Flexible Computing with Virtual Machines

by

H. Andrés Lagar-Cavilla

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Flexible Computing with Virtual Machines

H. Andrés Lagar-Cavilla

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

This thesis is predicated upon a vision of the future of computing with a separation of functionality between core and edges, very similar to that governing the Internet itself. In this vision, the core of our computing infrastructure is made up of vast server farms with an abundance of storage and processing cycles. Centralization of computation in these farms, coupled with high-speed wired or wireless connectivity, allows for pervasive access to a highly-available and well-maintained repository for data, configurations, and applications. Computation in the edges is concerned with provisioning application state and user data to rich clients, notably mobile devices equipped with powerful displays and graphics processors.

We define *flexible computing* as systems support for applications that dynamically leverage the resources available in the core infrastructure, or *cloud.* The work in this thesis focuses on two instances of flexible computing that are crucial to the realization of the aforementioned vision. *Location flexibility* aims to, transparently and seamlessly, migrate applications between the edges and the core based on user demand. This enables performing the interactive tasks on rich edge clients and the computational tasks on powerful core servers. *Scale flexibility* is the ability of applications executing in cloud environments, such as parallel jobs or clustered servers, to swiftly grow and shrink their footprint according to execution demands.

This thesis shows how we can use system virtualization to implement systems that provide scale and location flexibility. To that effect we build and evaluate two system prototypes: *Snowbird* and *SnowFlock*. We present techniques for manipulating virtual machine state that turn running software into a malleable entity which is easily manageable, is decoupled from the underlying hardware, and is capable of dynamic relocation and scaling. This thesis demonstrates that virtualization technology is a powerful and suitable tool to enable solutions for location and scale flexibility.

Dedication

A los que no están pero siguen.

A los que llegan con una sonrisa sin igual.

Y a los que siempre están junto a mí.

Acknowledgements

First of all I must recognize the invaluable help that my academic advisors have provided in guiding my very stubborn self through the quandaries of uncertain research directions, chronic rejection of papers, and the overall existential graduate student angst. In other words, thanks for all the support, and specific gratitude to one Prof. Eyal de Lara for his undeterred willingness to lock horns with me in the domain of ideas on a weekly basis. Gracias, Eyal. Profs. Satya (M. Satyanarayanan) and Mike Brudno have also been tremendous sources of guidance, each in his own characteristic style, and each almost a de-facto co-advisor for a little while. Profs. David Lie, Angela Demke-Brown, Bianca Schroeder and Ashvin Goel have always been around to bounce ideas off. Finally, Prof. Steve Hand was an excellent, candid and insightful external.

On a more practical note, this thesis describes work performed throughout a period of four years in collaboration with a number of co-authors whom I'd like to acknowledge: Niraj Tolia, Adin Scannell, Joseph Whitney, Stephen Rumble, Philip Patchin, and Profs. Dave O'Hallaron, M. Satyanarayanan, Michael Brudno and Eyal de Lara. I am profoundly grateful for the advice and feedback provided by these people throughout these years, and for the help provided when writing papers for submission to peer-reviewed venues.

Friends and occasional collaborators, in Argentina and in Canada, new and old, thanks, always: Jing Su, Adin Scannell, Patricio Simari, Lionel Litty, Sebastian Sardiña and everyone else I forgot or have included already.

Claudia, Beatriz, Gustavo, Abuelos, y ahora Lucas, para ustedes y por ustedes.

Contents

Chapter 1

Flexible Computing

This thesis is predicated upon a vision of the future of computing which is shaped, primarily, by the following three trends:

- First, a pervasively-available, high-speed, wide-area interconnect: the Internet.
- Second, the economies of scale in raw processing and storage power that make widespread deployment of datacenters possible.
- And third, the specialization of user-centric client hardware into relatively less powerful GUI-oriented devices, which we will call *rich clients*.

The confluence of these three trends lends credence to a model with a separation of functionality between core and edges, very similar to that governing the Internet itself. In this model, the core of the computing infrastructure is made up of vast server farms with an abundance of storage and processing cycles. Centralization of computation in these farms, coupled with high-speed wired or wireless connectivity, allows for pervasive access to a highly-available and well-maintained repository for our data, configurations, and applications. Computation in the edges is concerned with provisioning application state and user data to rich clients, notably mobile devices such as smart phones or small portables, equipped with powerful displays and graphics processors.

While sounding admittedly utopian, this vision has the potential of combining the salient characteristics of two previous iterations in our collective computing experience: the mainframe and the personal computer.

The mainframe era made management of computing systems simple(r) by tasking personnel with maintaining a single (complex) computer. Users could connect at will, within the premises of the organization, and access their data and environment with a seamless transition between endpoints or terminals. These endpoints, however, were *thin clients* or *dumb terminals*, which were incapable of providing a proper interactive experience due to a lack of sophistication in their design, as well as exposure of the user interactions to network latency and jitter.

Personal computers provided the average user with a dedicated private computing system. They brought along the excellent interaction capabilities crystallized in modern GUIs, and made possible the application domains of multimedia, office productivity, visualization and design. However, the side-effects of endowing each user with one (and recently more than one) full-blown computer system include botnets, worm spread, widespread bitrot of domestic systems, fatal data loss or compromise, and major administrative burden in the enterprise environment.

Our vision endows users with the ability to leverage excellent interactive performance with rich edge devices, yet preserves a centralized model providing pervasive access to their data and applications residing in the core. The model of computing where applications reside in big clusters accessed via Internet connections – possibly wide-area connections to geographically distant locations – is usually referred to as *cloud computing*.

1.1 To the Point: What is Flexible Computing Then?

In this thesis we define cloud computing as the ability to seamlessly reach into Internet-connected server farms to tap into computation resources exceeding those which are locally available. This notion requires a degree of flexibility from applications, such that they are dynamically able to leverage cloud resources. Our focus is on providing systems support to afford that flexibility to applications. We define *flexible computing* as systems support for applications that dynamically leverage cloud resources.

The scope of this definition is rather vast, and the work presented here focuses on two specific manifestations: *location* and *scale* flexibility. The realization of our vision is critically dependent on these two building blocks. *Location flexibility*, or more simply application relocation, aims to, transparently and seamlessly, migrate applications between the edges and the core based on user demand. When performing interactive tasks the application resides on a rich-client capable of graphics acceleration located on the user end. When the application enters a processing-intensive stage, it is relocated over the wide-area to more powerful hardware, potentially with higher-bandwidth access to a dataset of interest.

Scale flexibility is the ability of applications executing in cloud environments to swiftly grow and shrink their footprint according to execution demands. As more and more of our logic resides in cloud servers, the need for dynamic scale flexibility increases. Many applications are intrinsically in need of scaling as a function of incoming load, underlying processing availability, or other environmental conditions. Some prominent examples include parallel applications, clustered web servers, and data-intensive jobs involving information of interest to the end-users. Such information may be provided from sources such as bodily sensor readings, preference elicitation engines, multi-media annotations, social networking traces, etc. Cloud-based deployment promises the ability to dynamically leverage large amounts of infrastructure resources for these applications. Scale flexibility provides this notion with a well-known API that has simple semantics:

stateful replication of an entire cloud computing element, much like UNIX fork logically copying the entire address space of a parent process to a child process. We achieve these useful and intuitive semantics in an efficient manner by recognizing that the main bottleneck in a cloud or cluster environment is the scalability of the shared network interconnect. Without sacrificing runtime application performance, we show that we can very conservatively utilize the network pipe to replicate executing tasks to multiple physical hosts within sub-second time frames.

1.2 Virtualization

A unifying theme throughout this thesis is the use of virtualization technology. *Virtualization* can be defined as the ability to sandbox an operating system (OS) by using software constructions that provide the OS with the illusion that it is running on top of real hardware in isolation. Conceived initially in the 1960s, and overlooked for a couple of decades, virtualization is now a fundamental building block of a multitude of systems in use today. Virtualization offers a number of desirable properties, such as isolation, security, tight user control of her own sandbox, and ease of configuration and accounting for system administrators, that have fostered its adoption particularly as a cloud computing enabler.

