SMS. Centralized control means the SMS can provide distinct identifiers to each VM when the VMs initially connect to the server. The identifier for each VM, or *VM ID*, is a 16-bit unsigned integer in the range of 0 to 2^{16}. Therefore, in addition to receiving the access to the shared memory, the SMS will provide each VM a unique VM ID.

It is also worth mentioning that while VM IDs were initially added to Nahanni to support an interrupt mechanism, they are useful for any scenario in which VMs may want to identify each other. For example, since VM IDs are simply an integer they could be used to statically partition the shared memory by using a VM's VM ID as an offset into fixed-sized buffers in the shared-memory region.

Now that VMs can be identified via their VM ID, we move on to providing an interrupt transport mechanism.

4.6.3 Interrupt Transport

Sending interrupts between VMs will require some form of Linux IPC since QEMU processes are Linux processes. To support interrupts, Nahanni uses a relatively new Linux IPC mechanism called *eventfds*. Eventfds are intended to be simpler and more efficient than other IPC mechanisms such as pipes or sockets. Eventfds are created via the `eventfd()` system call which is similar to the `pipe()` system call for creating a pipe. The `eventfd()` system call returns a file descriptor that is the reference for the eventfd. Eventfds can be shared between processes like any file descriptor, through forking or through passing them to other processes.

Eventfds can be read to and written from using the standard POSIX file operations. One important distinction is that, by design, eventfds do not provide buffer semantics. This design is intentional to avoid the overheads that buffering incurs [9]. Eventfds behave like a register, storing a single value that can be overwritten. We will use the writing to an eventfd as the mechanism to send an interrupt to another guest.

As mentioned, eventfds are referenced within processes by file descriptors. Similar to the VM ID, it will be the task of the SMS to distribute file descriptors for the eventfds when new guests join. The SMS is able to create and distribute the eventfds because Linux processes can pass file descriptor across a UDS. In the following discussion, we will use the phrase “interrupt endpoints” to refer to the eventfd file descriptors, which are different from the file descriptor for the POSIX shared-memory region.

It was decided that each VM would have a single eventfd it would listen on for interrupts to keep the numbers of eventfds linear with respect to the number of guest VMs. Each QEMU process will be passed an interrupt endpoint for every other VM in order that they can send interrupts to all other guests connected to the same SMS.

It is worth mentioning that the choice of eventfds could be easily changed to another Linux IPC mechanism because of the abstraction that file descriptors provide. If a different communication mechanism, say pipes, were deemed preferable to eventfds, the changes necessary would be limited to the SMS since the QEMU processes simply receive file descriptors and are otherwise agnostic to the underlying IPC mechanism.

Coordinating the sending and receiving of VM IDs and file descriptors (for the shared memory and eventfds) requires a protocol that the QEMU VMs and SMS will follow in order to communicate with each other. The next section we return to discussing the SMS in detail and introduce the protocol used by Nahanni VMs to communicate with the SMS.

4.6.4 Shared-Memory Server Protocol

In any multiprocess configuration, a protocol must be in place that the clients and server will use. For Nahanni, a protocol for VMs to join and leave the SMS is necessary so that VMs can acquire the necessary information from the server to use interrupts. The protocol is simple. The VMs connect and disconnect to the SMS listening socket using a QEMU chardev that is added to the command-line (see Figure 4.11).

The SMS is only involved when guests join the SMS and when they leave. The actual sending of interrupts does not involve the SMS since the guests are fully connected by their interrupt endpoints.

In the protocol that follows there is one invariant worth mentioning: By design, guest VMs never send data to the SMS, they only receive information from the SMS. The SMS only needs to know when guests connect or disconnect from itself and detecting connections and disconnections does not require the sending of data.

New Guest Connections

Figure 4.12 illustrates the communication exchange with the server that will occur when a new guest connects to the server. As was mentioned in the previous section, notice in Figure 4.12 that the VMs only receive data from the server, they never send data to the server.

When a guest joins the server, it will receive the following:

1. its own VM ID,
2. a file descriptor for the shared-memory region,
3. a file descriptor for its interrupt endpoint, and
4. a file descriptor/VM ID pair for each already existing guest

The first three values are sent in the first three messages after the guest connects in Figure 4.12. Once the server has sent the new guest its own data, it must send the data for all existing guests to the new VM. These messages (one for each existing guest) transmit the VM IDs and interrupt endpoints for each existing guest already connected to the SMS to the new guest. These endpoints are how the new guest will send interrupts to existing the guests.

Once the new guest is properly configured by the above steps, the server sends *updates* to the existing guests that a new guest has joined. This update, sent to each existing guest,

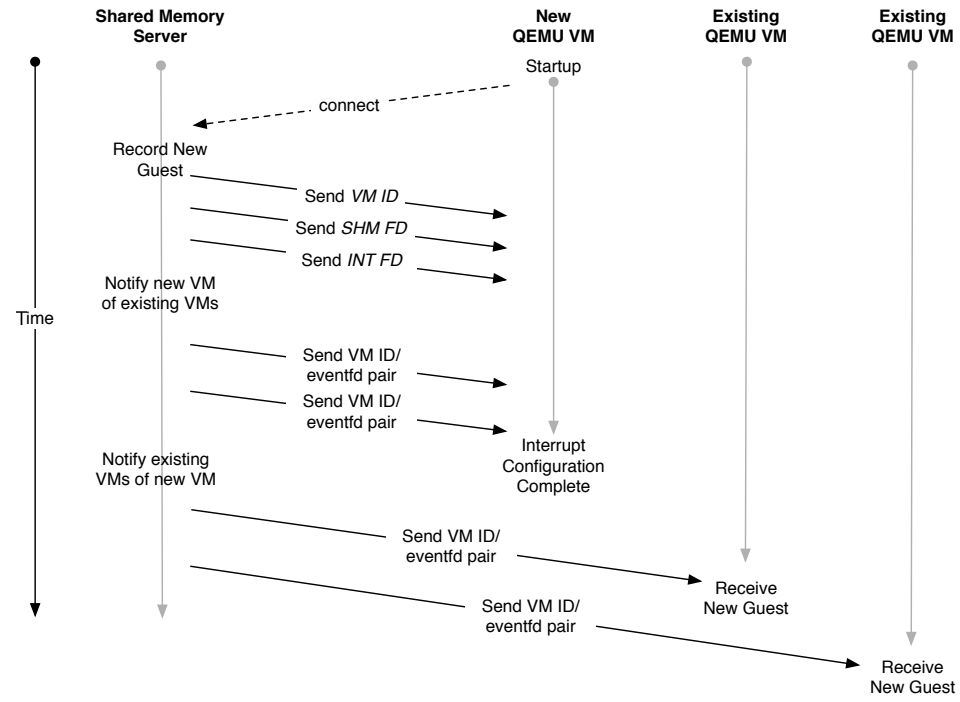


Figure 4.12: Server-Guest communication when a new VM is launched

will consist of the new VM's VM ID and its interrupt endpoint. With this new endpoint, existing guests can send interrupts to the new guest.

Once all existing guests have received the update about the new guest, the VMs are now all directly connected, each VM can send an interrupt to any other VM via an eventfd. It is worth reiterating that the SMS is not involved in the transmission of interrupts, it is only involved in configuration when VMs join or leave.

Guest disconnections

The other event that the SMS is involved in is a guest disconnection. Figure 4.13 illustrates the protocol behaviour when a guest disconnects from the SMS. When a guest VM shuts down it will disconnect from the server by closing its connection with the SMS. The SMS detects the connection has been closed and notifies the other VMs that the guest with the given VM ID has disconnected. The SMS sends a disconnection update to each remaining VM. When a VM receives the update, it closes the interrupt endpoint for that guest and deletes its VM ID from its internal data structures.

To this point we have discussed in detail the SMS, our identification method (VM IDs) and interrupt transport mechanism (eventfds). We have also just illustrated the protocol

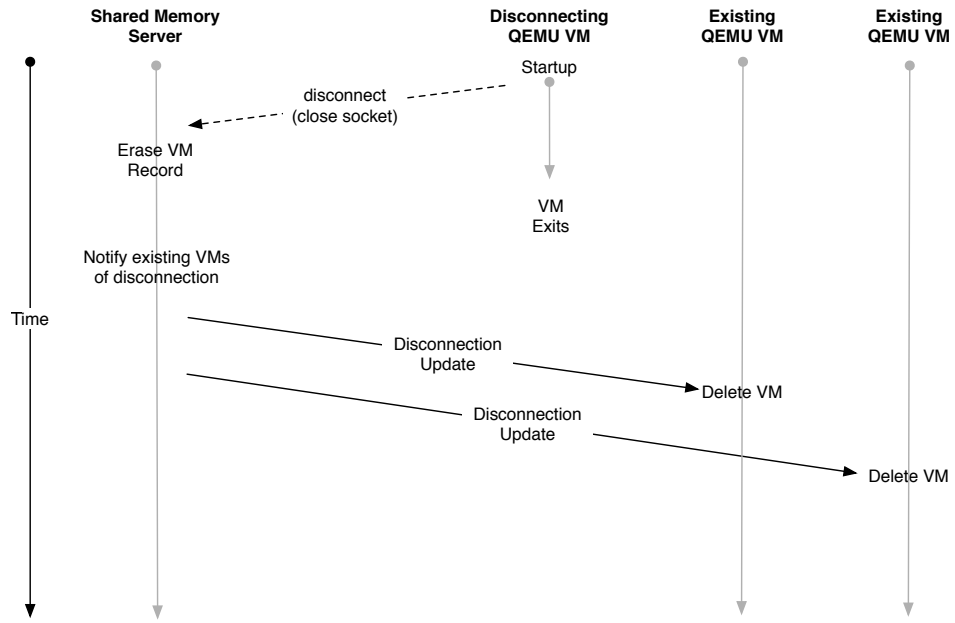


Figure 4.13: Server-Guest communication when a VM disconnects

when guests connect to and disconnect from the SMS. The two remaining items from the list in Section 4.6 are: (3) The mechanism to trigger the sending of an interrupt and (4) the mechanism to receive an interrupt. Since Nahanni shares memory to user-level, our interrupts must also be able to be sent from and receive into applications running at user-level. In the next section, we will describe Nahanni's device registers and how the UIO driver model allows user-level access before discussing how interrupts are sent and received.

4.6.5 Nahanni Device Registers

User-level applications inside the guest will interact with the ivshmem device through the device driver. In particular, a guest's VM ID must be made available to applications. Applications will also need to be able to send and receive interrupts. The functionality for these tasks will be provided by the ivshmem device and, in particular, by reading and writing device registers on the device.

In general, PCI devices exchange usage-specific information (such as a VM ID) with the kernel and applications and trigger actions via PCI device registers. Registers are small regions of memory (typically 16 or 32-bits) that can be written to or read from much like CPU registers. Generally speaking, reading a PCI register retrieves some information from a PCI device. Writing to a PCI register can make a PCI device perform a particular action

```
1    info->mem[0].addr = pci_resource_start(dev, 0);
2    if (!info->mem[0].addr)
3       goto out_unmap;
4    info->mem[0].internal_addr = pci_ioremap_bar(dev, 0);
5    if (!info->mem[0].internal_addr)
6       goto out_unmap;
7
8    info->mem[0].size = pci_resource_len(dev, 0);
9    info->mem[0].memtype = UIO_MEM_PHYS;
```

Figure 4.14: Nahanni: kernel driver initialize for ivshmem registers

such as sending a network packet in the case of a PCI network device.

For QEMU PCI devices, registers are implemented as regions of memory on the device much like our shared-memory region. Note that the register memory region labelled as **Register Memory** in Figure 4.3. To be consistent with UIO terminology, we sometimes use the shortened phrase "register region" in this discussion. The main differences between the register memory region and the shared memory are that the register region will be much smaller and will not be shared between guests. Another difference is that when a write to a register memory occurs inside the guest, control is transferred from the guest OS to the QEMU hypervisor and the software in QEMU can emulate the behaviour of the device.

We chose to make our registers 32 bits each to correspond to the standard size for an integer. The ivshmem device has a 256-byte register region which allows for up to 64 32-bit (4 byte) registers. Currently, only four registers are used with the remaining registers available for later development.

Following with the UIO design, the register region can be mapped into user-level much like the shared-memory region. The register region is assigned to BAR0 in the PCI configuration (see Figure 4.3). To allow the mapping of the registers to user-level by applications, the registers are indexed as memory region 0 in the UIO driver as shown in Figure 4.14. Notice on line 4 on Figure 4.14 that BAR0 is remapped via `pci_ioremap_bar(dev, 0)` to memory region 0 (`info->mem[0]`).

Therefore, to map the register region into an application, a value of 0 is passed as the sixth argument (offset) `mmap()`. For example, the following `mmap()` call:

```
int *registers = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
    0)
```

will map the register region into user-level. By mmaping the Nahanni device registers into

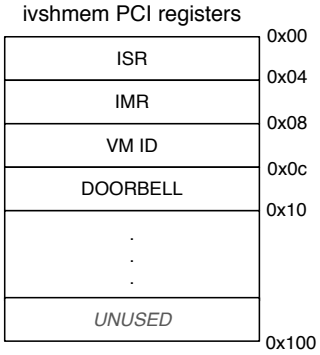


Figure 4.15: The register layout of the ivshmem device

The register region contains up to 64 32-bit registers, of which 4 are currently used. The values on the right side of the figure are the offsets for the individual registers. The offsets are multiples of 32 bits or 4 bytes.

---

user-level as shown, applications will be able to send interrupts without trapping to the guest kernel.

The registers are involved in sending and receiving of interrupts. The individual registers are illustrated in Figure 4.15. The first register, at an offset of 0 bytes, is the interrupt status register (ISR). The second register, at an offset of 4 bytes, is the Interrupt Mask Register (IMR). ISR and IMR registers are common to PCI devices that receive interrupts.

The other two registers are specific to Nahanni. The third register which is at an offset of 8 bytes is used to store the VM ID of the guest. Finally, the fourth register is the *Doorbell* register that will be used to trigger the sending of interrupts to other guests connected to the SMS. The remaining registers are currently unused and left for future development.

4.6.6 Interrupt Transport

Now that Nahanni's PCI register memory has been explained, as well as how an application can map the registers, we will discuss the semantics of interrupts.

The sending and receiving of interrupts involves the QEMU hypervisor, which runs as a user-level process on the host. When a write to and read from a PCI register occurs inside the guest, control then passes to the QEMU hypervisor, which executes a function to emulate the behaviour of the device in response to that register being written to or read from. For Nahanni, a write to the Doorbell register results in code being executed in QEMU that will write a value to the eventfd.

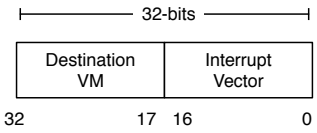


Figure 4.16: An Illustration of the `ivshmem` Doorbell register

The Doorbell register on the Nahanni ivshmem PCI device is divided into two 16-bit fields that specify the recipient VM ID and the interrupt to trigger, respectively.

Each ivshmem device has a single Doorbell register, but possibly multiple other VMs to send interrupts to, the value written to the Doorbell must indicate which VM to send the interrupt to. Nahanni also supports assigning multiple eventfds to each guest to allow for multiple interrupt types. To deal with this complexity, we split the 32-bit doorbell register into two 16-bit fields, one for the recipient VM and the other for the interrupt number. When only one eventfd is assigned to each VM, the interrupt number must be 1. The write to a register is a single operation, so the two 16-bit values must be combined into a single 32-bit value.

4.6.7 Sending an Interrupt from User-level

Figure 4.16 illustrates the two fields of the Doorbell register. The code shown in Figure 4.17 is executed within the hypervisor in response to a write to the Doorbell register inside the guest. The two variables on lines 6 and 7 separate the two 16-bit fields from the 32-bit value written to the Doorbell register into the variable `dest`, the destination VM ID, and `vector`, the vector to signal in the other guest. After various checks are made, the eventfd is written to with a value of 1 on line 18. The value written to the eventfd by the QEMU hypervisor is always 1. It is possible to write any 64-bit value to an eventfd. But, as mentioned, eventfd values are not buffered so trying to communicate a value is error prone as the value may be overwritten before it can be retrieved by the receiver.

Once the value of 1 has been written to the eventfd, the sending half of the communication is complete. Next, we will discuss the receiving half.

When a value has been written to a guest's eventfd, the receiving VM's QEMU hypervisor detects that a value is available on the eventfd's file descriptor using the well-known `select()` system call. Figure 4.18 shows the callback function that is registered with the eventfd is then executed to receive the interrupt. The value of the eventfd is passed to the callback function. Recall that the value is always 1 since that is the only value ever written

```
1 static void ivshmem_io_writel(void *opaque, target_phys_addr_t addr,
      uint32_t val)
2 {
3   IVShmemState *s = opaque;
4
5   uint64_t write_one = 1;
6   uint16_t dest = val >> 16;
7   uint16_t vector = val & 0xff;
8
9   /* check that dest VM ID is reasonable */
10  if (dest > s->max_peer) {
11     IVSHMEM_DPRINTF("Invalid destination VM ID (%d)\n", dest);
12     break;
13  }
14
15  /* check doorbell range */
16  if (vector < s->peers[dest].nb_eventfds) {
17     IVSHMEM_DPRINTF("Writing %" PRId64 " to VM %d on vector %d\n",
           write_one, dest, vector);
18     if (write(s->peers[dest].eventfds[vector], &(write_one), 8) != 8)
           {
19        IVSHMEM_DPRINTF("error writing to eventfd\n");
20     }
21  }
22 }
```

Figure 4.17: Nahanni: QEMU code to send an interrupt via an eventfd

The `write()` system call on line 18 writes to the eventfd which will result in an interrupt being triggered in the guest listening on that eventfd.

```
1 static void ivshmem_receive(void *opaque, const uint8_t *buf, int size)
2 {
3    IVShmemState *s = opaque;
4
5    ivshmem_IntrStatus_write(s, *buf);
6
7    IVSHMEM_DPRINTF("ivshmem_receive 0x%02x\n", *buf);
8 }
9
10 static void ivshmem_IntrStatus_write(IVShmemState *s, uint32_t val)
11 {
12    IVSHMEM_DPRINTF("IntrStatus write(w) val = 0x%04x\n", val);
13
14    /* set the ISR */
15    s->intrstatus = val;
16
17    /* signal interrupt into guest*/
18    ivshmem_update_irq(s, val);
19
20    return;
21 }
```

Figure 4.18: Nahanni: QEMU code to receive an interrupt via eventfd

to eventfds in Nahanni. The callback function stores value in the ISR (line 15) and triggers an interrupt inside the guest OS (line 18).

It is worthwhile to mention that Nahanni also supports message signalled interrupts (MSI), which are part of the PCI specification and are an alternative to regular interrupts for PCI devices. MSI allow for multiple interrupt vectors per device, so a device can receive different kinds of interrupts. The application semantics of receiving both regular interrupts and MSI are the same, and eventfds are the delivery mechanism for both types of interrupts. The MSI support could provide a richer interrupt in the future and is included for that reason.

4.6.8 Receiving an Interrupt to User-level

When the interrupt is raised inside the guest, the device driver must handle the interrupt. Here again, the design of UIO allows applications running at the user-level to receive and respond to interrupts, something usually reserved for kernel-level drivers. The method of handling interrupts at the guest user-level level is simple. A simple block of code that will receive an interrupt from a UIO device is shown in Figure 4.19.

Guest applications receive interrupts by performing a `read()` system call on the `/dev`

```
1   int rv, buf, fd;
2
3   fd = open("/dev/uio0", O_RDWR);
4
5   buf = 0;
6   rv = read(fd, &buf, sizeof(buf));
```

Figure 4.19: A simple example of receiving an interrupt via UIO in a guest application

The `read()` call on line 6 will block until an interrupt is delivered to the Nahanni device associated with `/dev/uio0`.