In particular, we identify two properties of virtualization which are of great value for this work. First, by sandboxing an entire software stack, including the OS, a Virtual Machine (VM) encapsulates most internal software dependencies and can effectively *decouple* software from hardware. This property turns hardware resources into fungible entities that can be re-purposed on the fly, by applying new software stacks to them. The second – and highly related – property is that a serialized VM is a "bag of bits" that can be run on any x86 hardware, i.e. on the vast majority of hardware in existence. While there are a few caveats to the above assertion, this property allows for malleable manipulation of computing state, such as the ability to relocate or replay execution, in ways not achieved by techniques operating at the language or OS level.

This thesis demonstrates the use of virtualization to implement systems that provide scale and location flexibility. It presents techniques for manipulating VM state that also turn running software into a malleable entity which is easily manageable and re-purposed on-the-fly, capable of dynamically relocating and scaling. As a consequence, this thesis shows that virtualization technology is the right enabler for solutions to the scale and location flexibility problems. Furthermore, it shows that virtualization can be used for other purposes beyond collocation and isolation, such as the transparently relocation or resizing of computing tasks.

This thesis encompasses two main projects: **Snowbird** and **SnowFlock**. In Snowbird we explore an implementation of location flexibility based on virtualization. The SnowFlock project studies scale flexibility in shared clusters of virtual machines.

1.3 Snowbird: Location Flexibility

We call *bimodal* those applications that alternate between *crunch* stages of resource intensive computation, and *cognitive* stages of involved user interaction. Bimodal applications can be found in scientific computing, bioinformatics, graphics design, video editing, and other disciplines. Consider, for instance, a 3D modeling application. The user spends a significant amount of time interacting with the application through a sophisticated 3D GUI, designing avatars and scripting an animation. Afterwards, she launches a full-scale rendering of the animation, which is an easily parallelizable CPU-intensive task. Similar workflows can be observed in scientific applications in which a calculation-intensive simulation is followed by a GUI-intensive visualization of the results.

The crunch tasks are better served in large servers, where multi-core parallelism can be leveraged aggressively. Furthermore, if the crunch phase consumes a large dataset, execution at a location granting high-bandwidth access to the dataset is desirable. Crunch phases are thus executed removed from end-users, who use thin clients to control the remotely executing application. However, cognitive phases are better served by executing as close to the user as possible. First, thin clients are limited by the propagation delay, which is hard to optimize and ultimately limited by the speed of light and geographical distances. Furthermore, desktop users, and particularly those who use sophisticated GUIs, are used to leveraging local graphics hardware acceleration to provide a smooth, fast, and visually appealing interactive response. Remote execution with thin clients is thus at a fundamental disadvantage when attempting to deliver a crisp interactive experience comparable to that provided by desktop computing as we understand it today.

Bimodal applications are thus torn between two conflicting performance goals, and two conflicting modes of execution. We attack this problem in the first part of this thesis. With Snowbird, we use virtualization to provide application migration in a seamless and transparent manner. This allows bimodal applications to execute locally during cognitive phases and remotely during crunch phases. With virtualization, we provide transparency by preserving the user environment, and allowing unmodified use of legacy or closed source code. Seamlessness is achieved by leveraging live-migration techniques.

Three contributions complete the Snowbird project. First, an interaction-aware migration manager monitors the application's activity to detect changes of modality and to trigger the necessary relocations, particularly those related to a decay in the quality of the application's interactive performance. The migration manager completes the transparency arguments: no changes to applications are required to make them migratable. Second, a graphics acceleration virtualization subsystem allows applications running inside a virtual machine to use hardware acceleration and achieve near-native graphics performance during cognitive phases. Third, a wide-area, peer-to-peer storage subsystem addresses the challenges of migrating the large amount of state contained in a virtual machine image.

1.4 SnowFlock: Scale Flexibility

Cloud computing promises the ability to quickly scale the footprint of applications executing on third-party premises, without troubling the application provider with infrastructure acquisition and maintenance costs. This nascent paradigm is founded on the economies of scale of storage and processor hardware, and the maturity of virtualization technology. We claim, however, that the current practices are based on recycling existing techniques that are inadequate for future uses of the cloud environment.

We identify two problems with existing cloud practices to scale the footprint of an application. To leverage more physical resources, VMs need to be instantiated to populate those resources. VMs are typically created by booting off a template image: this demands the "swapping in" of large amounts of VM disk state, typically in the order of four to sixteen GBs, from backend storage. With multiple GBs of VM state being transmitted over the network, this process takes "minutes" (as anecdotically pointed out in the trade literature by the Amazon Elastic Compute Cloud (EC2) [10] and others), and lacks scalability. This is because the network fabric already sustains intense pressure from application data needs, which scale aggressively with the abundance of processors and disk platters. The interconnect thus becomes one of the most crucial performance bottlenecks in a datacenter, and multiple creations of dozens of VMs by hundreds of users will only compound the latency of application scaling, and result in a lack of scalability.

In addition to the performance constraints of the current techniques, dynamic scaling in cloud computing environments is unreasonably hard to program. An application provider is tasked with dealing with complexity that is not germane to the application logic. First, the developer must find work-arounds for the latency involved in spawning new virtual machines. This task typically involves engines performing load prediction for the purposes of pre-allocating VMs, and also keeping VMs idle once they are not immediately useful anymore. Second, since new VMs are typically fresh boots from a template image, current application state needs to be explicitly pushed to the new pristine VMs –

typically by logging in, copying files, and configuring and starting services.

We propose a cloud API that simplifies the management of spawning new VMs and propagating application state to the new computing elements. SnowFlock implements the notion of VM fork: replication of an entire VM to multiple copies in parallel. Since the objective is application scaling, each VM clone resides on a different physical host, ready to leverage the compute resources available there. SnowFlock mimics the well-understood semantics of UNIX fork: it provides logically instantaneous stateful replication of the entire address space of the computing element, in this case a VM. Full replication of processes and threads is a paradigm that programmers understand very well, and that over decades have used very effectively to program applications that dynamically grow their footprint. Two prime examples of the usefulness of fork include parallel computation and scaling a clustered load-balancing server by growing the worker pools.

SnowFlock thus provides an API that programmers can easily use to develop dynamically resizable parallel tasks in the cloud. To be relevant, SnowFlock must not only provide stateful replication of VMs, but it must also do so with excellent performance. We achieve this by relying on three insights. First, that although the entire address space is logically replicated, child VMs tend to access a small subset of the parent VM state. Second, that memory and disk access patterns exhibit high similarity among clones. Because the VM clones are all involved in the same computation, they tend to need the same pieces of code and data, one of the main reasons why we provide stateful cloning. We exploit that similarity in access patterns to better provide VM state to clones. And third, that after being cloned, VMs tend to include in their working set important chunks of state they generate on their own, and that does not need to be fetched from the parent. SnowFlock combines these insights to enable almost instantaneous (i.e. less than a second) scale flexibility for parallel services running in virtualized clouds, and frugal utilization of shared network resources (for example, less than a hundred MBs of VM state transmitted for 32 replicas of a VM with 1GB of RAM and several GBs of disk).

1.5 Outlook

The contributions in this thesis provide two building blocks for the vision outlined at the beginning. They enable computation to seamlessly move between the edges and the core, according to the role that is most convenient in each case, i.e. resource-intensive computation remotely, and interaction-heavy cognitive loops using rich client hardware locally. For applications residing in the core or cloud, this thesis shows how to efficiently scale the footprint of the task dynamically, provides a natural API that simplifies cloud programming, and addresses applications such as parallel backends or clustered load-balancing servers.

These are two crucial building blocks in our vision, but not the only ones. For example, security is a paramount concern. Cloud facilities offer a prime environment for launching high-scale malicious endeavours without risking identification; we have begun to look at combating these problem [113]. Additionally, our vision pushes for a specialization of hardware and functionality that could lead to a specialization of the software itself. Software running on specialized edge and core devices does not need as much versatility as current platforms, which could lead to efforts in replacing or pruning the functionality of general purpose operating systems.