`/uioN` device file that was created for the Nahanni device. UIO uses the semantics of the `read()` call to allow the application to receive an interrupt.

Under normal usage, a `read()` system call will block until data is available to be read. In the case of UIO, the read call returns when an interrupt is received. The value stored in the buffer when the read returns is a 32-bit integer that indicates the number of interrupts the driver has received since it was loaded. The read must be 32 bits (4 bytes) in size as that is the size of data that a UIO device always returns on a read operation.

To explain these semantics a bit further, consider a driver that is loaded into the kernel and has not received any interrupts. After it receives the first interrupt and an application performs a read, the application will be returned a value of 1, since one interrupt has been received since loading. When a second interrupt is delivered and the application performs a read, the value of 2 will be returned.

Because interrupts are received asynchronously by the device driver, it is possible that an ivshmem device may receive more than one interrupt between reads. After reading the values of 1 and 2 respectively as above, if two interrupts are received before the next read, that read will return a value of 4, since a total of 4 interrupts have been received since the device driver module was loaded. These interrupt semantics are unique to UIO. Recall that the motivating principle behind UIO is to move the majority of the driver logic code (e.g., handling of interrupts) to the user-level. It is entirely up to the applications that use UIO as to the meaning of an interrupt and what should be done when one is received.

Section 4.8 will elaborate on writing applications to access Nahanni shared memory as well as to send and receive interrupts.

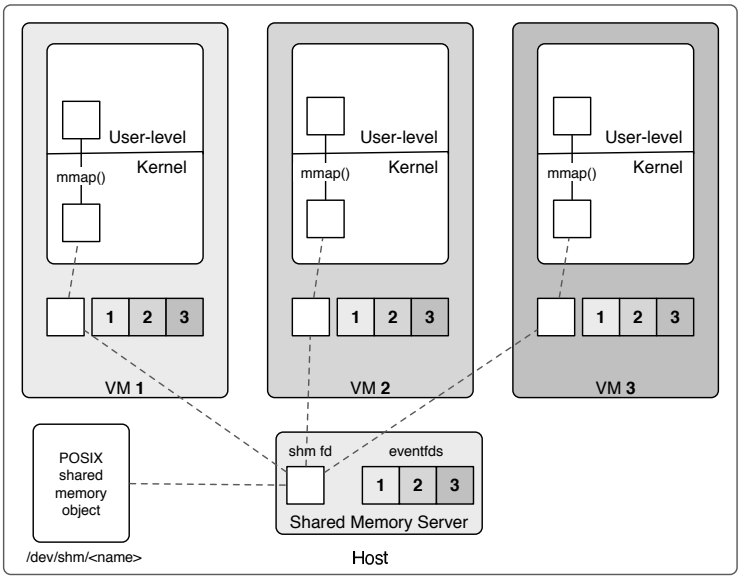


Figure 4.20: Three Virtual Machines using Nahanni

Each VM has 3 eventfds for sending interrupts (shown as numbered boxes) and one file descriptor for the shared-memory region (white box). All file descriptors (for both the shared-memory object and the eventfds) were received from the Shared-Memory Server.

4.7 The Big Picture

To this point we have discussed the details of how Nahanni enables shared memory between VMs and how interrupts between guests are supported when using the SMS. Figure 4.20 illustrates how these different pieces come together to allow multiple VM guests (in this case, three VMs) to share memory and send interrupts between one another.

The small, square white box in each VM represents the file descriptor for the POSIX shared memory itself. Notice how the white boxes appear in multiple places (e.g., user-level, kernel, shared-memory server) to reflect the different layers and components that can see the file descriptor. The shared-memory region is also received from SMS when the guest VM connects to SMS on startup. The dotted lines indicate the mappings that allow the POSIX shared object to be shared up to user-level without incurring any copying.

The other three shaded boxes in the SMS and in each VM in Figure 4.20 represent the interrupt endpoints (i.e., eventfds) that support interrupt delivery. In Section 4.6, we introduced the SMS and described how it distributes interrupt endpoints to each guest VM. Interrupts are sent by a guest by writing to the endpoint that the receiving guest is listening on. Each guest stores endpoints for all other guests and listens on one particular endpoint

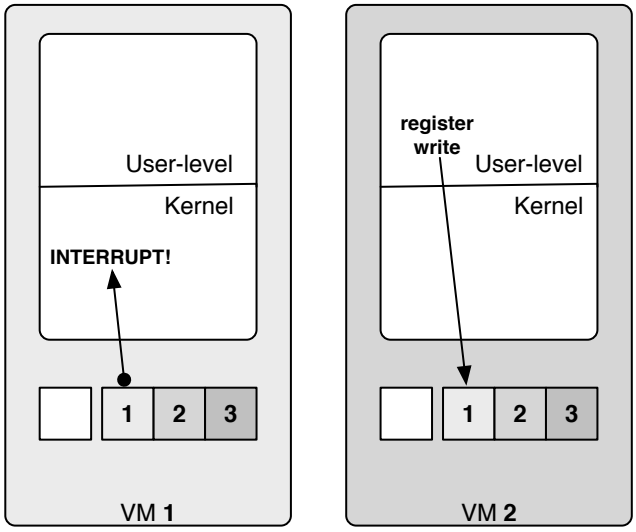


Figure 4.21: Triggering an interrupt using eventfds

This figure illustrates the interrupt transport using eventfds as VM 2 writes to the eventfd for VM 1 which triggers an interrupt into VM 1.

---

assigned to them.

As discussed in Section 4.6.5, Nahanni's ivshmem device implements the sending of interrupts by having a Doorbell register that, when written to by an application in the guest, triggers a `write()` to the correct endpoint for the receiving guest. Consider Figure 4.21 that illustrates the sending of an interrupt between two VMs. In the illustration *VM 2* triggers an interrupt in *VM 1* by writing to the endpoint associated with *VM 1*. The transmission of the interrupt involves the four stages described above, namely:

1. Writing to the Doorbell register,
2. QEMU writing to the corresponding eventfd,
3. The receiving guest triggering an interrupt, and
4. The receiving application reading from the `/dev/uio0` device.

Figure 4.21 represents a guest-to-guest interrupt, but the mechanism can be used for host-to-guest as well. Eventfds are a general Linux mechanism, they are not specific to KVM or virtualization in general. As such, eventfds can be used by any Linux application. An application running on the host can communicate with SMS and receive file descriptors (just like a guest VM does) by following the protocol described in Section 4.6.4 in order

to communicate with the VMs by writing to the POSIX shared-memory object and sending and receiving interrupts by using the eventfds. For example, as part of our benchmarking in Section 5.3, we will show that a user-level application can transfer a file into a VM from the host. This application first communicates with the SMS to receive file descriptors for the POSIX shared memory and eventfd file descriptors and then uses those values to transfer the file into the guest.

4.7.1 Using KVM to Accelerate Interrupt Delivery

The choice to use eventfds as the signalling mechanisms for interrupts has one additional benefit specifically related to KVM. In addition to accelerating VMs via the hardware support for virtualization, the KVM kernel module also supports a low-overhead signalling mechanism for VMs. This mechanism consists of two parts, one for sending and the other for receiving interrupts in VMs. The optimized mechanism for sending an interrupt is called *iosignalfd* and the receiving mechanism is called an *irqfd*.

Ioeventfds can improve performance by eliminating heavy-weight VM exits when register writes occur to send interrupts. Rather than requiring a full guest exit, an ioeventfd registers a lightweight exit case that lowers overhead by requiring less state to be saved by the hypervisor when simply writing to an eventfd (which is what Nahanni interrupts do). Nahanni can take advantage of ioeventfds for writing interrupts since they are triggered by writes to the register region of the ivshmem device. Making use of ioeventfds is simple and is enabled by adding `ioeventfd=on` the ivshmem device command-line.

Similar to ioeventfds, irqfds allow interrupt injections into guests that can require less overhead. Use of irqfds requires eventfds as the mechanism that triggers the interrupt. When a write occurs to an eventfd by a guest sending an interrupt to another guest, an irqfd bound to that eventfd will trigger an interrupt into the receiving guest. Similar to ioeventfds, irqfds are enabled by adding `ioeventfd=on` to the ivshmem `-device` parameter.

Ioeventfds and irqfds are optional accelerations support by Nahanni. They do not change the semantics of Nahanni interrupts, only the performance. Both optimizations can be enabled when the VM is launched, but they cannot currently be enabled when an ivshmem device is already attached to a guest.

4.8 Accessing Nahanni Shared Memory from Applications

In this section, we will elaborate on the application programming interface (API) that applications can use to access Nahanni and all its features. Since Nahanni builds upon POSIX

mechanisms on both the host (POSIX shared memory) and in the guest (UIO) interface), a completely new API is not necessary. So in this discussion we will emphasize how to access Nahanni using existing system calls. We will first discuss accessing Nahanni within a guest VM before moving on to the host.

4.8.1 Access From Within a Guest VM

Nahanni is designed to allow access to inter-VM shared memory from guest user-level. Moreover, the choice to use the UIO driver model greatly influences how applications access Nahanni from within guest VMs. As mentioned, Nahanni like any UIO device, is accessed via a file in the /dev file system. For this discussion, presume that a Nahanni device corresponds to the device file `/dev/uio0` (following the naming convention for UIO devices).

As discussed above in Section 4.4.1, the Nahanni device enumerates two memory regions that can be mapped into user-level. The first, memory region 0, contains the four registers also described in Section 4.6.5. The second UIO memory region, region 1, contains the shared memory itself. Recall that while BAR2 references the shared memory in the ivshmem PCI configuration space (see Figure reff:pciconfig), BAR2 is mapped to UIO memory region 1 (see Figure 4.7).

A UIO device-memory region is mapped into user-level using the `mmap()` system call. Recall that, by design, UIO overloads the *offset* parameter of the `mmap()` system call. In `mmap`'s normal usage, the *offset* parameter normally indicates an offset into a file that is to be mapped. For example, mapping 1,024 bytes from a file at offset 512 would map the byte range from 512 up to 1,536. UIO uses the *offset* argument to specify the UIO region to map. An application maps a particular memory region by passing the region number multiplied by the kernel page size:

$$\text{(memory region) \#} \times \text{(page size)}$$

In Linux, the default page size is 4,096 bytes. Linux provides a function named `getpagesize()` that returns the current page size in a way that avoids hard-coding the value. Therefore, a UIO device's memory region n can be mapped by passing an offset of $n \times getpagesize()$. In the case of Nahanni, two memory regions are available to be mapped (the registers and the shared memory itself) so n must be 0 or 1.

Once the register and shared-memory regions are mapped into an application, they may be used by that application like any array and can be passed to functions or made accessible

```
1   int fd;
2   int regs[4];
3   void * shmem;
4
5   fd = open("/dev/uio0", O_RDWR);
6
7   regs = (int *)mmap(NULL, ...,0);
8
9   shmem = (int *)mmap(NULL, ..., 1 * getpagesize());
```

Figure 4.22: Mapping the Nahanni Device-Memory Regions

A simple C program that opens the `/dev/uio0` device file (line 5) and then maps the register region (line 7) and then the shared-memory region (line 9).

---

via a global variable.

Mapping the Register Region

The register region is used to read from and write to the four registers of the ivshmem device described in Section 4.6.5, namely the Interrupt Status Register, Interrupt Mask Registers, VM ID register and Doorbell register. The registers are each 32 bits, so the simplest programmatic access is to access the mapped region as an array of 32-bit integers. By casting the mapped region as shown in Figure 4.22.

Once mapped to an array of integers as in Figure 4.22, the registers can then be read and written via the different array offsets. For example, consider a case when a guest application wants to send an interrupt to a guest with a VM ID of 4. Following the initialization in Figure 4.22, the following code snippet will send an interrupt to VM ID 4.

```
int dest = 4;
int vec = 1;

regs[3] = dest << 16 | vec;
```

The destination VM ID value stored in the integer variable `dest` is shifted 16 bits (since the upper 16 bits of Doorbell register specify the destination VM ID, see Figure 4.16) and then bit-wise ORed with the interrupt vector. The combined value is then written to Doorbell register at array offset 3 (12 bytes) of the `regs` array which mapped the register region of the ivshmem device.

In addition to writing, ivshmem device registers can also be read. The following line will read the VM ID of the guest VM from the VM ID register and store it in the variable

```
my_vmid:
    int my_vmid = regs[2];
```

Shared-Memory Region

The shared-memory region is straightforward in terms of its use. The shared memory can be mapped from the host through QEMU/KVM and then mapped to user-level by the application using the `mmap()` system call following the UIO semantics. Line 9 of Figure 4.22 shows the mapping call that will open the shared-memory object via the UIO driver. If the mapping is successful, the pointer returned can be used like any dynamically allocated memory.

Unlike the register region, reads and writes to the shared-memory region do not trigger any action by the hypervisor. The semantics of how cooperating applications use the shared memory is completely at the discretion of those applications.

Section 4.9 will elaborate how guest applications can make use Nahanni memory including dynamic allocation within the Nahanni memory region and sharing pointers across guest VMs.

4.8.2 From Host Applications

The shared-memory region is a POSIX memory object on the host that is made available to guest applications through Nahanni. The memory object remains accessible on the host after Nahanni guests have accessed it. Any POSIX operations that are valid on memory objects in general can be applied to the Nahanni object as well. An application running on the host, with the appropriate permissions, can open and map the Nahanni object and access it exactly how a guest application would. The only difference in the host case is that the POSIX operations `shm_open()` and `shm_close()` would be used to pass the name of the memory object (the same name that was passed to QEMU/KVM at startup) instead of the UIO interface.

Any host applications that access the shared-memory object must have permission to do so. POSIX shared-memory objects are accessed via the file system and so are protected using regular file protections.

If the SMS is being used with multiple guests, the host applications can also connect to the SMS over the UDS if they know the file name of the socket and have appropriate permissions. Host applications that connect to the SMS will be treated exactly like guest VMs. In particular, they will receive all the necessary resources (a VM ID, interrupt end-

points and the shared-memory region file descriptor) to communicate with the VMs across shared memory, and to send/receive interrupts. The communication endpoints are used just as with eventfds, with their associated semantics, since no UIO interface is necessary for a non-VM application running on the host.

In Section 5.3, we will describe a file staging application that runs on the host will use the SMS and the signalling mechanism to stage a file into a VM via a ring buffer in shared memory.

4.9 Nahanni Memory as Dynamic Memory

Broadly speaking, the semantics of Nahanni shared memory are similar to dynamically allocated memory that will be accessed using pointers. One challenge is how to allocate memory within Nahanni shared memory, that is how can cooperating applications allocate separate regions within Nahanni? Another important question is how can concurrent applications share pointers within Nahanni since the shared-memory region may be mapped to different virtual addresses within the respective applications. We will discuss solutions to these two important questions in this section.

4.9.1 Dynamic Memory Allocation with Nahanni

Being able to dynamically allocate memory from the Nahanni shared-memory region is an important task as the complexity of the applications grow. Fixed buffers could be used in certain circumstances, but applications will inevitably require variable-sized buffers in which to share data. Providing a mechanism to support allocating memory within the Nahanni shared-memory region is an important mechanism for developers building Nahanni applications.

If co-located, virtualized applications may be allocating memory concurrently within Nahanni then self-containment and thread safety are the two most important issues for a shared-memory manager. Self-containment requires that all data and metadata describing the allocation must be stored within the shared memory because applications in different VMs will need to be able to update and modify the allocation metadata. Concurrent allocation within the same Nahanni shared memory by cooperating processes also makes synchronization necessary to ensure metadata is kept consistent. Synchronization ensures that concurrent processes can allocate memory with proper isolation from another.

To address the two issues above, a simple memory allocator library called *shm_alloc* was written for Nahanni by Adam Wolfe Gordon that uses spinlocks for synchronization

and a simple allocation scheme. All metadata is stored in the shared-memory region itself so that all guests can access and update it as allocations and deallocations occur.

When using the shm_alloc allocator all sharing guests must run an initialization step. One guest, the master, must perform additional initialization. A small region of fixed memory at the start of the region is used for synchronizing allocation. The memory allocator provides the expected functions for memory allocation, reallocation and deallocation. Spinlocks contained in the Nahanni memory itself are used to ensure mutual exclusion when updating metadata. The shm_alloc library provides allocation functions that can be used as drop-in replacements for the well-known `malloc()` library of C functions. By using the shm_alloc library, applications can allocate variable-size regions of memory within the Nahanni memory region.

4.9.2 Avoiding Pointer Swizzling

At its lowest level, shared memory must be accessed using pointer-based structures, either arrays or linked structures of some kind. As shown in Figure 4.22, shared memory is accessed by memory mapping the Nahanni region to the applications that need access to it. Typically, when a file is memory mapped, it can be mapped almost anywhere in the mapping application's address space. The location of the mapping is returned from the `mmap()` system call at runtime. In general, it is difficult to predict the address that a file will be mapped to within an application's virtual address space. A file can even be mapped at different locations between different runs of the same program. The variability in mapping locations means that applications will likely have the shared region mapped to different virtual addresses in their respective virtual address spaces. A problem arises then if two cooperating applications wish to communicate via shared memory using absolute memory addresses (i.e., pointers). If the shared memory is mapped to different virtual addresses in each application, then the virtual address within one application, may not be a valid virtual address within the other application.

If the layout of the data to be shared is fixed in size at compile time, the problem can be solved by treating shared memory like a fixed structure or array. Consider an application that shares an array of 1,000 integers. Each application can simply assign an array pointer to the pointer returned from `mmap()` and then both applications will access the array since the offsets of the different array items are consistent. Array item n is stored at $mapaddr + n \times$ $\langle$ *size of an integer* $\rangle$. This approach can work because an array by design uses the concept of offsets. An application can access the array in its address space by adding an offset to the

address returned by the `mmap()` function. Even if cooperating applications map the Nahanni memory to different virtual address, the offset approach will work for both since addresses that are used to access the array are dynamically calculated in each application.

As mentioned, the above case only applies if the size and layout of the data can be completely specified at compile time. But, in a case with two arrays of data to be shared that are of arbitrary length. The first array can begin at $mapaddr$. However the beginning of the second array cannot be known if the length of the first array is unknown at compile time. A straight forward solution is to have one application, Process A, decide where to lay out the data and then pass the layout to the other application, Process B. However, Process A cannot simply pass a pointer to Process B because B may have mapped the shared memory at a different virtual address and the pointer would be invalid. Using offsets can still provide a solution in this case, Process A can pass an offset into the shared-memory region to Process B and B can then add that offset to its map address to get a pointer to the second array within its virtual address space.

A static data structure works well in simple cases. However, when a more complicated structure, such as a linked list, is stored in the shared memory the problem becomes much more difficult. A pointer cannot be stored in shared memory as it would be unclear which application it applies to. A general solution would be to only store offsets in the shared memory itself. Storing offsets will work, however doing so requires constantly calculating offsets before storing pointers and calculating actual addresses after loading from shared memory. Handling pointers from another address space is referred to as *pointer swizzling* [10]. Pointer swizzling is a common challenge when using pointers to memory that are shared between processes each with their own address space. A solution to pointer swizzling for Nahanni exists because the `mmap()` function is used to map the memory into the sharing processes respective address spaces. The first parameter to the `mmap()` function is a pointer. If the value passed is a null pointer, then the function will map the file at any address that is available. If a user has an allocated buffer that she wants the memory mapped onto, that address can be passed (along with the `MAP_FIXED` flag as the fourth parameter). This fixed mapping specifies that if the file cannot be mapped at the provided address then fail. A third use of the pointer is to pass a *hint* address for to map the file at. The mmap function will attempt to map at the hinted location first and then try subsequent locations if the hinted location is not available. If all sharing guests select the same hint address, and that address is available, then no pointer swizzling would be necessary as the memory will be mapped to same address in all sharing guests.