The use of system virtualization is limited by one important factor: the compatibility of the underlying processor ABI. This has not been in general a major concern, as most hardware in use is x86-compatible, with minor variants. The vision outlined here, however, is susceptible to the emerging importance of ARM-based devices in the smartphone arena, which are simply not binary-compatible with x86 “big iron” [1]. This is an important challenge to our vision, that for some cases might demand shifting our focus up the stack into language-based virtualization (i.e. bytecode interpretation).

[1]a toyish relativism.

1.6 Thesis Statement

To wrap up this opening chapter, we summarize the position of this thesis in the following paragraphs. We then describe the organization of this thesis and briefly introduce the contents of each chapter.

Computing is bound to change fundamentally in the next few years. The traditional model of personal computing will be reshaped by three cooperating forces: the popularity of rich user-centric client devices such as netbooks and smartphones; the pervasive access to high-bandwidth connectivity made possible by the Internet; and the existence of large compute farms made possible by processor and disk economies of scale.

The confluence of these three forces can result in a paradigm shift that promises to combine the best of the previous two iterations of our collective computing experience: the mainframe and personal computing eras. Our applications and data can be hosted in pervasively-accessible, powerful, well-maintained, and highly reliable datacenters, providing the simple yet powerful model of an always-on centralized point of access reminiscent of the mainframe era. The excellent connectivity, processing and graphics power of client devices can allow us to retain the rich and crisp interactive performance of personal computers that made multimedia, design and productivity applications possible. This paradigm of applications residing in the network is called cloud computing.

In this thesis we focus on two fundamental challenges preventing the realization of a cloud computing vision. We first show how applications can be automatically moved between the client devices at the user location (the edge) and the powerful datacenters buried deep within the network (the core.) Applications from domains such as design, engineering, multimedia or scientific computing can, then, seamlessly switch their site of execution to best accommodate the current activity. When designing an animation or visualizing the result of a simulation, applications execute at the user end, eliminating the negative effects of network delay and leveraging graphics acceleration hardware. When performing intensive computations to align DNA sequences or to produce the final rendering of a feature film, applications execute in the datacenter, leveraging compute power far beyond that available to a single user. These relocations are achieved without requiring any changes to existing applications or imposing the use of any programming language, operating system, or software toolkit.

The second challenge pertains to efficient high-performance execution in the core or datacenter. We propose a new approach to scale out applications: in under a second, a single virtual computer encapsulating an application can replicate itself to take over hundreds of processors. This is achieved with semantics that are easily understandable by programmers, closely matching standard operating system facilities in use for over two decades. Our proposed approach is an order of magnitude more efficient in its utilization of network resources, the fundamental bottleneck in a datacenter, and is thus in turn an order of magnitude more efficient in its capability to scale an application to hundreds of computing nodes. Our work opens the door to much more efficient and opportunistic use of datacenter compute power by applications such as parallel jobs, clustered web servers, or computing tool chains. It also enables new applications, such as instantaneous disposable clusters, on-the-fly debugging of distributed systems, and opportunities to save energy by turning off servers that are now idle thanks to the increased efficiency.

Both challenges described here, and their solutions, share a common thread: the ability of systems software to allow applications to dynamically leverage the resources of the cloud computing model. We call this ability flexible computing.

The other common topic impregnating this work is the use of system-level virtualization. VMs have the ability to encapsulate an entire software stack, serialize it, and allow it to run at any location supporting the x86 ABI, without any other software or hardware demands. This powerful property allows for the use of sophisticated techniques when manipulating VM serialized state that enables many applications, such as the two presented in this thesis: the ability to dynamically relocate and scale applications within a cloud framework. We use this property as an organizing principle to put our flexible computing work in the context of contemporary and relevant systems work. Additionally, we show that this property makes system-level virtualization a powerful and suitable tool to solve some of the challenges brought forward by the impending paradigm shift.

1.7 Organization

The contents of the thesis are organized as follows:

- Chapter 2 reviews previous work related to this thesis. It presents a historical survey of virtualization and compares virtualization to other techniques for encapsulating and transporting execution state. In this chapter we argue the case for virtual machines as the right vehicle for location flexibility. We conclude by showing how the flexibility in manipulating computing state afforded by virtualization is being put to use in the recent systems research projects relevant to both scale and location flexibility.

- Chapter 3 presents Snowbird [107], our virtualization-based location flexibility prototype.

- Chapter 4 describes VMGL [103], a building block of Snowbird that evolved into a widely used software package.

- Chapter 5 motivates SnowFlock [105, 104], our scale flexibility prototype, introduces the techniques behind its realization, and presents a comprehensive evaluation.

- Chapter 6 concludes this thesis with a look towards future directions.

Chapter 2

Related Work

In this chapter we review the abundant literature regarding the subjects to be discussed throughout this thesis. The chapter opens with a section describing virtualization history and techniques. This is followed by a study of the methods employed to perform computation migration at different levels of the software stack, i.e. the OS, the programming language, etc. This study serves to articulate one of the main points in this thesis, namely that VMs are an appropriate level of abstraction to encapsulate (and move) computation because they can be easily serialized, they internally satisfy all application software dependencies, and they demand a very widely available interface for execution, the x86 ABI. Once having established this point, we present an exploration of recent contributions in the area of virtualization-based systems research. The common thread in all these contributions is in exploiting the flexibility VMs provide in terms of manipulating execution state. This is a property that the work presented in this thesis relies on extensively.

2.1 Virtualization

We present definitions and study the evolution of Virtual Machine Monitors. In the first half of this Section we introduce fundamental definitions and chronicle the inception of

Virtual Machine technology in the late 1960s, followed by a discussion of its rebirth as a hot topic for systems research. In the second half, we review the state of the art in the very active field of virtualization, as well as modern variations on the original concept of Virtual Machine Monitors. Finally, we review microkernels and operating system (OS) containers, two techniques that provide kernel virtualization while being substantially different at an architectural level.

2.1.1 A Bit of History on Virtual Machine Monitors

The earliest work on the concept of a Virtual Machine (VM) dates back to as early as 1969 [61]. IBM is credited with the earliest implementations of virtualization, dating back to the M44/44X [166]. The efforts by IBM were best crystalized in their seminal work on the VM/370 system [88, 39]. A thorough review of that early work can be found in Goldberg's 1974 survey [67].

While many definitions of the VM and VMM terms exist, we choose to cite the widely accepted definition put forward by Popek and Goldberg [144]:

> *A virtual machine is taken to be an* efficient isolated duplicate *of the real machine. . . . As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical to the original machine; second, programs running in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.*

One fundamental property established by this definition is *isolation*: communication between two virtual machines co-existing in the same real machine faces the same limitations as it would if the two virtual machines were actual physically disjoint real machines. The definition also highlights three basic properties which we further describe:

The efficiency property : A statistically dominant part of the instructions of an arbitrary program execute directly on the hardware, without VMM intervention.

The resource control property : An arbitrary program cannot affect system resources allocated to it. The VMM must be invoked instead.

The equivalence property : A program running under a VMM environment performs in a manner indistinguishable from an execution without a VMM, barring two exceptions: timing and resource limits.

More informally, a VMM is a software layer that inserts itself between the Operating System (OS) and the hardware. The VMM multiplexes and virtualizes the hardware: it gives multiple OSs the illusion that they are each executing on top of hardware with full privileges and in isolation, and it can hide the actual hardware characteristics, offering instead virtual constructions. This allows commodity software stacks, from the operating systems onwards, to be run without modifications in a sandboxed environment.

VMMs, as exemplified by the IBM 370 case, were initially used for time-sharing facilities. As multi-programming operating systems like UNIX evolved, the use of VMs started to conversely dwindle. During most of the eighties, VMMs remained in the fringes of the sphere of interests of the systems community. Slowly, interest in VMMs regained traction as their use for novel applications became practical. Microsoft's Windows 95 OS contained a light-weight VMM to run legacy DOS and Windows 3.1 applications in MSDOS mode [97]. Research on fault tolerance virtualized the interrupt architecture of a machine so that a hypervisor could shield unmodified operating systems from hardware faults [27].