With the availability of 64-bit architectures, the virtual address spaces are now up to 48-bits or 256 terabytes. (At the time of writing, architectures do not support full 64-bit addressing, but 48-bits is common). The key to selecting a hint address that is likely to succeed is to pick an address in the virtual address space that is above the code and heap and below the stack. Both the heap and stack should be small if the mapping is done early in program execution. With a virtual address space on the order of several terabytes and by picking a *hint* address that is in the middle of the address space, it is probable that address will be available, even if several gigabytes of virtual memory are required for the shared memory. In testing with a hint address of 0x100000000000 (256 GB), our test applications have always succeeded. By adding a check after the `mmap()` call to ensure the hint address was used for the mapping, applications are able to read and write pointers for the shared-memory region in the region itself without modification.

On systems where the possibility of hint address failure is more likely, guests could try a sequence of addresses until an address was found that worked for all guests. The shared memory itself could be used to coordinate what addresses were successful.

4.10 Synchronization

Cooperating applications typically require synchronization. Synchronization can be provided by message-passing or by using synchronization mechanisms such as locks, barriers or semaphores. The synchronization requirements and mechanisms that are chosen will vary depending on the application using shared memory.

For many applications synchronization primitives are provided by the OS kernel. However, as discussed previously in Section 3.4, since Nahanni involves multiple, independent guest OS kernels in different VM instances, kernel synchronization is not available. However, user-level synchronization is possible via variables in the Nahanni shared memory and accessible by all cooperating processes.

Given the load-store nature of using shared memory, we expect a common method of synchronization will be to store synchronization variables within shared memory. A condition variable in Nahanni needs to be able to synchronize processes that may be running in different guests. Similar to memory allocation, all the necessary metadata for a synchronization variable must be stored entirely in the shared memory so that separate guest applications can access and update it.

Of course, another synchronization mechanism available within Nahanni is the sig-

nalling mechanism described in Section 4.6. By sending and receiving interrupts cooperating processes could synchronize their execution. However, given that synchronization via the shared-memory region will undoubtedly be important for certain applications, we elaborate on specific mechanisms that work across shared memory, in particular atomic operations.

4.10.1 Atomic Operations in Assembly Language

Support for atomic operations is important for synchronization variables that are accessed via shared memory. Atomically reading and updating a shared value is crucial to implementing synchronization primitives in Nahanni shared memory. For example, lock-free data structures typically rely on atomic operations for synchronization. Atomic operations rely on architectural support from the processor. CPU instructions such as *compare-and-swap* and *fetch-and-add* are two examples of atomic read-modify-write instructions for the x86 architecture. Atomic operations are not widely supported in higher-level languages like C and C++. When atomic operations are not available, assembler linkages or compiler extensions are necessary.

Assembler linkages are typically small sections of architecture-specific assembler code that are linked into higher-level languages like C. An example assembler-linkage that uses atomic compare-and-swap (the `cmpxchg8b` instruction on line 8) is shown in Figure 4.23.

4.10.2 GCC Atomic Operations

The GNU compiler collection supports atomic operations for C through library calls that hide the necessary assembler from the application writer. Compiler *built-ins*, as they are called in GCC, are simpler to use than writing error-prone assembler code by hand and do not require any knowledge of assembler.

For example, the GCC C language built-in for atomic fetch-and-add is

```
__sync_fetch_and_add(&v->counter, i);
```

A fetch-and-add instruction could be used for a counter variable in shared memory that is accessed concurrently by Nahanni applications running in different VMs. GCC supports numerous atomic built-ins for other arithmetic and logic operations. Atomic operations that are provided by GCC extensions or implemented via assembler linkages work in Nahanni shared memory without modification since they are simply memory operations and Nahanni exports flat shared memory.

```
1    /* compiling position-independent code */
2    // EBX register preserved for compliance with position-independent
         code
3    // rules on IA32
4    asm  volatile  (
5          "pushl %%ebx\n\t"
6          "movl   (%%ecx),%%ebx\n\t"
7          "movl   4(%%ecx),%%ecx\n\t"
8          "lock\n\t cmpxchg8b %1\n\t"
9          "popl   %%ebx"
10          : "=A"(result), "=m"(*(int64_t *)ptr)
11          : "m"(*(int64_t *)ptr)
12          , "0"(comparand)
13          , "c"(&value)
14          : "memory", "esp"
15 #if _INTEL_COMPILER
16          ,"ebx"
17 #endif
18    );
```

Figure 4.23: Example assembler linkage for C to implement atomic compare and swap

The above assembler linkage is taken from the Intel Thread Building Block code [25].

4.11 Security

An important issue with any mechanism that allows external processes to access memory (in a VM or otherwise) is security. Following the KVM philosophy of building upon the mechanisms and policies of Linux, Nahanni relies on protection mechanisms provided by the Linux operating system.

4.11.1 Host security

The first concern of security is the POSIX shared-memory object on the host. POSIX shared memory is accessible via the host file system and so is subject to the POSIX file system permissions. In POSIX-compliant file systems, each file and directory is protected by an array of permission bits that are divided into the three groups:

user Permissions for the owner of the file or directory. A file can only have a single owner in the POSIX model.

group Permissions for the group that owns the file or directory. Each file or directory can only have a single group owner. All users that belong to the owning group may access the file or directory as the group permissions allow. Group permissions allow

a restricted group of users access the file without necessarily allowing all users on the system to access file.

other Permissions for all other users excluding the owner and users that part of the owning group. These permissions apply to every user that is known to the file system. These permissions are typically more restrictive that *user* and *group* as they apply to all users that are not the owner of the file or belong to the owning group.

Each of these three categories of users are granted some combination of permissions to *read*, *write* or *execute* the file or directory. The same permissions described above apply to POSIX shared-memory objects. Therefore a single user can restrict access to their own VMs and host applications. As well, users can also expand permissions to other users that are in a common group with them if that user is the group owner for the POSIX shared-memory object. A user could choose to make the shared object available to all users on the system. In each of these three cases, the creator of the shared-memory object could also choose to grant permissions for the group or all users to be read-only.

In general, Nahanni retains the expressiveness of the POSIX permission model in respect to sharing memory between VMs and host applications. Nahanni leaves the choice of permissions to the user that creates the shared-memory object to increase or restrict permissions as they see fit for the particular application.

The POSIX model is sometimes considered overly restrictive by only having three levels of access control. In particular, POSIX does not support different permissions for individual users, other than the owner, or for multiple groups. Another security control mechanism, *Access Control Lists* (ACL), support a more flexible permission model by allowing different users and groups to have separate and different permissions. If and when Linux supports a more flexible permission model such as ACLs for files, directories and, by extension, POSIX shared-memory objects then Nahanni shared-memory objects will be able to take advantage of that mechanism as well.

Security of the Shared-Memory Server

Using the Shared-Memory Server (SMS) does not add any additional security concerns versus the direct method, but there are some differences worth discussing. If users choose to use the SMS, then the SMS is the only application that will be directly accessing the POSIX shared-memory object. Guests wishing to access the shared-memory region will communicate with the SMS to gain access. When SMS is started, it will have the permissions of

the user that started it and so will require the necessary permissions (user, group or other) in order to open the object. The Unix domain socket used to communicate with the SMS is subject to the same file system permission model previously discussed. Each QEMU/KVM process that uses the server will open the socket at boot time and the permissions of that guest will be checked when that occurs.

The above security discussion adds an additional level of concern when requiring guest VMs to run as root, which some of the network configurations (e.g. tap interfaces) discussed in Chapter 5 may require. If all guests are run with root permissions, then they are able to open any shared-memory object and so may open objects that other users did not intend them to use. The root requirement issue for KVM is under regular scrutiny and will be resolved in time, but until that happens certain security trade-offs will exist.

In the meantime, because Nahanni is designed to be used without requiring root permissions (e.g., no need for tap interfaces) to get high-performance IPC, it does not have those security issues.

4.12 Guest Security

In shifting the discussion to security within the guest, it is important to clarify the role of a user. The guest OS is a completely separate OS from the host. In general, user accounts in the guest have no relationship to user accounts on the host. Guest OSes may be configured to match host system user accounts, but that is completely at the discretion of the host and guest system administrators. For the purpose of this discussion, we will refer user accounts that exist in the guest OS as *guest users* and administrators of the VM(s) as *guest administrators*.

Within the VM, a guest administrator may want to restrict access to the shared memory to certain guest users. Inside the guest, the Nahanni memory is accessible through a device file, typically named `/dev/uioN` where N is an integer greater than or equal to 0. Linux file system protections apply in the same way to device files as they do for regular files, directories and POSIX shared-memory objects. Guest administrators can restrict the access to the shared-memory region by setting the appropriate user, guest and other permissions to the device file that is associated with the Nahanni shared memory. With the permissions properly set, only certain users that are either the user owner or in the owner's group can access the `/dev/uioN` file. As well as restricting users, guest administrators can also restrict access to device file by making the device file read-only for either the user owner or group

owner. Doing so would only allow the guest owner to map the shared memory for reading, not writing.

In general, whatever protections are available for device access by the guest OS can be used to restrict access to the Nahanni shared-memory region.

4.13 Discussion

As mentioned in Section 4.2, other designs for Nahanni were possible. One possible design of particular note that was explored in great detail was using the virtio paravirtualization framework to implement Nahanni.

4.13.1 A Virtio-based Nahanni Device

When initially approaching the QEMU community with the idea of an inter-VM shared memory implementation, one of the principal maintainers of QEMU insisted that the interface be implemented in the virtio paravirtualization framework discussed in Chapter 3. As mentioned above, this design and implementation was later deemed inappropriate to the goals of Nahanni, however it is still fruitful to discuss the details of that implementation.

Recalling the three components of the Nahanni implementation discussed in Section 4.1, using virtio would impact the second and third components, namely the Nahanni device and the guest kernel driver. The first component of the design, POSIX shared-memory objects, were still used as the backing for shared memory in our virtio implementation. It is also important to state that the semantics of Nahanni would not change either by using a virtio implementation, just the implementation within QEMU and the guest kernel.

As mentioned, virtio [52] is a standard that was established in 2008 as a generic, high-performance transport for virtual devices. The virtio framework has been used to implement a block device, a network device and serial port that are part of QEMU/KVM. The goal of the virtio framework was to provide a standard device interface that could be used by any data-intensive devices (e.g. block, network, graphics) that move data into or out of VMs. Virtio was designed to behave similarly to a DMA engine in that the hypervisor would copy the data directly from kernel memory to the device (i.e., out of the guest). Virtio, like Nahanni, aims to minimize data copies while data is in transport. However, virtio does not share memory between guest VMs or between guests and the host per se. Instead virtio leverages the fact that the QEMU hypervisor can access any part of the guest memory since the VM is within the QEMU address space. In short, rather than copying to and from an flat, shared region, the guest will notify the hypervisor of the location of data to be moved via

pre-allocated buffers called *virtqueues*. Virtqueues can be used like ring buffers. Typically bi-directional devices, such as network cards, will employ two virtuqueues, one for sending data and one for receiving data. Virtqueues have a fixed number of slots that is configured when the device is created.

Virtio virtqueues are configured by a guest-hypervisor interaction when the virtio device driver is loaded by the guest OS. The interaction consists a series of reads and writes to the device's configuration space that allocate the virtqueues and then notify the hypervisor of their location in the guest memory.

The challenge in trying to create a virtio implementation of Nahanni was to adapt the existing virtio model to support memory regions that could be shared between guests and/or the host.

Since virtqueues are simply regions of guest memory, similarly allocated regions could be used as a target to map memory and have that memory shared directly between multiple guests and the host. Instead of passing the size of a virtqueue, the device passed the size of the memory region that was to be created in guest kernel memory. Once created, the hypervisor was passed the guest address of the allocated memory and the hypervisor then mapped the host POSIX memory object at the guest memory address passed from the device.

Ultimately, the implementation was rejected by the virtio maintainers (despite their initial insistence) because it was said to break the DMA model that was part of the virtio design.

4.14 Concluding Remarks

This chapter has explained how the Nahanni device was designed and implemented. The implementation involves three primary pieces including the ivshmem device, the UIO guest device driver and the Shared-Memory Server. As well, we have highlighted alternative designs that were possible to help explain why we made the design choices we did.

Simplicity and flexibility are the focus in Nahanni's design. Higher-level abstractions may be implemented on top of Nahanni (e.g., memcached [22, 63]) that are more convenient for particular use cases than the mechanisms described above.

At its most basic level, Nahanni can be adapted to numerous uses for virtualized applications because of the flexibility in sharing memory to the user-level which supports a load-store interface for cooperating applications. The addition of a simple interrupt mechanism provides an alternate technique that can replace the need for spinlocks for synchronization.

Chapter 5

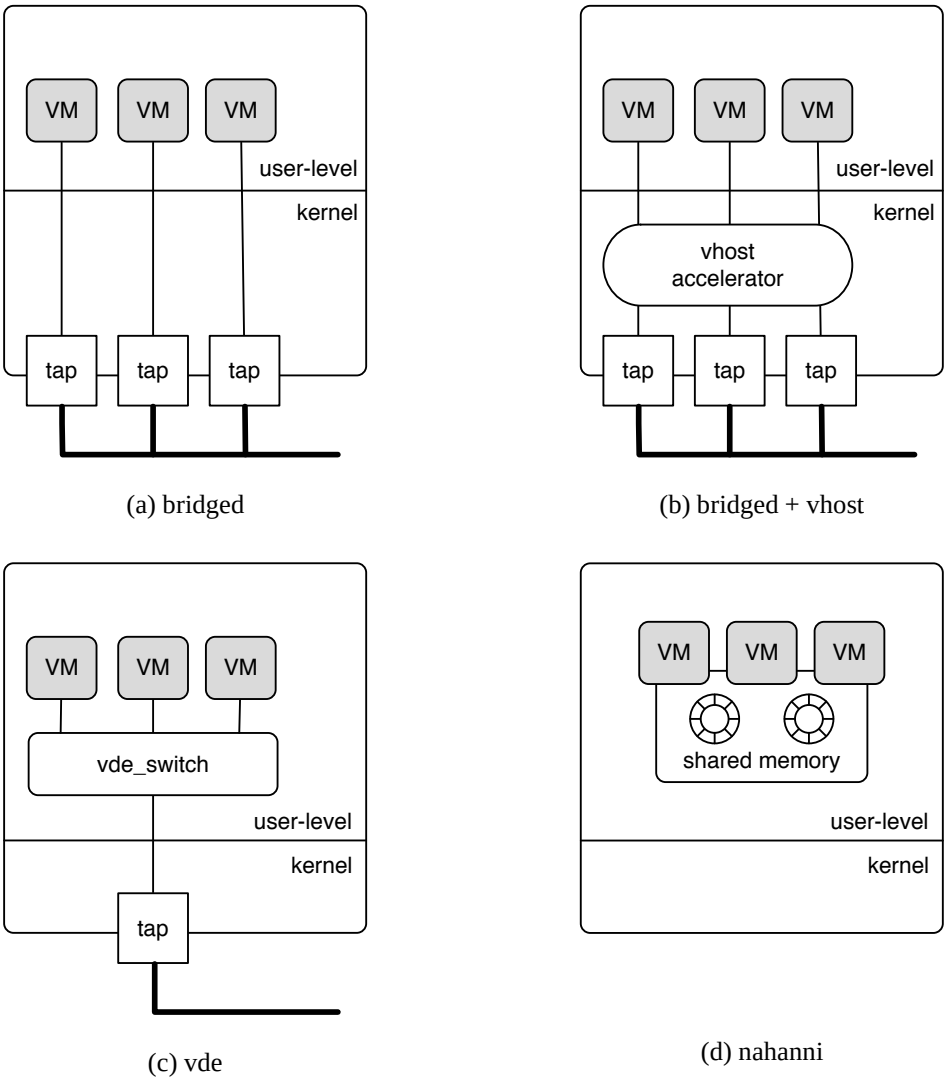
Evaluation

In the previous chapters, the background, design and implementation of Nahanni have been presented. As mentioned, the ultimate goal of Nahanni is to provide a high-bandwidth, low-latency shared memory mechanism that provides excellent performance and enables applications to take advantage of that performance in both streamed data and structured data use cases.

The experiments presented in this chapter will demonstrate that Nahanni provides higher bandwidth (e.g., file staging and message passing) and lower latency (e.g. LMbench's "hot potato"), when compared to existing network and virtio-based mechanisms. First, we discuss a series of microbenchmarks in Section 5.3 that focus on the speed of data movement and synchronization that can be achieved with Nahanni shared memory. Second, small application benchmarks will be shown in Section 5.4 that demonstrate the benefit of Nahanni in the context of applications which intersperse communication with computation. Third, a full application benchmark, the General Atomic and Molecular Electronic Structure System (GAMESS), will be compared using Nahanni versus using the well-known Message-Passing Interface (MPI) to demonstrate Nahanni's benefit in a full-scale scientific application that can use shared memory for interprocess communication (IPC). In our final benchmark, we compare the SPEC MPI2007 benchmark suite using a MPI library modified to use Nahanni against the same MPI library using the virtual network.

5.1 Experimental Methodology

All experiments were run on an 8-way (two quad-core 2.67GHz Intel X5550 Xeons) Linux box with 48 GB of RAM. The host operating system is Fedora 11 and the guests are Ubuntu 9.10 (Karmic Koala). The guests and host both run a 2.6.37 Linux kernel. The hypervisor is the KVM version in the Linux Kernel v2.6.37. In our default configuration, each virtual



(a) bridged

(b) bridged + vhost

(c) vde

(d) nahanni

Figure 5.1: Comparison of inter-VM communication mechanisms for QEMU/KVM

machine (VM) is configured with 4 virtual CPUs and 8 GB of RAM for benchmarking. It will be made clear if any benchmarks require deviating from the default configuration. Depending on the benchmark, we may use 2, 4 or 8 VMs. We use the average of 5 runs for all benchmarks. When error bars are shown, they are equal to 1 standard deviation.

5.2 Definitions

In understanding the experiments, one should be familiar with the following terms some of which have been introduced previously in Chapter 3:

QEMU/KVM QEMU/KVM, sometimes shortened to KVM, is a Linux-based hypervisor

that runs VMs. QEMU/KVM is a two-part solution involving a modified QEMU [8] user-level process and the Kernel-based Virtual Machine (KVM) that runs in the kernel. The KVM part of QEMU/KVM is implemented as a Linux kernel module and has been part of the mainline Linux kernel since version 2.6.20. QEMU/KVM is the default hypervisor in the major Linux distributions Ubuntu and Red Hat Enterprise Linux. To run a VM, a modified QEMU executable running at user-level communicates with the KVM kernel module. For the remainder of this chapter we will use KVM to refer to the QEMU/KVM hypervisor in general.

Virtio Virtio [52] is a standard for implementing paravirtual devices that provides a straightforward, yet high performance transport for virtual devices. Paravirtual devices are a class of devices that are designed specifically for VMs and do not attempt to emulate a real hardware device. Virtio defines an interface between guest drivers and the hypervisor that minimizes copying of data when data moves from the guest to an external device such as the network or virtual disk. Virtio has been used to implement a virtual network card, a virtual block device and a host-guest file system, 9P. Virtio drivers for these devices have been part of the mainline Linux kernel since version 2.6.24. With respect to Figure 5.1, virtio network devices are used for all configurations, however they are only involved heavily in (a), (b) and (c).

Co-located VMs Co-located VMs are VMs that run on the same physical host machine. Co-location is a requirement for using Nahanni as only machines running on the same host can share memory. Co-location of VMs is a common practice for other reasons than allowing shared memory such as increasing resource usage of host machines. Note that all four configurations in Figure 5.1 show three co-located VMs running on a single host machine.

$N \times M$ **Notation** In Sections 5.5 and 5.6, we run several parallel application benchmarks to evaluate the inter-VM communication potential of Nahanni. These benchmarks involve running multiple parallel processes across multiple, co-located VMs. To provide clarity we introduce an $N \times M$ notation to allow us to succinctly express the number of processes and the number of VMs that are executed for each benchmark. The notation $N \times M$ indicates that a total of N processes were run across M VMs. The number of virtual CPUs per VM is N/M. For example, a 4×2 configuration runs 4 parallel processes across 2 VMs. With a 4×2 configuration, each individual VM has $4/2$, or 2, parallel processes executing within it. By contrast, a 4×4 con-

figuration is comprised of 4 parallel processes running across 4 VMs with 1 parallel process per VM. The experiments in Sections 5.5 and 5.6 involve configurations of 4×2, 2×2, 4×4 and 8×8.

KVM's networking configuration has numerous options depending on the requirements of the VMs being run. For example, VMs can be configured to be visible on the hardware network or can be hidden by *network address translation* (NAT). Figure 5.1 illustrates the different communication options available for KVM. The following description will elaborate on the flexibility and trade-offs for the different networking options.

Bridged Networking Using a host network bridge is one possible setup for KVM. Figure 5.1 (a) illustrates this configuration with three VMs. Bridge networking adds a network interface to the bridge for each guest VM via a tap interface (see Figure 5.1 (a)). Bridged VMs are visible on the hardware network (i.e., can be pinged from a different machine) and therefore require an IP address on the network they are bridged on to. Similar to physical machines on the network, IP addresses for the VMs may be statically allocated or dynamically allocated using a DHCP server. One caveat of using bridge networking with KVM is that it requires running VMs with root permissions which opens up a variety of security concerns that are beyond the scope of this thesis.

Vhost Vhost, illustrated in Figure 5.1 (b), is a virtio network acceleration extension for KVM. As Figure 5.1 (b) shows, the vhost setup is similar to a bridged setup with one tap interface per VM. The difference between vhost and regular bridged networking is that vhost requires an additional kernel module to be loaded that accelerates network performance. Vhost improves performance by reducing the number of expensive VM exits when sending or receiving data on the network. Vhost provides a substantial increase in bandwidth and a modest reduction in latency of the network versus virtio alone. Since vhost requires bridged networking it requires running guests as root processes. We consider vhost from a performance point of view. Vhost networking is our main point of comparison for all network benchmarking.

VDE Virtual Distributed Ethernet (VDE) [12], illustrated in Figure 5.1 (c), is a networking system that eliminates the need to run QEMU/KVM VMs as root to perform the necessary network setup. VDE implements a software network switch that performs NAT to allow multiple guests to share the single tap interface on the host. VMs

running with VDE networking are not visible to the hardware network because VDE uses only a single tap interface for all VMs. VDE is a convenient setup because VMs need not run with root permission as is needed with bridged networking and vhost. The convenience comes with the trade-off of poorer performance than bridged networking. More broadly, VDE offers virtualized overlay functionality that is not considered in this work.

Our goal in this chapter is to compare the best possible virtual network configuration to using Nahanni, our shared memory mechanism for communicating between host and guest VMs as well as between guest VMs. Nahanni is illustrated in Figure 5.1 (d). Nahanni does not rely on network connectivity or the virtio subsystem as the other mechanisms do. As described in the previous chapter, Nahanni exposes a region of POSIX shared memory from the host into one or more guest VMs. Host applications and guest VMs that share the same POSIX shared memory object may communicate across it. Nahanni is not a networking technology or optimization, but a different mechanism altogether. Nahanni requires the writing or modification of applications and libraries specifically to use it. Figure 5.1 (d) illustrates one possible use of Nahanni with ring buffers allocated in the shared memory for inter-VM communication. For each benchmark we will describe precisely how Nahanni was used and the modifications that were necessary in order to use it.

For the benchmarks in the upcoming sections, Nahanni (Figure 5.1 (d)) will be compared to vhost-enabled bridged VMs (Figure 5.1 (b)) since the vhost configuration has the best networking performance. For our application benchmark, GAMESS, in Section 5.5, a VDE system (Figure 5.1 (c)) is also included in the comparisons, however our 30% improvement is relative to vhost, which is the best-case performance for MPI.

The microbenchmarks in the following sections will also compare to the virtio-based 9P file system for transferring data from host to guest as well as the network. Note that the solution illustrated in Figure 5.1 (a) is not evaluated because it is architecturally similar to the solution in Figure 5.1 (b), only slower.

5.3 Microbenchmarks

Previous work has shown that shared memory should provide lower latency [60, 20] and higher bandwidth, however we want to verify this hypothesis through microbenchmarks. IPC performance is typically benchmarked in terms of latency and bandwidth, therefore the following two sections will explore these metrics with respect to Nahanni and seek to

answer two fundamental questions:

1. Does Nahanni have lower latency than other mechanisms?

2. Does Nahanni provide higher bandwidth than other mechanisms when moving data?

Nahanni should have two advantages over existing techniques in that it reduces memory-to-memory copies and avoids crossing protection barriers between user-level and kernel inside the guest as well as between guest and host. These two advantages should improve both latency and bandwidth versus existing methods.

5.3.1 Latency: The Hot Potato Benchmark

Our simplest microbenchmark is to compare the round-trip latency of a virtual network versus Nahanni shared memory. We will compare two VMs communicating via the virtual network as shown in Figure 5.1 (b) to two VMs communicating across Nahanni as in Figure 5.1 (d). In this benchmark, a round-trip is the sending of a notification from an application running in a VM to a recipient running in a co-located VM and then receiving a notification back from that recipient. In an application, notifications may be used to transfer control between cooperating applications. Latency should be as low as possible.

We compared the well-known LMbench [32] "hot potato" test to a similar control mechanism in Nahanni between two co-located VMs. The "round trip" in shared memory transfers control of a critical section from one process to another, similar to a semaphore that requires the processes to strictly alternate. We did not use the LMBench code since it is socket-based, but wrote an application from scratch. Two applications running in different VMs will access shared data in the Nahanni shared region that will constitute passing control back and forth. To be clear, no data is exchanged in the Nahanni case. The TCP and UDP LMbench benchmarks send the smallest message possible using the virtual network back and forth between two applications running in different VMs.

LMbench measured a UDP latency of 200 microseconds and a TCP latency of 230 microseconds for its "hot potato" test. Our Nahanni semaphore has a round trip of 0.5 microseconds to transfer control between VMs. This result demonstrates that the latency of shared memory can be much lower since no data is copied and no protection boundaries are crossed which is consistent with previous research [20]. The above results demonstrate an improvement of two orders of magnitude.

The overhead in the networked case is caused by memory-to-memory copies and crossing protection boundaries. The "hot potato" benchmark sends network packets which re-

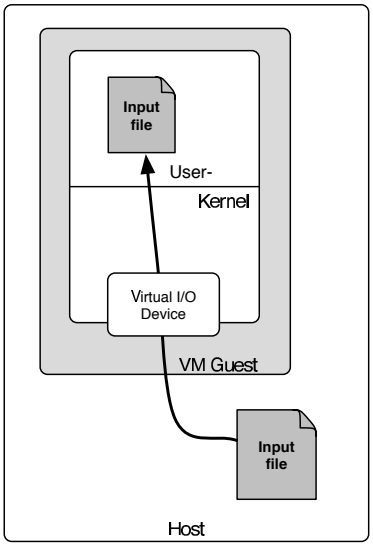


Figure 5.2: Staging a file from the host to guest

This abstracted illustration shows how an input file is copied from the host file system into the guest via a virtual I/O device. The device could be a virtual network card, virtual file system or a Nahanni shared memory device. The input file would be staged in order to be used by a virtualized application running inside the guest.

---

quire system calls on both the sender and receiver as well as the moving of the data packet, either 1-byte for TCP or 4-bytes for UDP, across the virtual network. Using Nahanni to signal an application inside a co-located guest does not require any of the overheads that the network packets incur since the communication is entirely contained within memory that is shared to the user-level in both guests. Avoiding context switches and network transport is what accounts for the much lower latency of Nahanni.

Running Nahanni in a multi-core environment is important for this latency benchmark since it eliminates the need to context switch between VMs because the VMs run on separate cores. Nahanni should still provide a performance advantage on a single-core system, but the latency performance shown here is achieved due in part to the multi-core machine the experiments were run on.

5.3.2 Bandwidth: Host-to-guest File Transfer

The other important metric of IPC performance is bandwidth, that is: being able to move large amounts of data quickly. In high-performance computing, the movement of data files is typically called *staging*. Figure 5.2 illustrates staging of a file from the host file sys-

tem into the virtualized guest. For this benchmark, four different file sizes will be staged: 350MB, 700MB, 2GB and 4GB, using three of the mechanisms illustrated in Figure 5.1 (configuration (a) is excluded) as well as a paravirtualized file system, 9P.

Staging data efficiently is important to keep overhead low when running an application inside of a VM instead of running natively. For example, if a virtual machine is used to decode video, the source video must be staged into the virtual machine. As video files can be large in size, bandwidth is an important metric for this use case. Using Nahanni shared memory that is accessible from user-level in the guest eliminates some of the overheads that other mechanisms impose.

Our bandwidth microbenchmark measures the time to copy a file into a VM using Nahanni versus other paravirtualized mechanisms. In particular we compare a file staging mechanism based on Nahanni (Figure 5.1 (d)) to two well-known network-based file transfer utilities. In particular, we compare a Nahanni-based mechanism that we wrote to the netcat utility [42] and to SSH layered on top of virtual networking as shown Figure 5.1 (b). We also introduce the paravirtualized 9P file system as a point of comparison. The 9P file system is included in these benchmark as it is the suggested method to share host files inside a guest VM when using KVM.

Table 5.3.2 highlights the features of each of the four transport mechanisms that are compared. SSH, netcat and 9P all use the virtio framework for data movement. SSH and netcat use a virtio network device and 9P uses the virtio-9P file system support that is part of QEMU/KVM. Vhost is enabled in the VMs for this benchmark. Briefly, we describe each of these mechanisms to help the reader understand the trade-offs with each.

netcat Netcat [42] is a well known Unix utility for testing applications that use TCP and UDP sockets. Netcat is commonly available on many Unix systems. By design, netcat provides no network security in terms of authentication or encryption of data. Netcat is a useful utility for streaming data and testing networked applications (e.g., clients and servers). For these benchmarks we will use it to copy data across a socket into a VM.

SSH SSH [1] is the well-known secure shell application. For the following experiments we use the SSH-HPN [50] patches to disable encryption of data packets so only authentication is used, to avoid the unnecessary per-byte encryption overheads. Within single servers and private networks, encryption of data during transfer is not essential due to the isolation already present. We leave authentication turned on as authentication

distinguishes SSH from netcat. Two different applications from the SSH suite will be used for benchmarking: SCP (secure copy) for staging a file and SSH (secure shell) for streaming data.

9P 9P [58] is a paravirtualized file system that is designed for the virtio interface that QEMU/KVM supports. 9P is a client/server file system protocol adapted from the Plan 9 operating system [46] that has been ported to Linux. Virtio-9P was added to QEMU/KVM as a mechanism to support accessing the host file system in the guest so virtualized applications access a file on the host without the need to explicitly copy it into the guest. In the case of KVM, the "server" is integrated into QEMU (therefore at user-level) and communicates with a virtio-9P device that is part of the guest. Integrating the server reduced the need for an external server such as SAMBA or NFS on the host. 9P also required adding a special kernel driver to Linux to support the virtio-9P device. The 9P driver must be loaded into the guest kernels used in these benchmarks.

Nahanni As mentioned usage of Nahanni requires applications be written specifically for it. For these bandwidth benchmarks, sender and receiver applications were written that create a simple producer/consumer ring buffer in shared memory. The ring buffer consists of 16 slots with each slot being of a fixed size (16 MB). The sender and receiver applications use Nahanni's interrupt mechanism for signalling. Signalling is used to notify when individual buffers in the ring have been either filled or emptied. The receiver program outputs data to a disk file in the guest or to standard out. Standard out can be redirected to `/dev/null` or another program (e.g. grep, FFmpeg [17]).

The various mechanisms just described also provide different levels of security that in turn have associated overheads. Nahanni relies on standard Unix file permissions (on the host) to protect the POSIX shared-memory object that is shared between guests. Therefore, guest VMs of different users cannot access the same Nahanni shared-memory region unless explicitly permitted to. Netcat, which uses the virtual network, does not provide any security nor is it intended to. Netcat simply listens on a IP port and copies the data sent to it. In general, netcat could not be used in a production environment as it would introduce an unacceptable security risk. SSH uses encryption to authenticate connections in our configuration. For SSH, the High-Performance Enabled version, SSH-HPN was used that allows disabling of encryption on the data transfer (encryption is still used for authentication). In

mechanism	network	paravirtualization	authentication	encryption
Nahanni	not required	none	yes	unnecessary
netcat	required	virtio-net, vhost	no	no
SSH	required	virtio-net, vhost	yes	disabled
9P	not required	virtio-9P	yes	unnecessary

Table 5.1: Comparison of file staging mechanisms
The table indicates the features of each of the transport mechanisms used for file staging and streaming.

---

some environments, not encrypting the transmitted data could be considered a reasonable trade-off. Similar to Nahanni, 9P uses Unix permissions on the host to protect the data that is exported via 9P through QEMU.

To provide an example of how the various mechanisms just described are run, we will provide command-line examples of each. The following command-line is used to stream data from IP port 2000 to `/dev/null` using netcat:

```
netcat -l -p 2000 > /dev/null
```

A similar execution of netcat on the host (not shown) will copy the input file from the host file system to IP port 2000. For SCP, the command-line to copy a file from the host file system into the guest is:

```
scp -oNoneSwitch=yes -oNoneEnabled=yes host:/local/data/inputfile /dev/null
```

In the above example, the file is copied to `/dev/null` inside the guest. The options "`-oNoneSwitch=yes -oNoneEnabled=yes`" tell the SCP application not to encrypt the data. See Section 5.3.2 for the discussion on why encryption is disabled.

For Nahanni, our sender program, named `put_file` is invoked as follows on the host:

```
./put_file /local/data/inputfile 16 1
```

The second parameter in the call, `16`, indicates the number of slots in the ring buffer and the third parameter, `1` indicates the VM ID of the VM that the receiver is running in.

Since 9P is a file system, it is mounted inside the guest VM and exposed via the file system namespace at a mount point. Once mounted, any Unix utility could be used to access the files. For example,

```
cat /mount/9P/inputfile > /dev/null
```

would copy the file from the mount point into `/dev/null`.

Figure 5.3 shows the runtime for each of the mechanisms. To isolate the speed of the

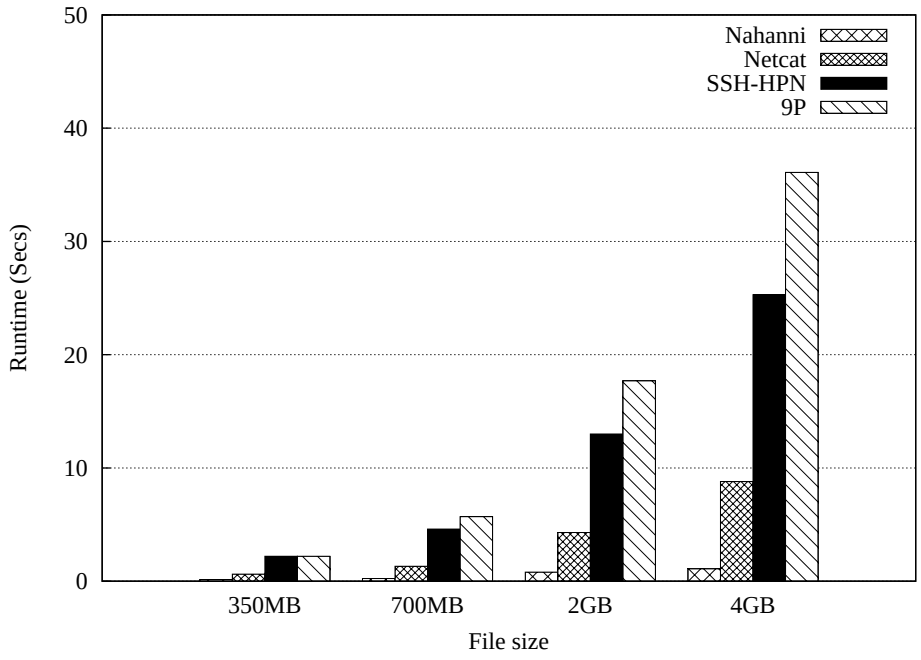


Figure 5.3: Comparison of runtimes for staging data with different mechanisms

different transfer mechanisms, these tests are run with a warm buffer cache on the host and the file is written to `/dev/null` in the guest to eliminate file system overheads.

As the results show, Nahanni is the fastest mechanism for transferring data versus the other mechanism for all 4 file sizes. Copying data across shared memory is between 4 and 8 times faster than netcat. When transferring the 350 MB file, Nahanni completes in 0.14 seconds, where as netcat takes 0.62 seconds, and SCP and 9P both take 2.2 seconds. In general, Nahanni completes at least 4 times faster than netcat and is an order or magnitude faster than SCP or the 9P file system. The graphs for the other file sizes in Figure 5.4 show that these trends continue with the larger files.

It should be mentioned that despite the slower performance, each of the other transport mechanisms have their respective benefits. In particular, netcat and SSH/SCP offer socket semantics such as non-blocking writes and buffering. SSH and SCP also offer network authentication and can access the VM from outside the host machine if the networking allows. 9P offers file system semantics (e.g., a file hierarchy, use of file system utilities). However, all these features come as a trade-off for absolute performance. Lowering the overhead of transferring data will improve the overall execution of data-intensive and latency-sensitive applications.

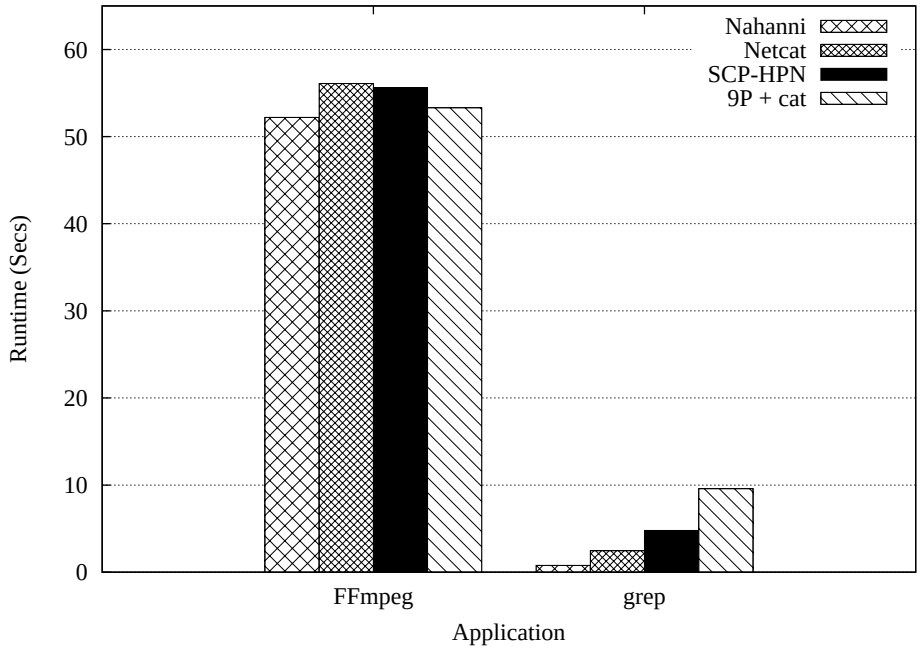


Figure 5.4: Comparison of runtimes for streaming data with different mechanisms

The bars show the runtime of streaming a 700 MB to the FFmpeg and grep applications running inside a VM. The FFmpeg application decodes the video file as it streams. The grep application searches for a word that is not contained in the file.