A very conspicuous incarnation of the virtual machine concept took place in the area of interpreted languages. The Forth language [150], a contemporary of the first system virtualization designs, used a virtual machine runtime to interpret subroutines called "words." Perhaps the most well known user of a language virtual machine is the

Java language [69], with other adopters including interpreted languages like Tcl [134] or Python [158]. A language VM like the Java Virtual Machine (JVM) virtualizes a fictitious hardware abstraction that allows an intermediate representation of the language to be portably executed on different underlying architectures, as well as sandboxed for security purposes.

The rebirth of "hardware" (or system or OS) VMs as a major force in the systems software research arena could be attributed to the Disco project [29] and its successor, Cellular Disco [70]. The main insight provided by Disco is that while hardware evolves continuously, commodity OSs have become stolid due to their size and complexity. Disco thus allows commodity OSs to take advantage of scalar multiprocessor hardware by implementing a VMM that isolates the OSs from the nature of the machine. Disco builds a cluster of virtual machines running on the multiprocessor, and obtains good performance by optimizing common distributed systems interconnect mechanisms (like NFS) through page-sharing tricks implemented by the VMM. It is worth noting that the authors of Disco later became the founders of VMware Inc. VMware has undoubtedly been the main industry player responsible for the growth of virtualization technology.

An important observation is that Disco uses VM technology in an unforeseen context: optimizing the execution of unmodified system software on a novel hardware architecture. In the last decade, the trend of leveraging VM technology for novel applications has remained vigorous. Heralded by Chen and Noble's widely cited position paper [34], applications of VM technology to a wide range of systems problems have surfaced in top systems research venues, covering subjects as diverse as security [48], network code injection [201], mobile computing [100, 165], and software maintenance [163].

2.1.2 Modern Realizations of Virtualization

The popularity of VM technology has also translated to multiple research VMMs, most notably Xen [21]. Xen is an x86 hypervisor architecture: the VMM code is kept to a bare minimum, while control tools and I/O device drivers are offloaded to a privileged administrative VM. A competing approach is that of hosted VMMs: the VMM runs as an extension to an otherwise unvirtualized hosting OS; no software layer interposes between the hosting OS and the hardware, and the VMs implicitly use the drivers of the hosting OS. VMware Workstation [196] and KVM [95] are two examples of hosted VMMs.

Unlike other VMMs, Xen has had a deep impact in the research and industry communities. This is due primarily to its adoption of an open-source license that fostered collaboration from multiple parties and greatly enabled research in the applications of virtualization. Before the advent of Xen, the best open source option for research in virtualization was the User Mode Linux architecture [43]. Another fundamental characteristic of Xen is its use of paravirtualization, a technique that differentiates it from other VMMs along two dimensions: Instruction Set Architecture (ISA) virtualization and I/O device virtualization.

Following Popek and Goldberg's criteria [144], the original Intel x86 architecture cannot be virtualized, at least in a straightforward-manner [156]. This is because the x86 ISA contains sensitive instructions that are not privileged; in other words, instructions that affect or whose behaviour depends on the configuration of system resources do not trap if executed in user mode. This is problematic because in typical operation a VMM will divest the OSs it virtualizes of their kernel mode execution privileges. However, when an OS attempts to configure a system resource (e.g. by manipulating the Global Descriptor Table with the `SGDT` instruction) in user mode, the instruction will fail but no trap will be triggered, thus preventing the VMM from being alerted and executing that instruction on behalf of the virtualized kernel.

The first widespread x86 VMM, VMware, used dynamic binary rewriting to virtualize x86 sensitive instructions on the fly and translate them to traps into the VMM. Xen's paravirtualization approach consists instead of modifying the kernel to replace sensitive but unprivileged instructions with hypercalls: jumps into the hypervisor to execute the virtualized equivalent of the instruction. The difference between dynamic binary rewriting and paravirtualization is akin to interpretation and translation of computer programs: paravirtualization holds the promise of better performance by modifying the kernel only once and producing binaries that do not need further supervision. As we will see shortly, paravirtualization opens the door to other performance enhancements. While paravirtualization conserves the same kernel ABI and consequently needs no modification to user-level code, it does need access to a kernel's source code and cannot support unmodified OSs. This is a serious disadvantage considering the widespread adoption of the closed-source Microsoft's Windows OS.

The dismal virtualization profile of the x86 ISA was recently changed by Intel and AMD's recent iteration on their line of processors, which includes virtualization extensions code-named Vanderpool [90] and Pacifica [3], respectively. With these extensions, Xen became able to support Windows VMs, and other VMMs entirely based on hardware support were introduced, like KVM. While [2] disputes the performance of hardware virtualization, multiple revisions released recently have substantially improved the performance of the virtualization extensions, by incorporating techniques such as nested paging for MMU virtualization, IOMMU for I/O address space protection, and tagged TLBs for more efficient context switching to and from the VMM.

Paravirtualization is also a defining characteristic in the area of I/O virtualization. Typically, full machine virtualization entails external emulation of I/O devices to match the state the drivers residing within the virtualized kernel expect to find. Even for modern VMMs, this arduous task has been only successfully tackled for rather simple hardware devices, such as the PCNet 100 Mbps network card or the Cirrus Logic VGA card. While VMware offer their own proprietary I/O emulation constructions, the QEMU project [23] for full system emulation has come in handy for open-source VMMs using processor virtualization extensions (like KVM or Xen). Unfortunately, emulated I/O performance is orders of magnitude worse than real I/O performance. Paravirtualization allows virtualizing at alternative interfaces that enable far better performance. By intercepting at intermediate stages of the I/O stack of a kernel, paravirtualization allows the virtualization software to service whole Ethernet packets, batches of generic block I/O requests, or SCSI command packets, rather than reverse-engineer the meaning of each single DMA request or MMIO write. For these reasons, I/O paravirtualization seems to be emerging as the *de facto* option for high-performance VMMs.

One crucial feature of modern VMMs is the capability to serialize the state of a VM. At a given point in time, this state is composed of the values of CPU registers, the contents of memory frames, and metadata concerning I/O virtualization. By canonicalizing host-specific variables, like mappings between VM physical frames and actual physical frames (or machine frames, in Xen and Disco parlance), the serialized state is made host-independent. This allows for suspend and resume capabilities: a VM can be paused, serialized into secondary storage, and then resumed at a later point in time in the same or other physical machine. This feature has been aggressively exploited by research projects exploring user mobility [165, 32, 30]. An interesting variant is live-migration [35], in which contents of memory frames are iteratively pushed to a destination host while execution continues. By marking pages as they are modified, successive iterations reduce the differences between source and destination VM images. When the differing state is sufficiently small, the VM is paused, the last set of frames copied, and execution transferred to the destination host. This enables minimal downtime while relocating an executing VM.

2.1.3 Microkernels and Containers

We close this Section by exploring two alternative designs to traditional VMMs: microkernels and container-based virtualization. In some quarters, the debate between microkernels and virtual machine monitors has become quite heated [77, 79, 157]. We will not dwell here on the details of microkernel implementation [1, 109], or the relative merits or disadvantages of microkernel virtualization vs. traditional VMMs. Suffice to say that a microkernel can mimic VMM functionality by adapting a commodity OS kernel to run on top of the microkernel, and have the commodity kernel offer the same ABI to user-level applications [101]. In other words, paravirtualization can be leveraged to adapt OSs to a microkernel setting. A microkernel can thus run multiple OS containers independently of each other, achieving very similar protection guarantees to those offered by a VMM.