5.3.3 Summary: Microbenchmarks

Revisiting the questions we sought to ask at the beginning of this section, we can say that Nahanni does provide lower latency and higher bandwidth than other techniques based on the microbenchmark results above. Microbenchmarks demonstrate performance that could be achieved in an ideal case. The next question to answer is whether applications will show a tangible benefit from using Nahanni. That is, we wish to see the performance of Nahanni in the context of being used by an application.

5.4 Benchmarks: Simple Applications

To further understand the benefits of Nahanni, we consider applications that can perform operations on streamed data. Specifically, we consider FFmpeg [17] and the well-known grep utility. We test Nahanni's ability to stream data to these applications when running them inside a VM. We show that Nahanni's latency and bandwidth advantages, as already demonstrated in the microbenchmarks, also improve the overall runtime of these virtualized

applications by between 3-fold and an order of magnitude for grep. However, for FFmpeg, Nahanni improves performance by between 2% and 4%. The difference in relative improvement is an expected result. Nahanni's performance advantage should depend on the application's ratio of computation to data transfer.

The results for these benchmarks are shown in Figure 5.4. For Nahanni, the same ring buffer implementation from the previous section is used to stream the data file into the guest (Figure 5.1 (d)). For these benchmarks, SSH replaces SCP as SSH possesses the ability to stream data into an application whereas SCP can only copy files to and from file systems.

The purpose of staging data into a VM is so the data could be processed by a virtualized application running in that VM. When an application is processing data, the speed of the transfer should keep pace with the speed at which the application can process the data. If an application can process data faster than the data can be transferred, the transport mechanism becomes a bottleneck. This balance is called a *compute-to-data ratio* of the application and it directly affects application performance. Our applications, grep and FFmpeg, were chosen since they have different ratios which are evident in their runtimes as shown in Figure 5.4. While Nahanni continues to show benefit, the amount of benefit depends on the compute-to-data ratio of the streaming application.

For this benchmark we will use the same 700MB file, a video file, that was used in the bandwidth benchmark. The following sections will provide a brief description of each respective benchmark as well as describing the performance of the different transport mechanisms in staging data to the two applications.

5.4.1 Grep

Grep is a common Unix utility that searches for a regular expression within a file. As mentioned, grep is less computationally intensive than FFmpeg and so the overhead of the transport mechanism should be more apparent. For this benchmark we search a 700 MB input file for a regular expression that the file does not contain, so no output is generated. The results are shown on the right histogram of Figure 5.4. Because grep has low computational overheads, the network is still the bottleneck.

We stream data into the grep application for this benchmark through `stdin`, the standard input stream into the application. An example command-line for netcat VM is:

```
netcat -l -p 2000 | grep <word>
```

The netcat program receives the data from the sending program on the host and passes the data to the grep program through a pipe (`|`) to the grep program which searches for

a word that does not exist in the file. The command-lines to execute the other streaming mechanisms are similar to the one above.

Because of the network bottleneck, Nahanni again provides the lowest runtime. With the low compute-to-data ratio of grep, bandwidth is a bottleneck as the runtimes are only slightly higher than from the file staging benchmarks (Figure 5.3). The grep application, when accessing the data through Nahanni, is able to complete in less than one second. The next lowest execution is when using netcat which takes almost 2.5 seconds, followed by SSH (4.9 s) and 9P (9.6s).

5.4.2 FFmpeg

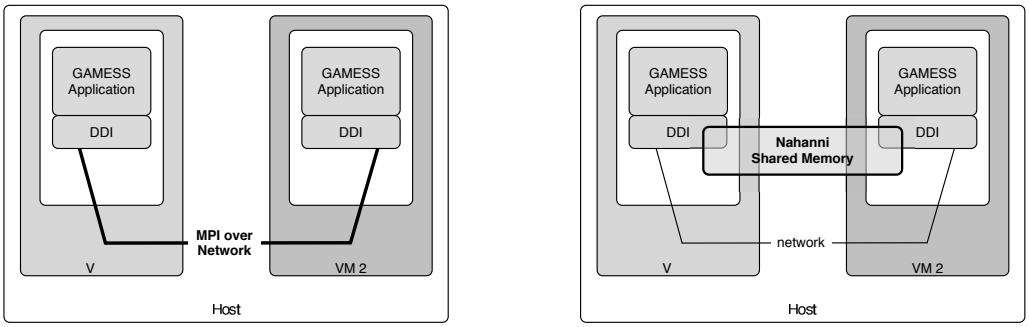
As a benchmark that involves more computationally intensive processing we ran the *FFmpeg* video processing application inside a VM to decode a file that was streamed from the host file system. The input file is a 700MB video file. The left histogram of Figure 5.4 shows the runtimes when using the different mechanisms. The four streaming methods from the previous bandwidth benchmark were used, namely Nahanni, netcat, SSH and the 9P file system, to access the video file. For Nahanni, SSH and netcat the video file is streamed via `stdin` into FFmpeg. For 9P, the file is read by FFmpeg via the 9P file system. Since FFmpeg decodes the video file inside the guest this benchmark represents unidirectional data streaming. The command-line execution to stream a file to FFmpeg from netcat is as follows:

```
netcat -l -p 2000 | ffmpeg -i - <ffmpeg options>
```

Nahanni has the lowest runtime (51.2 s) followed by 9P (53.3 s), SSH (55.6s) and netcat (56.1 s). FFmpeg is a useful benchmark that demonstrates a common application that needs good performance. However, FFmpeg is a relatively computationally-intensive application (as is video transcoding in general) and therefore Nahanni's benefit is limited by the computation overhead of FFmpeg.

5.4.3 Summary: Simple Applications

In this section, we have shown that both grep and FFmpeg achieved their best performance by using Nahanni when streaming input data from the host machine. When using netcat or SSH for staging the data transfer pathways include the networking stacks in the VM and on the host with associated data copying and protection-domain crossing. Copying overheads are minimized in Nahanni as the data is only copied once through shared memory. As with the results from our microbenchmarks, it is Nahanni's ability to minimize memory-to-



(a) GAMESS using MPI over the virtual network (b) GAMESS using Nahanni

Figure 5.5: Two configurations of GAMESS: using MPI (a) and Nahanni (b)

Illustrated are the two VM communication configurations that will be compared using the GAMESS application as a benchmark. The MPI configuration is illustrated in (a) where all inter-VM communication will occur over the virtual network. The Nahanni configuration (b) will use Nahanni shared memory for inter-VM communication. Note that some network communication still occurs in the Nahanni case (b).

---

memory copies and reduce crossings of protection barriers that leads to better performance.

It was also shown that the impact of Nahanni on the total runtime of an application depends on the nature of the application, in particular the ratio of computation to communication. Nahanni's impact was more significant for grep than it was for FFmpeg because of the difference in the compute-to-data ratios of the two applications.

In the next section, we will explore the use of Nahanni in a more complicated scenario than staging files or streaming data. We modify an existing scientific application, GAMESS, to use Nahanni as its communication layer when running across multiple co-located VMs.

5.5 Application Benchmark - GAMESS: Quantum Chemistry

Following the progression from microbenchmarks to stream-based applications, we now investigate whether Nahanni can be an effective inter-VM communication mechanism for a high-performance, parallel application. For this application benchmark we select the *General Atomic and Molecular Electronic Structure System* (GAMESS [62]).

We examine the ability of GAMESS, when run across co-located VMs, to take advantage of Nahanni as a communication layer. GAMESS is selected for this evaluation because it is a well-known, full-sized application (e.g., it is part of the SPEC CPU2006 benchmarking suite), it already has both shared-memory (e.g., DDI) and message-passing (e.g., MPI) implementations, and there is an established community of chemists who use it, including

at the University of Alberta. Although we show that GAMESS using Nahanni (via DDI) can be up to 30.7% faster than GAMESS using MPI, our larger conclusion is that existing applications can be programmed to use Nahanni shared memory, which in turn can have a performance advantage over network-based message-passing.

In this benchmark, we will compare GAMESS using the virtual network (Figure 5.1 (b)) versus Nahanni for inter-VM communication. The 9P file system is not part of these experiments as GAMESS is not able to use a file system for data sharing and 9P is designed specifically for host-to-guest data movement.

For this benchmark and the SPEC MPI2007 benchmarks following in Section 5.6, we will use our $N \times M$ notation introduced in Section 5.2 to describe the inter-VM configurations. All the GAMESS benchmarks in this section use a 4×2 configuration for both Nahanni and the virtual network configurations.

5.5.1 Benchmarking GAMESS

GAMESS is an *ab initio* quantum chemistry simulation program that simulates a wide range of molecular behaviour and properties. GAMESS is typically run in parallel and supports a number of communication subsystems such as sockets, MPI and shared memory. GAMESS is designed to be run in parallel on high-performance shared-memory machines or clusters. GAMESS can be memory-intensive and the ability to communicate efficiently is important to overall performance. The behaviour and performance of GAMESS is dependent on the input provided, namely the input molecule and the simulation to be performed.

The results in Figure 5.6 show the runtime of GAMESS across 4 different simulation inputs. The four input molecules, named *nic-ump2*, *aza-es*, *carbaphos* and *si9h12*, were provided by a chemist and GAMESS developer from the University of Alberta.

GAMESS uses its own communication subsystem called the Distributed Data Interface (DDI). DDI is a library that abstracts the underlying communication system (sockets, MPI, shared memory, etc) from the computational components of GAMESS to provide a cleaner and consistent interface between computation and communication for the GAMESS application itself. The purpose being to minimize development efforts when porting GAMESS to a new communication layer.

For shared memory specifically, DDI supports System V (SysV) shared memory when a simulation is being run on a single machine. When GAMESS is run on a single server or in a single VM, the processes exchange data through SysV shared memory. When run across numerous machines or VMs, DDI uses either network sockets or MPI. In contrast to

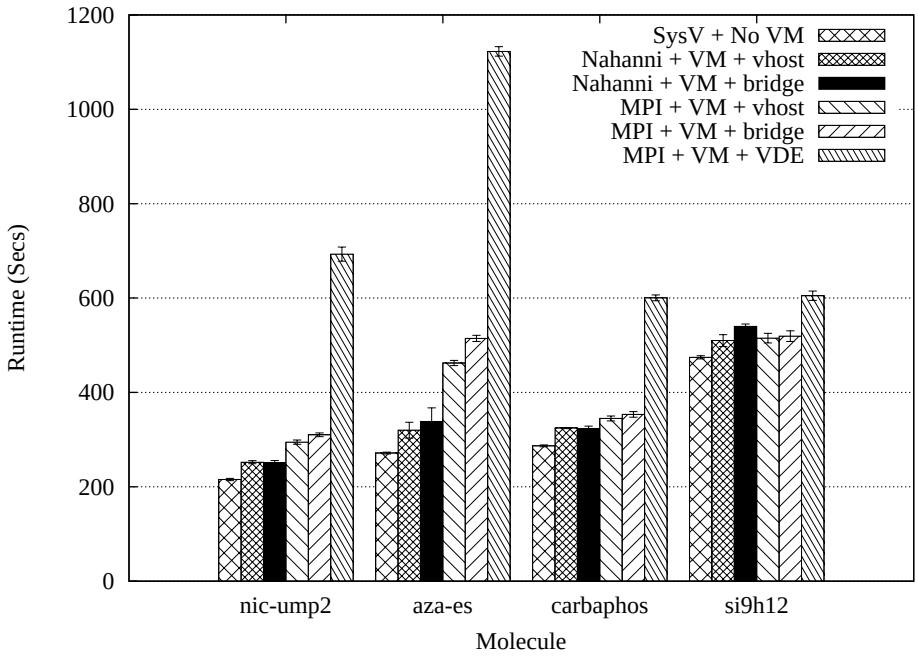


Figure 5.6: Comparison of GAMESS on 4×2 (smaller bars are better)

Note that the performance of Nahanni + VM + vhost is closest to non-virtualized SysV + No VM case.

---

the previous benchmarks, with GAMESS the communication is between guest VMs rather than between the host and guest VMs.

We modified DDI to support communicating over Nahanni if all processes are running on the same hardware host, but in different VMs. This configuration could occur if GAMESS were being run on a cloud provider with multiple co-located VMs running the concurrent GAMESS processes. Figure 5.5 illustrates the GAMESS configurations that we compared. We compare using network-based MPI (Figure 5.5 (a)) for IPC versus using Nahanni (Figure 5.5 (b)) for IPC. It should be noted that even when using Nahanni, some network communication does occur to launch the cooperating GAMESS processes, but the majority of communication will be over shared memory.

The conclusion to be drawn from the results in Figure 5.6 is that Nahanni is the fastest communication mechanism for GAMESS, in the range of 1% to 30.7% faster than the fastest MPI-based variant. That is, the runtime is reduced by up to 30.7% by using Nahanni instead of MPI over the virtual network.

The specific improvement of Nahanni over MPI correlates closely with the amount of

MPI-based communication (Table 5.2). For example, the aza-es molecule for GAMESS has the most MPI-based communication and shows the largest performance improvement of 30.7% (i.e., second vs. fourth bar). We consider an improvement of over 30% a significant result considering the high performance of the virtual network with vhost enabled.

Figure 5.6 compares 6 different configurations of GAMESS across the 4 input molecules. The individual bars in Figure 5.6, from left-to-right, represent the following configurations of GAMESS:

1. running natively on the host with SysV mechanisms (SysV + No VM)
2. virtualized using Nahanni with vhost-enabled (Nahanni + VM + vhost)
3. virtualized using Nahanni without vhost-enabled (Nahanni + VM + bridge)
4. virtualized using MPI with vhost-enabled (MPI + VM + vhost)
5. virtualized using MPI without vhost-enabled (MPI + VM + bridge)
6. virtualized using MPI with VDE (MPI + VM + VDE)

We evaluate the non-vhost cases to demonstrate the performance advantage of vhost. Recall that vhost is a network accelerator for bridge networking (Figure 5.1 (b)) that improves on the performance that regular bridged networking provides.

As mentioned all runs of GAMESS use a 4×2 configuration. The leftmost bar (SysV + No VM) is the native execution time, which is the baseline case without any virtualization or virtual network overhead since it uses host Linux processes and SysV shared memory only. We would not normally expect any VM to be faster than the leftmost bar since it is the non-virtualized case.

The two bars of most interest are the second (i.e., Nahanni + VM + vhost) and fourth bar (i.e., MPI + VM + vhost). The second bar is the best Nahanni performance and the fourth bar is the best MPI performance. Both of these configurations are run with vhost optimization enabled. Nahanni does not use the network nearly as much as MPI since MPI uses the network for all data movement and Nahanni only uses it for startup and synchronization. Notably, the second bar is always (within the error bars) the fastest VM data points, only the leftmost bar is faster which is the non-virtualized case. Due to the overhead of using MPI over the virtual network (instead of Nahanni), the fourth bars show that MPI is slower than Nahanni. Note that Nahanni and MPI both benefit from using vhost, although MPI benefits

	% reduction in runtime	% exec time spent in MPI
nic-ump2	14.4	18.1
aza-es	30.8	35.5
carbaphos	5.9	6.5
si9h12	1	3.6

Table 5.2: GAMESS: Speedup and percentage execution spent in MPI

Each row in the table shows the reduction in runtime from using Nahanni and the percentage of execution time spent in MPI functions across the four GAMESS inputs. The mpiP profiling tool, used within VMs, was used to determine MPI execution percentage.

---

more as it makes much greater use of the network, but vhost is not a deciding factor between Nahanni and MPI.

We include the sixth bar (i.e., MPI + VM + VDE) in Figure 5.6 for completeness. Note that our claimed advantages of up to 30.7% for Nahanni are not for the much-slower VDE cases. Although VDE is not universally used by the KVM community, it is a standard command-line option for QEMU, and without VDE the QEMU/KVM hypervisor would require root privileges to set up the virtual network (i.e., vhost or network taps), as discussed earlier. Our group considers running QEMU/KVM as root in the common case to be impractical (for security reasons), so we normally use VDE. Using vhost under root permissions has similar security concerns, but we are mainly making a performance comparison between Nahanni, vhost, and VDE.

To explore why the speedups varied between the different simulations, we used an MPI profiling tool, mpiP [39] to profile the MPI execution. mpiP is a statistical profiling tool for MPI that gathers data at regular intervals. mpiP measures the percentage of the total execution time that is spent in MPI functions overall as well as within individual MPI functions. Table 5.2 shows the reduction in runtime that is gained from using Nahanni and time spent in MPI functions for each of the simulations. The first column shows the molecule simulated in GAMESS. The second column reports the reduction in runtime from communicating via Nahanni versus MPI over the virtual network. The third column reports the time spent in MPI functions for as reported by the mpiP profiling tool. By comparing the second and third columns, it is clear that the reduction in runtime gained from using Nahanni strongly correlates with the amount of time spent in the MPI functions. This correlation demonstrates that Nahanni saves the intercommunication overhead of MPI by allowing the

parallel tasks to communicate efficiently through shared memory.

The GAMESS benchmark differed from the microbenchmarks and simpler applications discussed earlier in that making use of Nahanni required modifying the code of the application, specifically the DDI layer that implements the IPC within GAMESS. In the next section we will describe the changes that were necessary in order for GAMESS to make use of Nahanni.

5.5.2 Modifying GAMESS

To enable the comparison of GAMESS using Nahanni versus MPI, it was necessary to modify GAMESS to run across Nahanni (instead of the virtual network) when VMs (each running GAMESS processes) are co-located. As mentioned, when running on a single OS, GAMESS uses SysV shared memory and semaphores for IPC. We decided to convert the single-host SysV mechanisms to use Nahanni across multiple VMs. The following is a description of the changes necessary to DDI to maintain the SysV semantics. Retaining the SysV semantics minimizes the changes that are necessary to the DDI code.

After describing the conversion of the individual mechanisms from SysV to Nahanni, we will describe how these new mechanisms were integrated into the GAMESS code base.

Converting SysV mechanisms to Nahanni

As mentioned, DDI uses SysV shared memory for sharing data and SysV semaphores for synchronization when GAMESS is being run on a single OS. SysV mechanisms are supported by the Linux kernel and therefore provide blocking semantics. In particular, SysV semaphores can block if they cannot acquire a semaphore. Allowing processes to block requires kernel support via the scheduler that will not unblock processes that are waiting for semaphores until those semaphores are available. Having the kernel handle blocking is advantageous because the kernel is aware of all processes and their respective synchronization mechanisms. However, since Nahanni can be simultaneously used by multiple, independent, guest OS kernels in different VM instances (see Section 3.4) current OS schedulers cannot be used to provide blocking semantics. When GAMESS uses Nahanni, multiple GAMESS processes running on different kernels will be cooperating. If they require synchronization, it must be contained entirely at user-level as no single kernel can control all the cooperating processes. Therefore, Nahanni cannot rely on any kernel mechanisms and must keep all synchronization mechanisms entirely at user-level and stored in shared memory. User-level synchronizations are not novel. For example, user-level thread libraries have

been developed, such as GNU Portable Threads. Similarly, spinlocks are a simple mutual exclusion mechanism that require no kernel support, but only atomic operations.

Converting GAMESS to use Nahanni requires switching GAMESS' use of two IPC mechanisms, SysV shared memory and semaphores, to an implementation built upon user-level memory allocation and synchronization primitives that reside on Nahanni memory.

Dynamic Memory Allocation

The first mechanism to convert to work on top of Nahanni was memory allocation. Similar to synchronization primitives, memory allocation is typically handled by the kernel. To support multiple processes that are sharing memory, memory allocation and access must be maintained in the shared-memory region. That is, all allocated data and metadata must reside in the shared-memory region so that all processes can allocate, share and access the memory. Moreover, allocating memory and updating the metadata must be synchronized so that the metadata remains consistent at all times.

Wolfe Gordon [63] created a dynamic memory allocator for Nahanni called shm_alloc. Shm_alloc provides a library interface similar to the Linux `malloc()` library. Mutual exclusion during updates is maintained by using spinlocks.