Container-based virtualization has been used to provide a class of VMMs in which the isolation property is sacrificed for performance [168]. Containers implement OS multiplexing by interposing at the system call level and separating processes in different subdomains. Virtual machines can thus be realized by allocating different containers to each machine abstraction. Multiple file system views can be supported through `chroot` techniques, and multiple network identities through bridging of virtual network interfaces. Standard QoS and filtering techniques like `iptables`, `ebtables`, traffic control disciplines and token bucket filters overloaded on the standard CPU and disk schedulers complete the picture. The salient point of container-based virtualization is that all containers share the same kernel instance. While this promises better performance, one bug triggered in the kernel brings down the entire container population, thus creating a single point of failure. Prominent container solutions include Solaris zones [147] and Linux VServers [111], currently in heavy use by the PlanetLab platform [139].

2.2 Computation Migration Through the Stack

In this Section we review the different implementations of the computation migration paradigm. We move from higher-level abstractions to lower-level. For each realization, we point out its benefits and shortcomings. In Section 2.2.6, we argue for VM-based migration and show how it aggregates most benefits achieved by previous computation migration projects. We also review the grid as a prime example of a computation migration platform. We conclude by studying related work on migration policies.

2.2.1 Application-Specific Methods

When wishing to relocate application components on-the-fly, developers can always resort to careful application partitioning and custom communication protocols. This is particularly relevant for the domains where bimodal applications are found. Many custom application partitionings have been developed to deal with remote or distributed visualization of scientific or medical phenomena.

A good example of this approach is the Virtual Microscope project [5], which uses heavily optimized and custom-built server and client components coupled with sophisticated caching algorithms to achieve good interactive performance. The Virtual Microscope exploits dataset locality along a number of dimensions to intelligently prefetch data based on the current view. Microscopic slides are offered at different levels of resolution. When the user zooms into a particular region of a microscopic slide, she is likely to request neighbouring regions at the same resolution, or the same region at higher resolutions. Another typical action is changing the focal plane of the current view. Based on these semantic hints, the Virtual Microscope is able to hide latency during slide browsing by continuously predicting the images to be fetched. We point out that the driving design criterion of this system is maximization of the interactive response during visualization.

Benefits : This approach typically achieves maximum performance optimization. The application developer has full semantic knowledge of what should execute where and how to optimally transfer control and data.

Disadvantages : The heavy investment in developer effort cannot be easily adapted to other applications. There is a disregard for legacy code or closed-source applications.

2.2.2 Toolkit-based Migration

The previous approach clearly entails a high level of domain-specific developer effort involved in manually partitioning a bimodal application. In an attempt to achieve some degree of generality and to isolate common subtasks in the process of application partitioning, the scientific community has produced a number of programming toolkits. Again, most efforts have focused on distributed visualization tasks, highly relevant to the domains covered by bimodal applications. Examples include Dv [115], GVU [41], Visapult [25], SciRun [136], and Cactus [68]. We further describe Cactus and Dv as two representative examples.

Cactus [68] is a framework built around a (surprise) arboreal metaphor. There is a core called "the flesh" and pluggable modules called "thorns". Applications are written as a set of thorns that are invoked and coordinated by the flesh. Thorns interact using the flesh API. Utility thorns include IO routines, an http server, and an MPI wrapper for communication between thorns residing in separate hosts. Configuration files are provided in a Cactus Specification Language, describing the structure and inter-relations between a set of thorns. The Cactus Specification Tool parses and checks configuration files for consistency. Upon application startup, the flesh parses command-line arguments, launches the thorns, and schedules their execution. The amount of specialization that code must undergo to fit the Cactus framework is notable.

Dv [115] is a distributed visualization framework built on top of an *active frames* paradigm. Active frames encapsulate code and data and are executed and forwarded by Java user-level processes called frame servers. A set of frames can be spread out across multiple frame servers and serve as a pipeline of data filters. In this fashion, data can undergo multiple transformations before being rendered on the user's desktop.

Benefits : Toolkits achieve high performance optimization, with better generality than pure application-specific partitioning schemes.

Disadvantages : Applications must be written to a particular interface. Developer effort and source code availability are still needed, and application porting is in most cases far from trivial. Seemingly none of the toolkits mentioned above is widely used, implying that their design efficiently accommodates only a particular application sub-domain. Toolkits also constrain application development to the set of supported languages.

2.2.3 Language-based Migration

Language-based code mobility is another well-explored approach to moving computation. The best example of early work in this genre is Emerald [92]. A more recent example is *one.world* [73]. The growth in popularity of Java and its support for *remote method invocation* [140] has made migration of code components via language facilities feasible and relevant to a wide range of computing environments.

The main appeal of language-based migration is the possibility to hide the intricate details of how data and code are accessed behind the shroud of a uniform language abstraction. Emerald realizes this through an object oriented model. Polymorphism allows the developer to remain oblivious to the specific mode of communication used between objects; the underlying implementation can transparently switch from messaging remote objects to sharing memory with local objects. The object abstraction also allows for fine-grained encapsulation of small amounts of data for relocation.

As polymorphism and object encapsulation are salient features of object-oriented programming, most language-based code mobility methods are hosted by an existing or novel OO language. The symbiotic relation between both techniques is evident in the naming, as multiple research code mobility projects are also called “object migration” prototypes.

In the area of object migration we highlight Abacus [11] and Rover [91]. Both systems expose an API that allows developers to structure applications around the concept of relocatable objects. In Abacus, the objects, like Dv’s active frames, compose a pipeline of data filtering stages for cluster data-mining applications. Objects are checkpointed and dynamically migrated between cluster hosts to improve data locality. In Rover, Relocatable Data Objects (RDOs) enable mobile computing-aware application writing. When connectivity to databases or file systems is interrupted, and conflicts are detected during a later reconciliation phase, RDOs are shipped from the client to the server. RDOs express richer semantics, the desired operation(s) rather than the final value(s), and thus can be used to re-enact the queued operations in the server while guaranteeing total ordering and consistency.

One final relevant reference on the area of object migration is Coign [86]. Coign is based on the use of a library construct: Microsoft's Component Object Model (COM). Coign takes a Windows binary and profiles the communication pattern among the multiple COM components it is made of. Through graph-cutting algorithms, Coign can identify subgraphs of objects with high functional correlation and common data needs, and relocate the entire subgraph to maximize performance.

Benefits : Language constructs such as objects provide very fine-grained control on code relocation, with a seamless language interface.

Disadvantages : This approach does not work for legacy applications that were not written in the specified language. Legacy code is very common in scientific applications, with Fortran programs still in active use today. Many object oriented languages and relocation frameworks are supported by interpreters, resulting in a performance overhead that is undesirable for many applications.

2.2.4 Process Migration

Closest in spirit to the work in this thesis is the large body of research on process migration. This is an operating system abstraction that allows a running process to be paused, relocated to another machine, and be continued there. Process migration is completely transparent in the sense that it does not require any changes to applications once they are functional on a migration-enabled platform.

Many prototype implementations of process migration have been built over the last 20 years. Examples include Demos [146], V [183], Accent [207], MOSIX [19], Sprite [46], Charlotte [17], Utopia [210] and Condor [114]. Yet, despite its research popularity, no operating system in widespread use today (proprietary or open source) supports process migration as a standard facility. The reason for this paradox is that process migration is excruciatingly difficult to get right in the details, even though it is conceptually simple. A typical implementation involves so many external interfaces that it is easily rendered incompatible by a modest external change. In other words, process migration is a brittle abstraction: it breaks easily, and long-term maintenance of machines with support for process migration requires excessive effort.

Another important culprit against the adoption of process migration is the problem of residual dependencies [124], or non-transferable kernel state. Condor [114] best exemplifies this problem. Condor reflects all system calls on open files and sockets to the host where the process started execution. This is clearly inefficient for any I/O-bound task. Furthermore, reflecting system calls back to the originating host prevents Condor from supporting fairly useful OS abstractions like signals, shared memory, memory mapped files, and timers (!).

By closely integrating the process migration facility with an underlying distributed file system, many of the residual dependency problems are reduced to distributed file system efficiency problems. This is the case with Sprite [46]. Sprite, however, is still forced to forward many system calls (e.g `gettimeofday`) to the originating host, and in practice remains limited to batch-style processing tasks.