In the existing implementation of GAMESS, each cooperating processes allocates a SysV shared memory segment and then passes a reference to that segment to the other GAMESS processes in an all-to-all exchange via a socket-based mechanism. By using the shm_alloc library on top Nahanni shared memory in a similar manner, the changes required to GAMESS were not extensive.

Semaphores

The other important mechanism within GAMESS that needed to be adapted to use Nahanni was semaphores. SysV semaphores have slightly different semantics than, say, POSIX semaphores. SysV semaphores can be allocated in groups and allow arbitrary values to be added and subtracted from them. When analysing the GAMESS DDI code, it became apparent that the SysV semaphores were being used to implement ad hoc reader/writer locks. Reader/writer locks have more complicated semantics than basic mutual exclusion locks. Briefly, reader/writer locks will allow multiple readers into a critical section simultaneously as long as no writers are in the critical section. Once a writer requests access, all readers must exit the critical section before the single writer will be allowed to enter and given exclusive access (from other writers and readers). Depending on the particular implementa-

tion, reader/writer locks also offer varying guarantees of fairness since there are two classes of access (readers and writers) instead of one.

Our approach was to create reader/writer locks that could exist in shared memory. This latter requirement means that user-level mechanisms such as spinlocks need to be used for mutual exclusion. Just as before, no kernel primitives can be used single the readers and writers may be in different VMs. Intel's Thread Building Blocks (TBB) library [25] have an implementation of spinlock-based reader/writer locks that, when combined with the Nahanni memory allocator, can provide reader/writer synchronization through shared memory. The SysV semaphores are allocated and shared via integer references similar to the SysV shared memory segments. Within GAMESS, they are allocated separately by each process and distributed in a second all-to-all exchange.

Integrating Nahanni into GAMESS

Once the shm_alloc allocator and TBB's reader/writer were chosen as suitable replacements for the SysV mechanisms used by GAMESS, source-level changes were necessary to DDI to have GAMESS use the new mechanisms.

The first change was to have GAMESS use our shm_alloc library rather than SysV shared memory. When using shared memory for communication, GAMESS processes each allocate a region of shared memory for their respective calculations. To enable cooperating GAMESS processes to update eachother's shared memory, references to these regions must be distributed to all other GAMESS processes. This distribution occurs over the network. When SysV shared-memory regions are allocated with the `shmget()` function, an integer is returned that is a reference for that particular region. For example, the following call to `shmget()` stores the reference in the variable `shmid`:

```
if ((shmid = shmget(key,size,flag)) < 0)
```

Another process can gain access to the shared region by passing the same reference to the `shmat()` function (shmat's name is derived from "shmem attach") as shown in this statement:

```
if ((shmaddr = shmat(shmid,addr,flag)) == error)
```

The `shmat()` function returns a pointer that points to the shared-memory region that can then be used like any pointer.

The challenge with converting code such as the above that is targeted for SysV to Nahanni is that Nahanni's memory allocator does not return 32-bit integer references like those returned from `shmget()`, but simply 64-bit memory pointers. Since the SysV system is not

commonly used, it is not worthwhile to convert the shm_alloc library to mimic SysV memory (i.e., to use integer references) as opposed to the much more common `malloc()` library.

Choosing to use 64-bit memory pointers as the references required more wide spread changes as the all-to-all exchange within GAMESS described above is hard-coded to pass 32-bit values. Changing all necessary variables and function parameters to 64-bit values required minor (yet numerous) changes to variables to use 64-bit values. We could have decided to use 32-bit offsets rather than full 64-bit pointers. 32-bit values would support Nahanni regions of upto 4 GB, but handling offsets is inconvenient compared to simply using pointers directly. Also, using 32-bit offsets would enforce an unnecessary limitation of 4 GB of shared memory. Given that we could avoid pointer swizzling using the technique described in Section 4.9.2, we opted to make the necessary changes to support 64-bit pointer values for shared-memory regions. Using pointers also made the calls to the SysV "attach" function `shmat()` unnecessary since the pointers can be used directly to access shared memory.

The adaptation of the reader/writer locks followed similarly to the shared memory changes. In particular, 32-bit integer references had to be replaced by 64-bit pointers. Additional changes were required to change the semaphore *down* and *up* functions to reader/writer *lock* and *release* calls.

Once the shared memory and semaphores were adapted to work with Nahanni, cooperating GAMESS processes communicated across Nahanni for inter-VM IPC. The network is still used for the all-to-all exchanges for the semaphores and shared memory which is only performed once at the start of execution. Figure 5.7 illustrates how the Nahanni shared memory was used by GAMESS. The shm_alloc metadata, semaphores and allocated arrays are shown. The layout of the semaphores and arrays was a result of GAMESS particular dynamic memory semantics.

It is worth restating that Nahanni memory can be used differently than it was with GAMESS. Another application may use, say, fixed offsets with Nahanni instead of the dynamic allocation as described above. Similarly, spinlocks or the Nahanni signalling mechanism could be used for synchronization. Dynamic allocation and reader/writer locks were necessary for GAMESS due to its existing implementation. Nahanni's flexible architecture allows application writers to create the necessary IPC abstractions they require for their application.

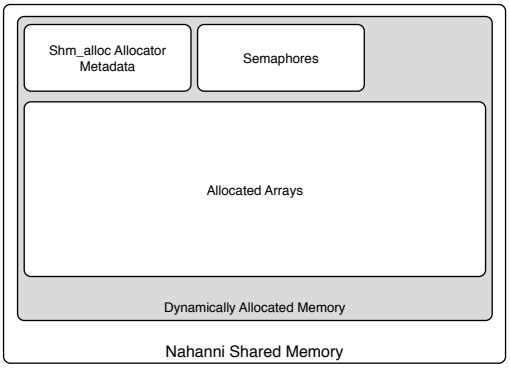


Figure 5.7: The layout of GAMESS structures in Nahanni Shared Memory

5.5.3 Summary: GAMESS

The benchmarks presented in this section serve as an example of modifying an existing application to take advantage of Nahanni. Allowing GAMESS processes to communicate across Nahanni provided an improvement in runtime between 1% and 30% depending on the GAMESS input. The results of this section show that directly modifying an application to take advantage of Nahanni shared memory can be beneficial. We also detailed the modifications that were made to GAMESS to provide an understanding of the effort required to achieve the given performance improvement. In the next section, we will explore modifying an IPC library, namely MPI, to take advantage of Nahanni shared memory and the performance advantage that can be gained when abstracting the use of Nahanni within a library.

5.6 SPEC MPI2007

SPEC MPI2007 is an industry standard benchmarking suite published by the SPEC corporation [56]. SPEC MPI2007 consists of several application benchmarks that are implemented using an MPI library and typically run on a cluster connected by a network, but can also be run on massively multi-core servers (e.g., SGI's Altix UV). All of the benchmarks of SPEC MPI2007 use an MPI library for IPC between parallel processes. The purpose of SPEC MPI2007 is to provide a consistent, application-level benchmark for different MPI libraries. The applications included in SPEC MPI2007 are all open-source and are written in either C, C++ or Fortran.

We modified an existing MPI library called MPICH2 [38] to use Nahanni for IPC between MPI processes that run in co-located VMs. We ran configurations in 2×2, 4×4

Benchmark	Language	Description
104.milc	C	Quantum Chromodynamics
107.leslie3d	Fortran	Computational Fluid Dynamics (CFD)
113.GemsFDTD	Fortran	Computational Electromagnetics (CEM)
115.fds4*	C/Fortran	CFD
121.pop2	C/Fortran	Ocean Modeling
122.tachyon	C	Parallel Ray Tracing
126.lammps	C++	Molecular Dynamics Simulation
127.wrf2*	C/Fortran	Weather Prediction
128.GAPgeofem*	C/Fortran	Heat Transfer
129.tera_tf*	Fortran	3D Eulerian Hydrodynamics
130.socorro	C/Fortran	Density-Functional Theory (DFT)
132.zeusmp2	C/Fortran	CFD
137.lu	Fortran	CFD

Table 5.3: Summary of SPEC MPI2007 benchmarks

The benchmarks that are part of the medium-size input set for SPEC MPI2007. Astrisks (*) indicate applications that are not included in the benchmarking due to compilation or runtime issues unrelated to Nahanni.

---

and 8×8. As a reminder, all these configurations run one MPI process per VM. As we will see, the performance improvement in SPEC MPI2007 runtimes by using a Nahanni-enabled MPI library is between 0% and 22% over MPI running over the virtual network. Most benchmarks see a benefit between 1% and 10%. One application, `pop2`, runs 79% faster when running over Nahanni versus the virtual network on 8×8. Given the fact that KVM's virtual networking is optimized with vhost and considered the "best practice", Nahanni's improvements are significant. We also observe that Nahanni scales better than the virtual network as the number of VMs increases (e.g. 2×2, 4×4, 8×8 configurations). As we will discuss later, Nahanni's scalability is inherent in the scalability of the underlying POSIX shared memory mechanisms, whereas virtual networking approaches have new mechanisms and components with their own scalability issues. In all SPECMPI 2007 benchmarks, we pinned the VMs to CPUs to limit thrashing of the CPU caches. On the 2×2 configuration, we pin the 2 VMs to separate cores. For the 4×4 and 8×8 configurations, we pinned half of the VMs to each respective CPU.

Overall, we have numerous motivations in benchmarking SPEC MPI2007 with Nahanni. First, is to evaluate Nahanni as part of a well-known abstraction, namely MPI. Second, SPEC MPI2007 benchmarks represent full application benchmarks like GAMESS,

but whereas our GAMESS benchmarks compared Nahanni-based DDI with MPI-based DDI, our examination of SPEC MPI2007 compares stream-based MPI with Nahanni-based MPI, so these experiments provide an additional data point. Recall that the motivations for Nahanni include supporting both new or application-specific data-sharing interfaces (e.g., DDI), as well as existing interfaces (e.g., MPI). Finally, the SPEC MPI2007 suite benchmarks are considered quality implementations of message-passing code, whereas GAMESS is primarily known the quality of the computational chemistry in the application, not necessarily its message-passing implementation.

5.6.1 The SPEC MPI2007 Benchmarks

Table 5.3 gives a brief overview of the SPEC MPI2007 benchmarks. The table indicates the programming language each benchmark is implemented in and provides a brief description of the application itself. Table 5.3 shows the benchmarks that are part of the *medium* input set. SPEC MPI2007 benchmarks are divided into two groups, *medium* and *large*, depending on the size of the data inputs that are distributed with the benchmarks. Some benchmarks in the suite include both medium and large inputs and so are part of both benchmark sets. For the experiments in this section we use the medium input set as the memory usage per VM is more reasonable and does not cause our VMs to swap. The medium benchmarks consist of 13 different applications. As will be discussed below, we were able to execute 9 of the 13 benchmarks. The 4 applications we were not able to benchmark encountered compiler or runtime problems unrelated to Nahanni. Note that from this point we will omit the 3-digit numerical prefixes for the SPEC MPI2007 benchmark names.

The Message-Passing Interface (MPI) is a library specification for message-passing IPC that was designed for high-performance parallel applications. The MPI specification was created and is maintained by organizations and individuals involved in high-performance computing. The latest MPI specification is currently version 2.2 which was completed in 2008. There are several implementations of the MPI 2 specification. Running SPEC MPI2007 necessitated modifying one of the available implementations of MPI to use Nahanni for inter-VM communication in order to compare the performance of MPI over Nahanni to the performance of MPI over the virtual network. We chose to modify the MPICH2 [38] implementation of MPI.

Xiaodi Ke, a student working on the Nahanni project, modified MPICH2 to use Nahanni for inter-VM communication. MPICH2 has an optional networking layer, called a *channel* in MPICH2, named *Nemesis* that has optimizations that use memory-mapped shared mem-

Mechanism	Modified MPICH2	Unmodified MPICH2	
	MPI-Nahanni (a.k.a 'Nah')	MPI-vhost (a.k.a 'vhost')	No VM (MPI on host)
MPICH channel	`nahanni`	`nemesis`	`nemesis`
VM	√	√	
vhost	√	√	
network	bridge	bridge	host
Nemesis shmem			inter-process
Nahanni shmem	inter-VM		
Network usage	initialization only	inter-VM	initialization only

Table 5.4: Benchmark configurations for SPEC MPI2007

The three benchmark configurations that are compared using SPEC MPI2007: MPI-Nahanni, MPI-vhost and No VM. MPI-vhost uses the virtual network for all communication and so does not either shared memory transport.

---

ory to accelerate IPC between MPI processes when running on the same host. Ke modified the MPICH2-Nemesis channel to run across Nahanni between co-located VMs. We refer to this modified implementation as MPI-Nahanni (Table 5.4).

Table 5.4 describes the differences between the MPI configurations compared in our SPEC MPI2007 benchmarking. There is a column for each of the three configurations we are benchmarking: modified MPICH2 using Nahanni (MPI-Nahanni) and unmodified MPICH2 using the virtual network between VMs (MPI-vhost) as well as unmodified MPICH2 running on the host without any VMs (No VM). The rows of the table highlight the similarities and differences of each configuration. The table indicates that vhost networking (Figure 5.1 (b)) is enabled in both virtualized configurations. However, with MPI-Nahanni, the majority of inter-VM MPI traffic will be communicated over Nahanni, not over the virtual network. The network is used minimally in the MPI-Nahanni case for initialization. The network is used for all inter-VM communication for MPI-vhost case. The non-virtualized case, No VM, is able to take advantage of the existing shared-memory optimizations of the MPICH2 Nemesis channel for most of the MPI communication, but may also use the host network for initialization. Since the No VM case does not run in VMs, vhost network acceleration and Nahanni shared memory are not available nor necessary.

5.6.2 SPEC MPI2007 Results

In this section, we compare the runtimes of the SPEC MPI2007 medium-input benchmark set when run on the three configurations just described (Table 5.4). To stress the inter-VM interconnect, we run one MPI processes per VM, therefore there is only one compute-intensive process per VM. Consequently, the number of virtual CPUs in each VM is reduced from four to two. As per the MPI-Nahanni implementation, the VMs share two Nahanni memory regions between all VMs that will be used for IPC [27]. Both shared-memory regions are 4 GB each.

Our results that follow in Table 5.5 will show that using Nahanni improves the runtime performance of the benchmarks by an average of 2.2% when running 2×2, by an average 4.3% when running 4×4 and by an average of 5.9% when running 8×8. We also observe that Nahanni scales better than the virtual network as the number of VMs increases. Although improvements of 2.2%, 4.3% and 5.9% are not large in absolute terms, the improvement is proportional to the amount of time spent in MPI functions (i.e., the bottleneck), and we observe increasing returns (as opposed to diminishing returns) as the number of VMs is scaled.

Although we do not have the experimental platform (along with a software environment that we control and can install Nahanni on) to test 16, 32, or 64 VMs, we speculate the trend will continue (Figure 5.11). We also speculate that the any bottlenecks in MPI-vhost are likely fixable but we make the case the Nahanni architecture avoids these bottlenecks entirely. If the bottlenecks in MPI-vhost are the result of particular trade-offs (i.e., bandwidth versus latency), then as the number of cores in servers (e.g., many-core systems [45]) increases over time, there will continue to be limits in scalability. Lastly, as we will see, although the average improvement across SPEC MPI2007 is not large, individual applications can see improvements of 17% or 22%, with an outlier example of a 79% improvement.

As mentioned, we run 9 of the 13 benchmarks (Table 5.3) in the medium benchmark set. We were not able to execute 4 of the medium benchmarks due to either compilation or execution errors unrelated to Nahanni. Specifically, we were not able to successfully compile the `tera_tf` and `wrf2` benchmarks due to a Fortran compiler error using GNU gfortran. The other two excluded benchmarks, `GAPgeofem` and `fds4` encountered runtime errors. The runtime errors occur when using either Nahanni or the virtual network, and so we expect that they are caused by issues with the MPICH2 library and not by Nahanni.

MPI-Nahanni outperforms the MPI-vhost configuration for all three configurations for

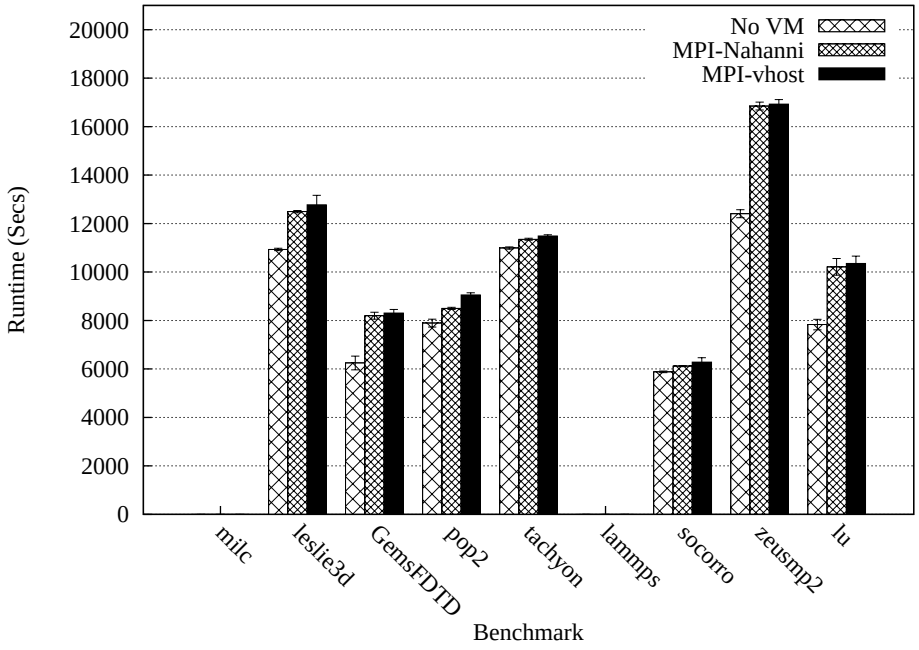


Figure 5.8: Comparison of SPEC MPI2007 on 2×2 (smaller bars are better)

The `milc` and `lammps` benchmarks do not have results due to not being able to run with 2 processes.

each of the benchmarks in the suite (Figures 5.8, 5.9 and 5.10). Table 5.5 summarizes the runtimes and shows the percent reduction in runtime (relative to MPI-vhost) for each benchmark across all three configurations. The reduction in runtime provided by Nahanni ranges from 0.6% to 13.9% with an outlier of 79.3% for `pop2` on 8×8. To explain our reporting method, we consider the `pop2` outlier case. Stated as an equation, the calculation is

$$\frac{|vhost\_time - Nah\_time|}{vhost\_time} = \%\ Improvement$$

As an example, consider the 8×8 `pop2` case where the MPI-vhost runtime is 11,833 seconds and the MPI-Nahanni runtime is 2,453 seconds, therefore the 79.3% improvement is calculated as follows:

$$\frac{11833 - 2453}{11833} = \frac{9380}{11833} = 79.3\%$$

Stated another way, a 79.3% improvement means that `pop2` runs 4.6 times faster with MPI-Nahanni than it does with MPI-vhost.

As a second example, the 8×8 case for `socorro`, a 13.9% improvement, is calculated as

$$\frac{2316 - 1994}{2316} = 13.9\%$$

Returning to overall results, Table 5.5 shows that the two virtualized configurations (MPI-Nahanni and MPI-vhost) are slower than the No VM case except for the `milc` benchmark. When running in the No VM case, the `milc` benchmark had a relatively large standard deviation of 75 seconds, so we do not consider the difference to be significant. We were unable to determine the cause of the high variance in the No VM case for `milc`. We did not see as large variances with either of the virtualized configurations. The other unexpected result was that `zeusmp2` and `lu` experienced larger runtime overheads, 24% and 21% respectively, when virtualized (both with MPI-Nahanni and MPI-vhost) than the other benchmarks. Similarly, we are unsure of the cause of this overhead.

Figure 5.8 shows the performance of the three configurations when running 2×2. Two applications, `milc` and `lammps` would not run with only 2 processes and so their runtimes cannot be shown. In the 2×2 case, there is not a large difference between MPI-Nahanni and MPI-vhost for most applications. `pop2` does see the largest speedup of 6.6% when running across Nahanni. The closeness of the runtimes reflects the quality of the virtual network implementation in that for 2 VMs, there is not a strong advantage for MPI-Nahanni over MPI-vhost. Still, we include the 2×2 configuration for two reasons. One, Nahanni does

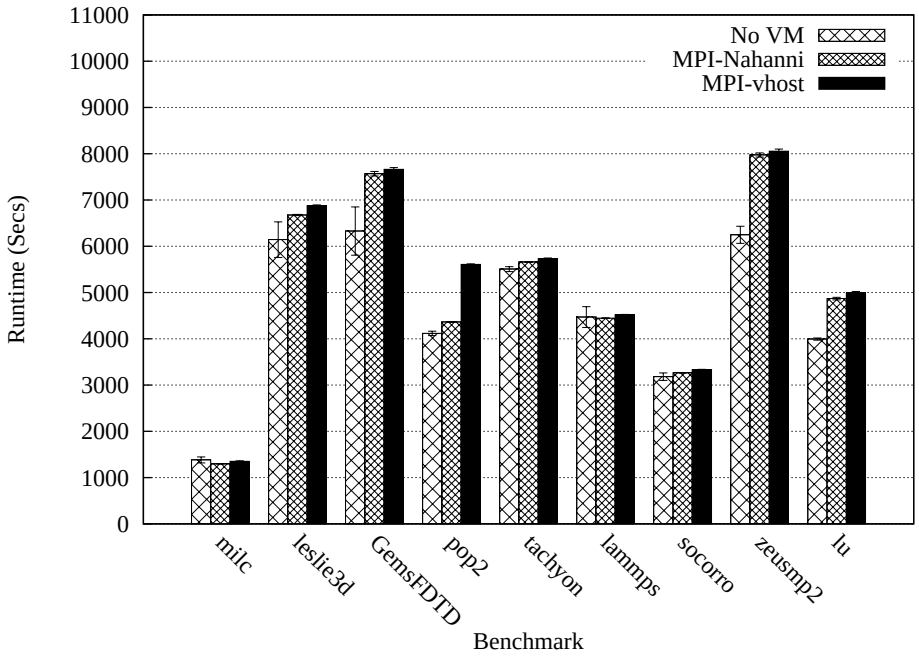


Figure 5.9: Comparison of SPEC MPI2007 on 4×4 (smaller bars are better)

show an advantage for some benchmarks and these results will help in establishing trends as we move to the 4×4 and 8×8 configurations.

Figure 5.9 graphs the results of the 4×4 configuration. MPI-Nahanni's performance advantage emerges as all benchmarks see improved performance from using MPI-Nahanni in the 4×4 configuration. The largest performance improvement is for `pop` at 22%. The average improvement across all benchmarks improves from 2.2% for 2×2 to 4.3% on 4×4. The improved performance can be attributed to Nahanni's scalability as compared to the virtual network.

We increased to 8 processes to further examine the scalability of Nahanni and the virtual network as the level of parallelism and inter-VM communication increases. Figure 5.10 shows the performance when we increase the number of processes and VMs to eight. The only change we make to the VM configuration is reducing the amount of RAM per VM from 8 GB to 4 GB due to having only 48 GB of RAM on our testbed machine.

As with the 4×4 configuration, Nahanni scales better than the virtual network. The performance improvement of Nahanni ranges from 0% (`tachyon`) to 17% (`socorro`). There is one outlier in this case which is the `pop2` benchmark. When using MPI-vhost with 8 VMs, `pop2` runs nearly 4 times slower than when using MPI-Nahanni with 8 VMs. We suspect that the slowdown is due to a scalability issue in the virtual network or vhost.

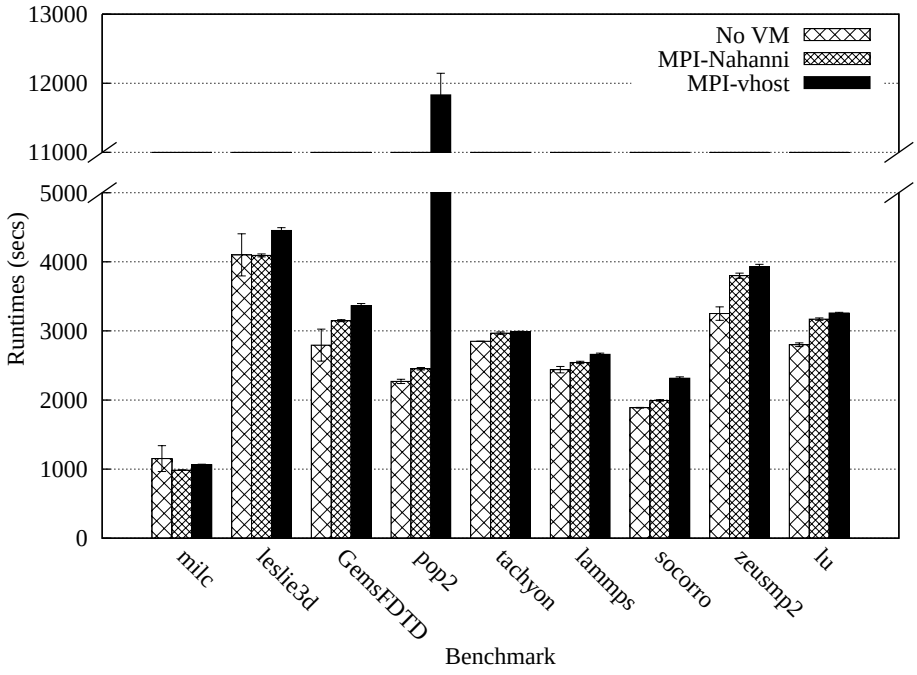


Figure 5.10: Comparison of SPEC MPI2007 on 8×8 (smaller bars are better)

It is worth mentioning that the SPEC MPI2007 benchmarks incur a modest overhead when running under virtualization. Some virtualization overhead is expected. As with the 4×4 configuration, `zeusmp2` and `lu` have slightly higher virtualization overheads than the other benchmarks.

To help focus on the virtualized executions that compare Nahanni to the virtual network, Table 5.5 summarizes the runtimes and percentage difference between running with MPI-Nahanni and MPI-vhost for all 3 configurations. Table 5.5 shows that using Nahanni for inter-VM communication improves the performance of all the SPEC MPI2007 benchmarks. Most benchmarks also so see increasing returns from using Nahanni as the number of VM scales. `socorro`, `tachyon` and `lu` do not scale in relation to the number of VMs, but they still benefit from using Nahanni.

As the number of communicating VMs increases across the VM configurations, the performance benefit of Nahanni also increases as shown by the percent reduction growing for nearly all the benchmarks. The average improvement from using Nahanni increases from 2.2% (2×2) to 4.3% (4×4) to 5.9% (8×8) when the number of processes and VMs is scaled up.

Benchmark	2x2			4x4			8x8		
	Nah	vhost	%	Nah	vhost	%	Nah	vhost	%
`milc`	n/a	n/a	n/a	1298	1352	3.9	985	1063	7.4
`leslie3d`	12493	12777	2.2	6675	6879	3.0	4093	4453	8.1
`GemsFDTD`	8198	8309	1.3	7568	7663	1.2	3148	3366	6.5
`pop2`	8498	9057	6.2	4364	5608	22.2	2453	11833	79.3
`tachyon`	11351	11486	1.2	5658	5733	1.3	2968	2987	0.6
`lammps`	n/a	n/a	n/a	4446	4521	1.7	2544	2662	4.5
`socorro`	6117	6282	2.6	3264	3335	2.1	1994	2316	13.9
`zeusmp2`	16847	16926	0.5	7973	8055	1	3799	3930	3.3
`lu`	10213	10354	1.4	4869	4993	2.5	3169	3261	2.8
average			2.2			4.3			5.9*
minimum			0.5			1			0.6
median			1.4			2.1			6.5
maximum			6.2			22.2			79.3

Table 5.5: Runtimes of SPEC MPI2007 in seconds

The runtimes of the benchmarks are shown for MPI-Nahanni (Nah) and MPI-vhost (vhost) as well as the runtime reduction for MPI-Nahanni (%). The results for 2×2, 4×4 and 8×8 are shown. All reported numbers are the average of five runs. (*) The average for the 8 VM case excludes the 79.3% improvement for `pop2`.

5.6.3 Analysis of SPEC MPI2007

We want to answer the question as to why the advantage of MPI-Nahanni increases as the number of processors and VMs increases, that is, as we move from 2×2 to 4×4 to 8×8.

Upon first considering the speedup results from the previous section, our intuition was that their are potential scaling bottlenecks with MPI-vhost. Specifically, since MPI-vhost exercises virtio pathways with QEMU/KVM it is highly likely one of these pathways has not been highly optimized for multiple, concurrent virtual machines, large amounts of data, frequent interactions or all of the above. There is no easy mechanism or tracing facility to pinpoint bottlenecks in these pathways over the execution of long-running applications like those in SPEC MPI2007, but one would expect such bottlenecks to appear as a greater proportion of time being spent in the communication phases of the applications. Therefore, we performed an analysis using mpiP [39] of the amount of time spent in MPI functions (as we did with GAMESS) for the MPI-vhost case. We wanted to see if the proportion of time spent in MPI functions grows as the number of communicating VMs grows. Of course, by Amdahl's Law [5] one does expect parallel applications to exhibit bottlenecks as the number of concurrent processes/VMs increases, but we argue below that some of the

bottlenecks we have seen go beyond Amdahl's law.

Specifically, the experiments discussed in this section lead us to believe that there are bottlenecks within virtio and vhost in the QEMU/KVM code base. In the previous comparison of MPI-Nahanni and MPI-vhost using the SPEC MPI2007 using the MPICH2 nemesis code base albeit with different configurations (i.e. using Nahanni or the virtual network).

The mpiP results can be found in Tables 5.6, 5.7 and 5.8 for the 2×2, 4×4 and 8×8 benchmarks, respectively. We discuss the results below but our conclusions are as follows:

1. As we scale the number of VMs, the configuration from 2×2 to 4×4 to 8×8 the percentage of time spent in MPI functions, as reported by mpiP, increases.

2. The performance of MPI-Nahanni more closely tracks the performance of MPI-Nemesis on the host.

3. As previously discussed the most dramatic performance difference is `pop2` with the 8×8 configuration (see Figure 5.10). In light of mpiP analysis (Tables 5.6, 5.7 and 5.8) we see the performance gap correlates strongly with the time spent in MPI functions (21.2% with 2×2 versus 83% with 8×8).

Figure 5.11 indicates a trend in the correlation of time spent in MPI functions to the speed up from using MPI-Nahanni instead of MPI-vhost. Figure 5.11 graphs a scatterplot of the all the individual application benchmarks across the three configurations. The x-axis plots the speedup achieved from using MPI-Nahanni (versus MPI-vhost); the y-axis plots the percentage of an application's runtime spent in MPI functions. We have also fitted a linear regression to the data to plot the trend. Figure 5.11 excludes the results for `GemsFDTD` for both 4×4 and 8×8 and for `pop` with 8×8 as we consider these values outliers. For completeness, Figure 5.12 is included which includes the outliers.

Tables 5.6, 5.7 and 5.8 summarize the results of the mpiP analysis 2×2, 4×4 and 8×8, respectively. The second column shows the percentage of total application runtime spent in MPI functions. The second and third columns show the top 2 MPI functions that had consumed the most application runtime. There are some particularly interesting trends. For example, `pop2` is the application that sees the most performance improvement from using MPI-Nahanni. `pop2` speedups up by 6%, 22% and 79% from using MPI-Nahanni for the 2×2, 4×4 and 8×8 configurations, respectively. The majority of `pop2`'s execution on 2×2 and 4×4 is spent in "WaitAll" functions. It is likely that `pop2` is sensitive to barrier functions.

Benchmark	% of runtime	biggest fn	fn % of runtime
milc	n/a	n/a	n/a
		n/a	n/a
leslie3d	1.3	Send	0.9
		Send	0.2
GemsFDTD	4.5	Sendrecv	3.9
		Sendrecv	0.3
pop2	7.9	Waitall	2.8
		Waitall	2.3
tachyon	0.4	Waitall	0.4
		Testsome	0.0
lammps	n/a	n/a	n/a
		n/a	n/a
socorro	4.9	Allreduce	1.7
		Send	1.4
zeusmp2	1.7	Waitall	0.3
		Waitall	0.2
lu	3.7	Recv	1.6
		Wait	1.0

Table 5.6: mpiP results for SPEC MPI2007 2x2
The applications are milc and lammps are excluded due to runtime issues (unrelated to Nahanni) when running with 2 processes.

pop2 on 8×8 spends over 80% of its execution in MPI functions and we consider this outlier along with GemsFDTD which sees high MPI execution time in 4×4 and 8×8 but sees little difference between MPI-Nahanni and MPI-vhost. The mpiP analysis for these three outliers indicate the majority of MPI time is spent in barrier functions which we believe indicates a problem related to load-balancing. Since pop2 does not experience this problem on MPI-Nahanni, we posit that MPI-vhost on 8×8 exacerbates the load balancing problem in pop2 which leads to the extreme performance degradation.

Table 5.9 presents the runtime reductions of Table 5.5 along with the mpiP results from Tables 5.6, 5.7, 5.8. These pairings of runtime improvement and MPI time are plotted as scatterplots in Figures 5.11 and 5.12. The specific values are shown in Table 5.9 to indicate the trends for individual applications. In particular, pop2 and socorro's speedups are within a percent or two of the MPI execution time. Other applications such as GemsFDTD show a weaker correlation, however the trend is consistent across all applications that the percentage of execution spent in MPI correlates to the percent reduction from using MPI-Nahanni versus MPI-vhost.

Benchmark	% of runtime	biggest fn	fn % of runtime
milc	3.6	Wait	3.1
		Wait	0.3
leslie3d	2.4	Send	0.6
		Send	0.5
GemsFDTD	44.3	Barrier	24.9
		Sendrecv	17.4
pop2	21.2	Waitall	7.6
		Waitall	5.7
tachyon	0.5	Waitall	0.5
		Testsome	0.0
lammps	1.7	Send	1.1
		Send	0.4
socorro	2.5	Send	1.1
		Allreduce	0.6
zeusmp2	3.5	Waitall	0.4
		Waitall	0.3
lu	1.6	Recv	0.3
		Recv	0.3

Table 5.7: mpiP results for SPECMPI 2007 for 4x4 configuration

Benchmark	% of runtime	biggest fn	fn % of runtime
milc	15.8	Wait	8.9
		Wait	4.2
leslie3d	8.9	Send	1.7
		Send	1.5
GemsFDTD	30.1	Sendrecv	24.2
		Sendrecv	4.9
pop2	82.22	Waitall	33.4
		Waitall	32.1
tachyon	0.3	Waitall	0.3
		Testsome	0.0
lammps	7.1	Send	4.1
		Send	2.3
socorro	15.2	Waitany	10.0
		Allreduce	1.9
zeusmp2	7.7	Waitall	0.6
		Waitall*	0.5
lu	5.0	Recv	1.6
		Recv	1.5

Table 5.8: mpiP results for SPECMPI 2007 for 8x8 configuration

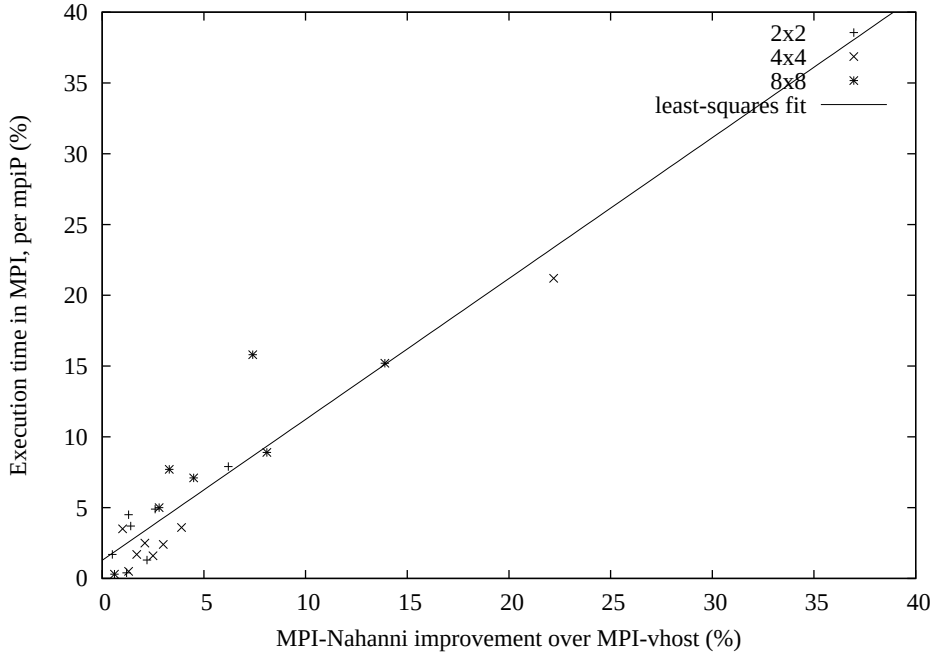


Figure 5.11: Scatterplot of SPEC MPI2007 with outliers removed

A scatterplot of the SPEC MPI2007 benchmarks. The y-axis plots the percentage of execution time spent in MPI functions and the x-axis plots percentage of improvement (i.e., runtime reduction) from using MPI-Nahanni versus MPI-vhost. We consider the results for GemsFDTD on 4x4 and 8x8 as well as the pop2 result on 8x8 to be outliers and so they are excluded in this graph. A linear regression is also plotted to show the trends of the two measures.

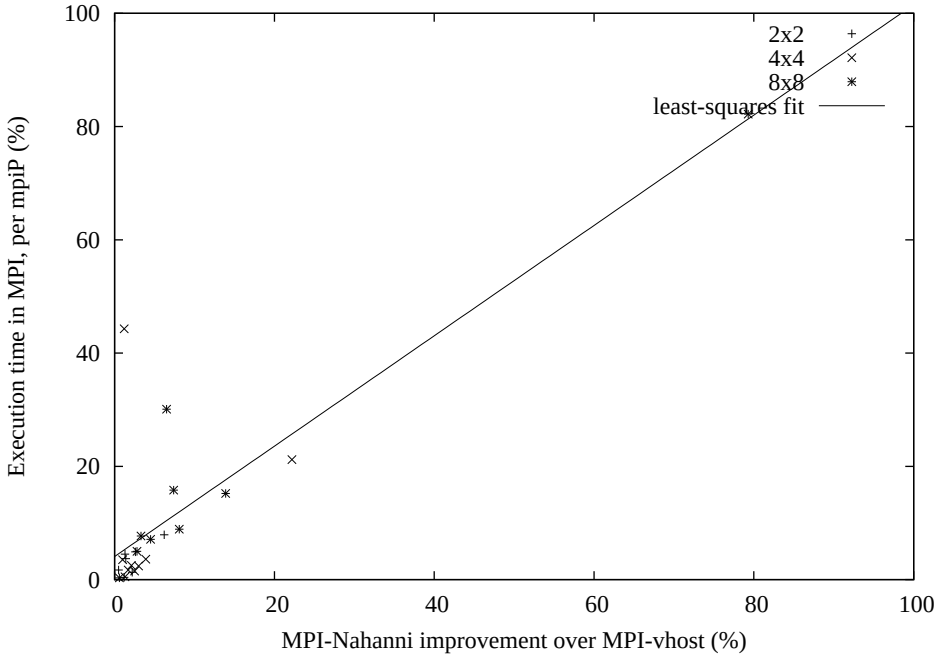


Figure 5.12: Scatterplot of SPEC MPI2007 with outliers included

A scatterplot of the SPEC MPI2007 benchmarks. The y-axis plots the percentage of execution time spent in MPI functions and the x-axis plots percentage of improvement (i.e., runtime reduction) from using MPI-Nahanni versus MPI-vhost. The 79.3% improvement for `pop2` is what causes the dramatic change in the graph. A linear regression is also plotted to show the trends of the two measures.