In Charlotte [17], process execution is entirely structured around message channel abstractions called *links*. The links interface is elegantly suited to shield the processes from where their dependencies actually reside – similar to the use of polymorphism in object migration. While this provides for a seamless interface, it still does not solve the performance problem, and sacrifices UNIX compatibility.

Process migration support through message channels is ideally suited to microkernel systems, given the omnipresent IPC abstraction dominating microkernel design. Demos [146] and MOSIX [19] are representative examples of message-based microkernels supporting process migration. Accent, the predecessor of Mach [1], did not follow a microkernel design but distinguished itself by introducing copy-on-write for a process' memory pages, or *lazy copying*. This successful optimization is based on the insight that processes touch a relatively small portion of their address space and it is thus wasteful to transfer the entire virtual memory image up front.

V [183] subsumes many of the aforementioned design points. V is a distributed diskless microkernel: it runs on diskless workstations supported by an external file server. The microkernel IPC abstractions lend themselves to message-channel process structuring, providing a high-degree of transparency in terms of dependency location. This transparency is curiously sacrificed by the need of explicit command-line binding of a task to a location (i.e. "start povray in host X"). V is a harbinger of VM live-migration: it uses *pre-copying* to iteratively prefetch the contents of a process' virtual memory.

By the early nineties, the frustration with the residual dependency problem resulted in implementations that attempted to completely avoid the problem. Utopia [210] is such an example, a load sharing facility that supports relocation of a task only upon startup. This restriction is based on the study by Eager et al. [49] that questions the value of relocating processes during execution for the purpose of load sharing. Utopia's constrained scenario enables the use of a relatively simple user-level harness in charge of resource book-keeping, thus providing compatibility with unmodified applications and OSs. A local-area distributed file system completes the picture.

Benefits : Process migration promises complete transparency.

Disadvantages : This approach suffers is plagued by residual dependencies. It is also dependent on a particular OS and set of libraries, and subject to incompatibilities even at the version level. Typically, fully functional and/or high-performing implementations do not support standard OSs. In general, process migration constrains the internal structure of the application to a single process.

2.2.5 Container Migration

OS containers like Linux VServers [111] and Virtuozzo [178] interpose on all system calls to multiplex the kernel to several process containers. As shown by VServers, this technique is able to provide great performance, although with a lesser degree of isolation than that provided by virtual machines [168]. The Zap system [133] uses the same principle to encapsulate multiple processes and serialize them in a virtual machine-like fashion. Several other projects employ the same techniques [20, 145, 102]. Encapsulation of multiple processes is appealing due to its promise of preserving the semantics of applications made up of several subcomponents.

Contrary to the claims of full OS independence, we perceive a high-degree of correlation of the container abstraction to a particular OS, and also high sensitivity to multiple implementation parameters. For example, Zap expends a substantial effort in virtualizing the `proc` and `dev` Linux filesystems, something that is alien to Windows and widely varying across UNIX implementations. In particular, device virtualization through `dev` is fragile, and exposed to the lack of semantics caused by the unification of most devices under a generic character device interface.

Genericity in UNIX's interfaces results in another important shortcoming for system call virtualization: dealing with `ioctl`s. The `ioctl` system call is a generic interface for component specific messaging between the kernel and user-space. As such, its arguments are pointers to unknown data, and domain-specific knowledge is needed to know how much data to transfer [179, 177]; deep copying may be needed if the structure being pointed to contains pointers itself. Zap and its descendants, while claiming to effectively virtualize `ioctl`s, do not acknowledge this problem.

A third problem of container virtualization is the extraction of kernel state kept on behalf of the application. This is very similar to the residual dependency issue afflicting process migration systems. As shown in the previous Section, either UNIX-compatibility is retained by ignoring the problem (Condor, Utopia), or sacrificed on behalf of a more suitable design (Charlotte, V). Zap claims to achieve the best of both worlds, while providing no hints as to how it extracts from the kernel buffered data from, e.g., open socket connections. The problem is actually acknowledged three years later [102], by reading all data out of a socket after the container has been checkpointed, and reinjecting it before it is resumed. This solution only covers receive-queues, and OS-implementation knowledge is needed for send queues. Furthermore, important TCP semantics (push and urgent flags, exact sequencing, etc) are lost with this method.

Benefits : Better performance than interpreted languages and full VMs. Containers wrap multiple processes and can thus relocate the full extent of applications composed of multiple process and helper scripts.

Disadvantages : Containers are deeply OS-specific, and as such, the virtualization effort can scale to the size of an OS. Otherwise, incompatibilities are exposed for multiple subsystems: ioctls, TCP semantics, etc.

2.2.6 The Case for Virtual Machine-based Migration

We have covered multiple computation migration systems. While all techniques reviewed are exceedingly beneficial in the context of a particular operative configuration, no system has achieved widespread usage. We identify three important problems:

- Transparent legacy support.
- Total OS and language independence.
- Bundling of all computational dependencies within the same container.

We argue that virtual machines (as in hardware VMs supported by VMMs like Xen or VMware) provide the right abstraction for computation relocation. This is not because of an "almost magical" quality of the PC ABI [157], but rather because "there is only one hardware manufacturer out there" [65]. In other words, VMs virtualize at the most universal interface, the x86 hardware interface.

An x86 VM requires no host operating system changes, and builds on the very stable and rarely-changing interface to hardware provided by its virtual machine monitor (VMM). As long as the VMM can be easily deployed, so too can serialized x86 VMs. The myriad interfaces that must be preserved during migration are part of the guest OS, and so the code and state implementing these interfaces is transported with the application. Computation migration enabled through x86 VMs does not require applications to be written for a particular OS or in any particular language. In addition, the internal structure of the application is unconstrained. For example, the application can be a single monolithic process, or it can be a tool chain glued together with multiple scripts and helper libraries.

The stability of interfaces provided by x86 virtualization comes at a price. The VM approach involves a much larger amount of state than any other method of computation migration. This is an acceptable tradeoff due to four reasons. First, network bandwidth, particularly between research institutions, is growing at an increasing pace with the deployment of wide-area gigabit and optic fiber networks. Second, storage is almost obscenely dense and cheap. Third, and fortunately, much of a VM state rarely changes and can therefore be persistently cached at potential destinations, or derived from similar

VMs through content-addressing techniques. Finally, library OSs, initially proposed by the Exokernel project [52] and now being actively deployed in Xen-like environments [12, 184], substantially reduce the amount of supervisor code attached to the application, removing unnecessary general-purpose support and resulting in "lean and mean" VMs.

The universality and uniformity of x86 hardware is not to be taken for granted. While we do not pretend to ignore the existence of MIPS, PowerPC, or other architectures, we do highlight that the deployment of x86 desktops and servers is several orders of magnitude larger. We also acknowledge the incompatibilities caused by vendor extensions to the x86 ISA, such as Intel's SSE and AMD's 3DNow!. While this problem can be solved through emulation or dynamic binary rewriting, we argue here that VMs are, for the most past, unaffected. Most extensions – power management, virtualization itself – do not affect VMs but rather the VMM and host system. For purely performance-oriented extensions like SSE, vendors are constantly upgrading competing extension support: AMD's Athlon 64 chips support Intel's SSE3 extensions since 2004. Ultimately, with the integration of heterogeneous components into the CPU die, like Graphical Processing Units (GPUs) or the Cell co-processing units [74], x86 performance extensions will likely be left unused. While it is unclear how processor heterogeneity will affect virtualization implementations, we have reasons to be optimistic. For instance, general purpose GPU programming, an incipient practice among scientific computing developers [22], is enabled through the use of the OpenGL API which we have conveniently already virtualized with VMGL.

2.2.7 The Grid

The importance of flexible computing, and particularly the need to relocate computing tasks, is acknowledged by the existence of harbingers like the Grid. In its fullest expression, the Grid would allow the matching of computing tasks with the best-suited provider, and the execution of the task at that location. Today, the Grid is implemented through a diverse set of middleware toolkits and standards of considerable density, all made possible by the proliferation of committees. Some examples of grid computing toolkits are Globus [59], Condor in its second iteration [181], and OGSA [87], all used by the scientific computing community today.

While there is considerable variation in the functionality provided by each toolkit, a representative sample includes finding idle machines, authenticating and logging in users, remotely executing an application on a machine, transferring results from one machine to another, checkpointing and restarting applications, and so on. A developer typically constructs a script or wrapper application that uses one of the aforementioned toolkits to chain together a sequence of individual computations and data transfers across a collection of machines.

What we argue for is orthogonal to the capabilities of the grid. Virtualization-based computation migration complements the functionality provided by Grid toolkits, by transforming a single monolithic application into an entity that can be easily migrated under toolkit control. The recent adoption of virtualization by Globus and other projects [57, 94, 174], indicates that the use of virtualization is a trend gaining critical mass. The closest work in the area intersecting grid computing and virtualization is from the Virtuoso project [110], focusing on resource scheduling for interactive VMs running on a single host.

Virtualization-based computation encapsulation is also advocated by other global computing initiatives, due to the efficient support that it provides for resource-control and secure isolation. The XenoServers [99] platform, from which the Xen VMM originated, and PlanetLab [139], are prime examples.

2.2.8 Automated Migration Policies

We close this Section by reviewing work on policies for automating the migration of computational components. Migration automation has focused exclusively on balancing and sharing of resources like CPU, memory and data locality, with no work using the quality of interactive response to trigger migrations.

Following good software practices, policy and implementation have been carefully separated in most computation migration prototypes. This is one of the strong points in Charlotte [17], and the overriding design prototype of Utopia [210]. Utopia performs load sharing by deploying Load Information Manager (LIM) modules on all participating hosts. Load information distribution and consensus decisions are achieved through classical distributed algorithm practices. The modular structure of Utopia is so robust that it persists today in the commercial products offered by Platform Computing Inc. [142].

The most complete effort in process migration load balancing policies is probably the work in MOSIX [19]. MOSIX integrates multiple features:

- information dissemination mechanisms to react to the changes in the load on the execution sites;
- migration cost estimation to decide whether to migrate based on competitive analysis;
- algorithms that prevent instabilities not only per application but for a cluster as a whole (e.g.. flood prevention); and
- consideration for multiple resources such as memory consumption and file system I/O.

Other systems worth noting are Demos [146] and Abacus [11]. Demos uses hysteresis techniques to prevent frivolous and repeated migration between two hosts, when an application is using multiple resources. Because Abacus is concerned with data locality, it creates a data flow graph monitoring the transfer of bytes of data across system objects. Abacus predicts future data consumption by extrapolating the data flow behaviour during the past H time units. The prediction is integrated into a cost/benefit analysis considering as well an estimation of the time it would take for an object to migrate, based on its size. The cost-benefit analysis drives object relocation in the cluster.

2.3 Managing Virtual Machine State

Throughout this thesis we will present methods to efficiently manage the propagation of virtual machine state. One of the salient advantages of virtualization has been its ability to reduce runtime execution state to a simple serialization (or “bag of bits”). As argued in Section 2.2.6, this is made possible by virtualization’s ability to encapsulate the entire software stack and thus self-contain all dependencies. Serialized execution state is very malleable and can be stored, copied, or replicated, much like this thesis will show in the next chapters. We will revisit now recent projects addressing ways to manage VM state.

The Potemkin project [198], implements a honeypot spanning a large IP address range. Honeypot machines are short-lived lightweight VMs cloned from a static template in the same physical machine, with memory copy-on-write techniques.

Live migration [35], previously introduced in Section 2.1, is a classic and very useful way of manipulating VM state through iterative rounds of pre-copying. Remus [40] provides instantaneous failover for mission-critical tasks by keeping an up-to-date replica of a VM in a separate host. Using a modified version of live migration, dirty VM state is continually pushed to the failover site, while a synchronization protocol (similar to Rethink the Sync’s [131]) ensures that no output becomes world-visible until synchronization of an epoch (typically the last 25 to 50 milliseconds) succeeds. Hines et al. [83] address post-copy VM migration, a scheme in which the site of execution of a VM is immediately changed while pages of the VM’s memory are propagated post-facto.

Another relevant project is Denali [201]. Denali dynamically multiplexes VMs that execute user-provided code in a web-server, with a focus on security and isolation. Finally, operation replay can be used to recreate VM state by starting from the same initial image and replaying all non-deterministic inputs to the VM. This technique is used for example for forensics studies of intrusions [48], and can be used to efficiently migrate VM state optimistically [175].

2.3.1 VM Consolidation and the Rise of the Cloud

The seminal work by Waldspurger [199] in the implementation of VMware's ESX server laid the groundwork for VM consolidation. Through sophisticated memory management techniques like *ballooning* and *content-based page sharing*, the ESX server can effectively oversubscribe the actual RAM of a physical host to multiple VMs. Content-based page sharing is a generalization of the notion of content addressable storage, by which similar objects are indentified and indexed via a cryptographic hash of their contents. Content-based indexing of VM state has been employed for security [112] and migration purposes [175]. Further work in the area was presented by the Difference Engine group [75], which claims to provide further memory savings by scanning RAM and synthesizing little used pages as binary diffs of other similar pages.

Content addressable storage is a promising technique for minimizing the amount of state to be transferred. Content-addressing has been heavily used for content sharing [185], and distributed file systems [186]. Content addressability allows cross-indexing of similar content from multiple caches, and thus enables the possibility of minimizing the amount of transferred data to that which is truly "unique" (i.e. no other cache contains it). We believe that the large amount of software sharing between VM images will enable the use of content addressability as a viable means to minimize the amount of disk state replication. For example, in a Xen paravirtual VM, it is extremely likely that the paravirtualized kernel in use will be the unique version distributed by the Xen developers.

VM-based server consolidation has become a *de facto* standard for commodity computing, as demonstrated by the Amazon Elastic Cloud [10], and other infrastructures like Enomalism [53], GoGrid [66] or Mosso [125]. Manipulation of VM state lies at the the heart of most industry standard techniques for managing commodity computing clusters [171], sometimes referred to as clouds. These techniques include suspending to and resuming from disk, live migration [35, 195], and possibly ballooning and memory consolidation. The current deployment models based on these techniques suffer from a relative lack of speed. With VM state ranging in tens of GBs (disk plus memory), it is no surprise that Amazon's EC2 claims to instantiate multiple VMs in "minutes". We will show in Chapter 5 that there is much room for fundamental improvements.

The ability to dynamically move, suspend, and resume VMs allows for aggressive multiplexing of VMs on server farms, and as a consequence several groups have put forward variations on the notion of a "virtual cluster" [51, 161]. The focus of these lines of work has been almost exclusively on resource provisioning and management aspects. Furthermore, implementation is typically performed in a very un-optimized manner, by creating VM images from scratch and booting them, and typically lacks the ability to allow dynamic runtime cluster resizing [58, 209]. Chase et al. [33] present dynamic virtual clusters in which nodes are wiped clean and software is installed from scratch to add them to a long-lived, on-going computation. Nishimura's virtual clusters [132] are statically sized, and created from a single image also built from scratch via software installation.

Further research in aspects of resource managing and scheduling VMs has been focused on scheduling VMs based on their different execution profiles [110] (e.g. batch vs. interactive,) grid-like queueing of VMs for deployment as cluster jobs [209], and assigning resources based on current use [9]. One particular resource of interest is energy management in server farms [129].

The Emulab [202] testbed predates many of the ideas behind virtual clusters. After almost a decade of evolution, Emulab uses virtualization to efficiently support large-scale network emulation experiments [80]. An "experiment" is a set of VMs connected by a virtual network, much like a virtual cluster. Experiments are long-lived and statically sized: the number of nodes does not change during the experiment. Instantiation of all nodes takes tens to hundreds of seconds.