Configuration	2x2		4x4		8x8	
	speedup (%)	MPI time (%)	speedup (%)	MPI time (%)	speedup (%)	MPI time (%)
`milc`	n/a	n/a	3.9	3.6	7.4	15.8
`leslie3d`	2.2	1.3	3.0	2.4	8.1	8.9
`GemsFDTD`	1.3	4.5	1.2	44.3	6.5	30.1
`pop2`	6.2	7.9	22.2	21.2	79.3	82.2
`tachyon`	1.2	0.4	1.3	0.5	0.6	0.3
`lammps`	n/a	n/a	1.7	1.7	4.5	7.1
`socorro`	2.6	4.9	2.1	2.5	13.9	15.2
`zeusmp2`	0.5	1.7	1	3.5	3.3	7.7
`lu`	1.4	3.7	2.5	1.6	2.8	5.0

Table 5.9: SPEC MPI2007: Speedup and percentage execution spent in MPI

The runtime reductions from using Nahanni and the percentage of execution time spent in MPI functions across the 9 SPEC MPI2007 benchmarks. The mpiP profiling tool, used within VMs, was used to determine MPI execution percentage. This table is a summary of the data presented in Tables 5.5, 5.6, 5.7 and 5.8.

5.6.4 Summary: SPEC MPI2007

The conclusion we draw from these experiments is that Nahanni scales better than the virtual network under the intensive use of SPEC MPI2007. We believe that Nahanni's ability to scale well is due to the use of POSIX shared memory as the underlying mechanism. POSIX shared memory avoids introducing scalability bottlenecks by allowing applications or libraries, such as MPI, to access it directly at guest user-level. Our MPICH2 implementation that uses Nahanni reduces the runtime all of the SPEC MPI2007 benchmarks versus using the virtual network. The margin of improvement from using Nahanni increased as the number of VMs increased from 2 to 4 to 8 showing that Nahanni also scales better than the virtual network. Finally, these experiments show that by modifying an MPI library numerous applications can benefit from Nahanni shared memory by abstracting the use Nahanni shared memory within a library. Such an abstraction eliminates the need for modifications at the source level as with GAMESS, but still provides the improved performance of Nahanni shared memory.

5.7 Other Nahanni Benchmarks

Nahanni has served as the basis for other research not discussed as part of this dissertation. Two Master's students in our research group, Adam Wolfe Gordon and Xiaodi Ke, have

explored other use-cases of Nahanni shared memory.

Wolfe Gordon's research showed that Nahanni can accelerate read accesses to a memcached [37] server that is co-located with VMs that access it [63]. Memcached is a distributed in-memory key-value store that is intended to provide low-latency read and write access to key-value pairs. Key-value pairs are a common data abstraction in web applications. Memcached is used by sites like Facebook [41] to accelerate the access of content.

Wolfe Gordon modified the memcached client and server to cache key-value pairs in Nahanni shared memory. Caching in shared memory allows concurrent access by the server and multiple clients despite the fact that they are running in separate VMs. Co-located virtualized clients and applications benefit from the temporal locality of accesses to key-value pairs. Using Nahanni shared memory to cache key-value pairs reduced the read latency by 29% over using the vhost-accelerated virtual network on the Yahoo Cloud Computing Benchmark [11]. Without vhost enabled for the virtual network, the benefit is shown to be as high as 45%.

Ke modified an MPI library, MPICH2, to use Nahanni for IPC between MPI processes which are executing on co-located VMs [27]. By communicating over Nahanni, the MPI processes avoid the overhead and any scalability limitations of the virtual network. The results from Ke's work demonstrated using Nahanni for IPC between co-located MPI processes reduced latency and increased bandwidth by an order of magnitude over the virtual network. In fact, the MPI-Nahanni implementation that runs across VMs is able to very nearly match the microbenchmark performance of MPI-Nemesis running on the host.

Similar to the benchmarks presented above, Xiaodi used the GAMESS benchmarks from Section 5.5 to compare MPI over Nahanni to MPI over the virtual network. Xiaodi's MPICH2 Nahanni implementation was used for the SPEC MPI2007 benchmarks discussed in Section 5.6.

Ideally, we could have done a head-to-head performance comparison of MPI-Nahanni versus the various systems based on Xen discussed in Chapter 3. Indeed, that head-to-head comparison is planned for future work. For now, to avoid the methodological complexities of comparing two substantially different software platforms (i.e., Xen vs. KVM), we have provided the MPI-Nemesis performance (i.e., without any VM overheads) as a baseline. Given the nature of MPI-Nemesis, it is unlikely that either any Xen-based or KVM-based approach is going to be faster than MPI-Nemesis running without VMs. Furthermore, given how closely MPI-Nahanni's performance tracks MPI-Nemesis, we conclude it is unlikely (short of a direct head-to-head comparison) that any Xen-based approach will be signifi-

cantly faster than MPI-Nahanni.

Our research group continues to explore applications and workloads that benefit from using Nahanni. Building upon the lessons and results from current work, we are examining the uses of Nahanni from several perspectives: inter-VM IPC, host-guest IPC, stream data, structured data, ease of programming, programming abstractions and performance.

5.8 Concluding Remarks

In this chapter, we have demonstrated that Nahanni shared memory can provide a performance improvement, shown by a reduction in total runtime, for both microbenchmarks and full applications. We have also demonstrated that a benefit can be gained from using the shared memory directly within an application (i.e., GAMESS) or by abstracting the use of Nahanni within a library such as MPI (i.e., SPEC MPI2007). For the implementations that use Nahanni in this chapter: file staging and streaming, DDI in GAMESS and MPICH2 for SPEC MPI2007, the ability to access shared memory from user-level was essential to providing low-latency, high-bandwidth inter-VM communication.

Our benchmark results also show that the benefit gained from using shared memory can vary. The GAMESS benchmarks show how the performance benefit that a single application can experience from using Nahanni can vary depending on the input. In particular, the si9h12 molecule simulation saw relatively minimal benefit from Nahanni, whereas the aza-es molecule saw a lower runtime by 30.7% from using Nahanni instead of the virtual network. With SPEC MPI2007, the runtime reductions of the different applications within the benchmarks varied from negligible to over 20% (and in the case of pop2 on 8×8, runtime was reduced by nearly 80%). The variance in performance is dependent on the sensitivity of the application to the latency and bandwidth of the inter-VM communication mechanism.

The results above serve as a guideline for understanding which applications may benefit most from using shared memory between virtual machines. Using Nahanni in applications will require writing applications, or at least libraries, that specifically target it. Our GAMESS results show that it can be worthwhile to modify a program directly. As well, we have also shown that a modified MPI library can abstract Nahanni and enable multiple MPI-based applications to benefit from Nahanni by simply linking a Nahanni-enabled MPI library.

The ongoing growth of cloud computing and virtualization requires an understanding

of virtualized workloads. Virtualized applications and their workloads will also be deployed on different target systems including desktops, servers and cloud environments. The individual needs for inter-VM and host-guest communication performance in these environments will create a spectrum from minimal to extreme. As the scope of virtualized workloads continues to grow, there will be workloads that are well-suited to using Nahanni shared memory and others that see minimal or no improvement. Applications may benefit by using Nahanni for the transport of stream data or for storing structured data directly. The level of benefit will, of course, depend on the application itself.

Chapter 6

Concluding Remarks

Virtual machines (VMs) have been studied as effective platforms for high-performance computing (HPC) where performance is the most critical attribute [18, 36, 53]. The study and deployment of VMs for HPC also led to investigations into the performance of inter-VM communication mechanisms [61, 10, 64, 24]. In some ways, optimized communication mechanisms emphasize performance above all else. After all, without improved performance, there is no reason to consider the optimizations. However, other considerations such as architecture and flexibility are also important and may, in fact, have longer term impacts on the theory and implementation of software systems. For example, optimizations often (by necessity) exploit specific characteristics of the hardware to maximize performance. Over many generations, hardware architecture might change and some optimizations may become eclipsed or even rescinded.

In the context of Nahanni, many of the performance benefits come from an assumption that the hardware supports shared memory, that the operating system (OS) can export shared memory to the user-level, and that the OS pathways and mechanisms are already optimized to share memory between processes. We feel that these assumptions are not likely to change in the near future, but there might be an evolution or retargeting of shared memory, OSes, and OS support for shared memory. For example, uniform memory access (UMA) shared memory has evolved to non-uniform memory access (NUMA), and now there have been proposals of asymmetric distributed shared memory (ADSM) [21]. In some situations, hardware-based cache-coherent shared memory is no longer a valid assumption (e.g. ADSM). The specific optimizations within, say, MPI-Nahanni ([27], Section 5.6) and Nahanni Memcached [63] would likely change if cache coherency is no longer available. In that sense, performance optimizations can be the most sensitive to the technology context.

But, some of the architectural elements and design decisions of Nahanni are likely to

outlive the specific optimizations presented and evaluated in this dissertation. For example, the design decision to implement Nahanni shared memory as a paravirtualized peripheral device offers many advantages. First, if a guest virtual machine does not want to use Nahanni, then the Nahanni guest device driver is omitted and no new Nahanni code is executed in that guest VM. Architecturally, Nahanni code is only executed to initialize the shared memory, but not to use it. Nahanni is outside the perform-critical pathways. In fact, if the Nahanni signalling mechanism (Section 4.6) is not used, which is true for our SPEC MPI2007 and GAMESS benchmarks (Sections 5.5 and 5.6), then no Nahanni code is executed for the common case of IPC. In contrast, modifications to the hypervisor pathways in previous work to either provide new application programming interfaces (APIs) (e.g., XenSocket, IVC) or optimizations (e.g., XenLoop, Fido, virtio) have new code interleaved among the common pathways. Recall that a virtio-based version of Nahanni (Section 4.13.1) was implemented but ultimately rejected because interleaved code changes required within the virtio and QEMU/KVM code base were too extensive. The orthogonality of the current Nahanni implementation was a key reason why the Nahanni code was accepted into the QEMU/KVM code base.

Second, as discussed in Chapter 5, introducing new OS pathways can lead to the need to optimize and re-optimize those pathways to solve the next bottleneck. Optimizing performance as we scale the number of VM instances often requires different algorithms and synchronization strategies, analogous to parallelizing sequential applications. In a different dimension optimizations for large and small data transfers can result in new protocols and the tuning of parameters such as ring buffer sizes. For systems such as virtio, XenLoop, XenSocket these algorithms and parameters are at the hypervisor or OS level. In Nahanni, the code changes occur in user-level libraries because architecturally Nahanni does not impose any algorithm in its design, but leaves that to the application. Nahanni's flat region of shared memory introduces no new pathways in the hypervisor (i.e., it uses the existing QEMU memory and PCI device mechanisms) and largely sidesteps the pathway optimization problem noted in, say, virtio. Optimizations are exported entirely to the user-level libraries and applications by Nahanni's UIO interface.

Similarly, the flat region of shared memory design decision embodied by Nahanni has flexibility advantages. The ability to port MPICH2-Nemesis to Nahanni was greatly simplified by the fact that Nahanni, despite being shared memory between VMs instead of between processes, looks and behaves just like the POSIX shared memory already assumed by MPICH2-Nemesis. Furthermore, since Nahanni memory is exported fully to the user

level, all changes to MPICH2-Nemesis, all future changes and optimizations, and all future libraries, only involve user-level code. No guest (or host) kernel changes were specifically needed by MPI-Nahanni. Admittedly, MPI programs have to be either recompiled or re-linked with MPI-Nahanni, but no code changes are required. Finally, although the exposed shared memory of Nahanni might be considered a source of extra complexity (as compared to XenLoop or virtio, which are completely hidden from the user), that complexity can be completely hidden behind a library (e.g., MPI-Nahanni [27], Nahanni memcached [63]).

Therefore, we have demonstrated both the performance advantages of Nahanni, which are proportional to the opportunities for performance optimizations, and can be substantial. But, we also wish to highlight the unique architectural contributions of Nahanni, which were in place before the complementary work of Ke [27] and Wolfe Gordon [63] within our research group.

Another measure of the significance of this work was its acceptance into the official QEMU/KVM code base. Nahanni was merged into the QEMU code base for version 0.13.0. We have received feedback from QEMU users that are experimenting with Nahanni for uses as diverse as caching to virtualization of reflective memory devices. Because of its inclusion in the standard distribution of QEMU/KVM, the Nahanni ivshmem device is available as part of the well-known Ubuntu and Fedora Linux distributions.

In revisiting the contributions laid out in Section 1.1 we are confident that Nahanni and inter-VM shared memory in general occupy an important niche as the use virtual environments continues to grow in desktop, server and cloud environments.

1. **Unintrusive Implementation Architecture.** Nahanni provides a carefully crafted shared-memory mechanism for guest-to-host and guest-to-guest IPC. The design choices allow Nahanni's components, namely the ivshmem device and guest UIO driver to be non-intrusive with no performance impact on VMs that do not use the mechanism. Nahanni also demonstrated scalability in its design. In particular, VMs using Nahanni for IPC for SPEC MPI2007 demonstrated increasing returns as the number of VMs increased.

2. **Low-latency, High-bandwidth Performance.** The overall performance that Nahanni is able to achieve against established best practices is significant. Our benchmarks for file staging and streaming as well as for full applications of GAMESS and SPEC MPI2007 demonstrate the wide variety of virtualized applications that benefit from Nahanni shared memory.

3. **User-level Architecture: Bypass OS and bottlenecks.** Nahanni allows host memory to be shared directly to the user-level within guests. User-level accessibility allows both stream data and structured data use cases for virtualized applications and libraries such as MPI [27] and memcached [63]. User-level access avoids overheads associated with kernel switches or VM exits that other kernel-level or hypervisor-level optimizations may incur. Avoiding kernel and host overheads is important in achieving the best possible performance.

In summary, this dissertation has explored the intersection of VM environments and shared memory. Adapting the well-known interface of shared memory to the continually growing platform of virtualization provides insight in the performance and programming challenges that will arise as the density of CPU cores and memory increases on the desktop, server as well as in nascent cloud platforms. We have demonstrated the utility that well-designed shared-memory interfaces will provide now and in the future.

Bibliography

[1] *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 2–13, New York, NY, USA, 2006. ACM.

[3] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 159–174, New York, NY, USA, 2007. ACM.

[4] Amazon, Inc. Amazon Elastic Computing Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 1967 Spring Joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[7] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[9] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 277–291, 1996.

[10] 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*, pages 25–25, Berkeley, CA, USA, 2009. USENIX Association.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, New York, NY, USA, 2010. ACM.

[12] R. Davoli. VDE: Virtual Distributed Ethernet. In *Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENTCOM)*, pages 213–220. IEEE, 2005.

[13] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the OS traffic jam. In *Proceedings of the 27th IEEE International Conference on Computer Communications (INFOCOM)*, pages 136–140, 2008.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[15] F. Diakhaté, M. Pérache, R. Namyst, and H. Jourdren. Efficient shared memory message passing for inter-vm communications. In *Proceedings of the Euro-Par Workshops*, pages 53–62, 2008.

[16] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–202, New York, NY, USA, 1993. ACM.

[17] FFmpeg. `http://www.ffmpeg.org`.

[18] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 550 – 559, may 2003.

[19] Flexiant Ltd. Flexiant FlexiScale. `http://www.flexiant.com`.

[20] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 208–211. CRC Press, 1994.

[21] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, New York, NY, USA, 2010. ACM.

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

[23] S. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 1999.

[24] 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 1–12, New York, NY, USA, 2007. ACM.

[25] Intel Thread Building Blocks 3.0 for Open Source. `http://threadingbuildingblocks.org/`.

[26] X. Ke. Interprocess communication mechanisms with inter-virtual machine shared memory. Master's thesis, Department of Computing Science, University of Alberta, 2011.

[27] X. Ke, C. Macdonell, and P. Lu. Fast inter-virtual machine message passing using kvm-based shared memory. *In Preparation*.

[28] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE)*, pages 11–20, New York, NY, USA, 2008. ACM.

[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 37–42, 2006.

[30] KVM. Kernel-based virtual machine. http://www.linux-kvm.org/.

[31] H.A. Lagar-Cavilla, J.A. Whitney, A. Scannell, P. Patchin, S.M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*, pages 1–12, 2009.

[32] LMBench - Tools for performance analysis. http://www.bitmover.com/lmbench/.

[33] C. Macdonell. Nahanni: Inter-vm shared memory. In *KVM Forum 2010*.

[34] C. Macdonell, X. Ke, A. Wolfe Gordon, and P. Lu. Low-Latency, High-Bandwidth Use Cases for Nahanni/ivshmem. In *KVM Forum 2011*.

[35] C. Macdonell and P. Lu. Fast Data Movement for Virtual Machines Using Shared Memory. *Concurrency and Computation: Practice and Experience*. Under submission.

[36] C. Macdonell and P. Lu. Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. In *Proceedings of the High Performance Computing and Simulation (HPCS) Conference*, pages 704–710, 2007.

[37] Memcached - a distributed memory object caching system. http://memcached.org/.

[38] MPICH 2. http://www.mcs.anl.gov/research/projects/mpich2/.

[39] mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net.

[40] Nahanni Guest Code Repository. http://gitorious.org/nahanni.

[41] L. Nealan. Caching & performance: Lessons from Facebook. O'Reilly Open-Source Convention (OSCon), July 2008.

[42] Netcat. http://netcat.sourceforge.net/.

[43] J. Nickurak. Slimming virtual machines based on filesystem profile data. Master's thesis, Department of Computing Science, University of Alberta, 2010.

[44] Parallels. http://www.parallels.com/.

[45] D. Patterson. The trouble with multicore. *IEEE Spectrum*, 47:28–32, July 2010.

[46] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[47] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[48] QEMU. http://repo.or.cz/w/qemu.git.

[49] Rackspace US, Inc. Rackspace Cloud. http://www.rackspace.com/cloud/.

[50] C. Rapier and B. Bennett. High speed bulk data transfer using the ssh protocol. In *Proceedings of the 15th ACM Mardi Gras conference*, pages 11:1–11:7, New York, NY, USA, 2008. ACM.

[51] RedHat, Inc. http://www.redhat.com/.

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

[53] S. Santhanam, P. Elango, A. Arpaci-Dusseau, and M. Livny. Deploying Virtual Machines as Sandboxes for the Grid. In *Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2005.

[54] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th USENIX Conference on System Administration (LISA)*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[55] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, sept.-oct. 2009.

[56] SPEC, Inc. `http://www.spec.org/mpi/`.

[57] E. Unal. Virtual Application Appliances on Clusters. Master's thesis, Department of Computing Science, University of Alberta, 2010.

[58] E. Van Hensbergen. Paravirtualized file systems. In *KVM Forum 2008*.

[59] VMware. Virtual machine communication interface. `http://pubs.vmware.com/vmci-sdk/`.

[60] T. Von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, 1995.

[61] J. Wang, K.-L. Wright, and K. Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *17th International Symposium on High Performance Distributed Computing*, pages 109–118, New York, NY, USA, 2008. ACM.

[62] T. L. Windus, M. W. Schmidt, and M. S. Gordon. *Parallel Implementation of the Electronic Structure Code GAMESS*, chapter 3, pages 16–28.

[63] 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)*, Athens, Greece, 2011.

[64] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. Xensocket: a high-throughput interdomain transport for virtual machines. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, pages 184–203, Berlin, Heidelberg, 2007. Springer-Verlag.