2.3.2 Cloning State

Cloning of an executing task is a fundamental OS technique, dating back to the initial implementations of UNIX [155]. Except during bootstrap, the only way to start processes in UNIX is by cloning the running structures of an already existing process, and adjusting relevant values. A separate `exec` call is necessary to repopulate the memory image of the process with a different executable.

When implementing a cloning facility, lazy-copying of contents, as introduced by Accent [207], is the norm. In other words, pieces of the cloned entity are fetched on demand. One incarnation of this technique is Copy-on-Write (CoW) cloning, in which the cloned entities share all possible resources initially, and only instantiate a private copy of an element before modifying it. Modern UNIX process forking implementations leverage the CoW technique to minimize the amount of memory copying for a clone operation.

CoW techniques are particularly useful in file system environments for performing snapshots, as exemplified by ZFS [172] and LVM [151]. Wide-area VM migration projects such as the Collective [163] and Internet Suspend/Resume [100] have also used lazy copy-on reference for VM disk state. The low frequency and coarse granularity of access to secondary storage allows copying large batches of state over low-bandwidth high-latency links.

Cloning and Copy-on-Write are frequently used techniques in research projects. In the area of virtualization we highlight the Cloneable JVM and the Potemkin Honeyfarm. The Cloneable JVM [93] allows a running Java servlet to effect a full clone of the underlying JVM, with a dual objective: first, to overlay additional MMU-based protection for the running server, on top of the isolation already provided by the JVM; and second, to minimize the startup time for a new JVM by sharing memory pages and “JiT-ed” (Just-in-Time compiled) code from the original JVM. This borrows on the implicit notion shared by many parallel programming paradigms based on cloning: that the new cloned copy is stateful and keeps a memory of all processing performed up to that stage, thus removing the overhead of a cold start.

In Potemkin [198], introduced at the beginning of this Section, lightweight Xen VMs are created when traffic on an unattended IP address is detected. This allows for the flexible instantiation and destruction of honeypot nodes on demand. Honeypot creation is achieved by cloning a base VM image and sharing pages through CoW mechanisms. The honeypot VMs are short-lived, have a very small memory footprint (128 MBs) and do not use secondary storage (the file system is mounted in RAM), thereby greatly simplifying the cloning task.

Chapter 3

Location Flexibility with Snowbird

In this chapter we describe the Snowbird project [107]. An overview of the work related to this part of the thesis is presented in Section 2.2.

3.1 Introduction

A growing number of applications in many domains combine sophisticated algorithms and raw computational power with the deep knowledge, experience and intuition of a human expert. Examples of such applications can be found in simulation and visualization of phenomena in scientific computing, digital animation, computer-aided design in engineering, protein modeling for drug discovery in the pharmaceutical industry, as well as computer-aided diagnosis in medicine. These *bimodal* applications alternate between resource-intensive *crunch phases* that involve little interaction, and *cognitive phases* that are intensely interactive. During the crunch phase, short completion time is the primary performance goal, and computing resources are the critical constraints. During the cognitive phase, crisp interactive response is the primary performance goal, and user attention is the critical constraint.

Optimizing both phases is important for a good user experience, but achieving this end is complicated by the large disparity in the performance goals and bottlenecks of the two phases. Today, developers must manually split a bimodal application into a distributed set of components; various programming frameworks have been proposed to assist with this challenging task [5, 25, 68, 136]. Crunch phase components are executed on compute servers or server farms. Distant data servers can be sometimes used if datasets are too large to cache or mirror locally, or are constrained by organizational or regulatory policies that forbid caching or mirroring. In contrast, cognitive phase components are executed locally where they can take advantage of local graphics acceleration hardware. Unfortunately, this approach requires developers to manage communication and coordination between application components, and to be aware at all times of whether a particular component will be executed locally or remotely. This adds software complexity above and beyond the intrinsic complexity of the application being developed, and hence slows the emergence of new bimodal applications.

In this chapter we introduce Snowbird, a system based on virtual machine (VM) technology that simplifies the development and deployment of bimodal applications. Snowbird masks the complexity of creating a bimodal application by wrapping the application, including all its executables, scripts, configuration files, dynamically linkable libraries, and the OS, into a migratable VM. During execution, Snowbird automatically detects phase transitions in the application and migrates the VM, containing the application and its complex web of dependencies, to the optimal execution site. Snowbird neither requires applications to be written in a specific language, nor to be built using specific libraries. Existing closed-source applications can use Snowbird without recoding, recompilation, or relinking.

Snowbird extends existing VM technology with three mechanisms: an interaction-aware migration manager that triggers automatic migration; support for graphics hardware acceleration; and a peer-to-peer storage subsystem for efficient sharing of persistent VM state over the wide area. Experiments conducted with a number of real-world applications, including commercial closed-source tools such as the Maya 3D graphics

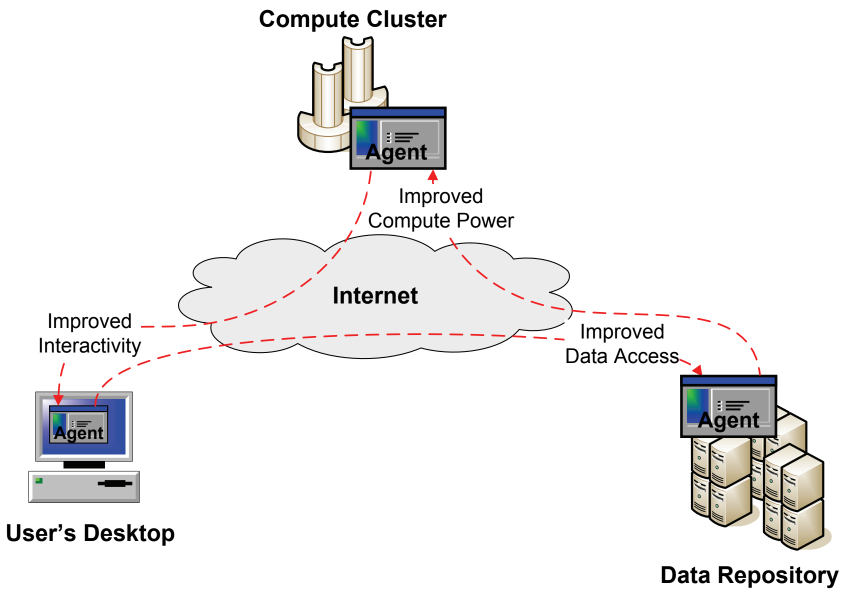


Figure 3.1: Agent Migration Concept

animation package, show that applications running under Snowbird come within 4% of optimal crunch completion times, while exhibiting crisp interactive performance that is comparable to native local execution.

From a more abstract perspective, Snowbird can be viewed as a tool that provides seamless transitions between thick and thin client modes of execution. It has long been known that the strengths of thick and thin clients complement each other. Thin clients are attractive in CPU-intensive and data-intensive situations because application execution can occur on remote compute servers close to large datasets. Unfortunately, high network latency and jitter between the application execution site and the user site can lead to poor interactive performance. Thick clients offer a much better user experience in that situation. By transparently morphing an application between thick and thin client modes of execution, Snowbird gives a user the best of both worlds.

3.2 Design and Implementation

The dominant influence on the design of Snowbird was our desire to simplify the creation and deployment of bimodal applications without imposing onerous constraints on developers. The use of VM technology is the key to achieving this goal. It allows a developer to focus on the creation of a single monolithic entity rather than the more difficult programming task of creating a distributed system. This monolithic entity, called an *agent,* is the migratable embodiment of an application that transparently and seamlessly relocates itself to achieve optimal performance. At run time, Snowbird automatically detects phase transitions and migrates the agent to the optimal execution site. The example illustrated in Figure 3.1 shows an agent that starts at the user's desktop to provide good interactive response during a cognitive phase. It then migrates to several remote sites, where it leverages the superior compute power of a shared-memory multiprocessor and improved I/O performance from proximity to a large dataset. The agent then returns to the desktop for the next cognitive phase.

The logical encapsulation provided by an agent eases the complexity of developing and deploying a bimodal application. Simplicity is reinforced by the fact that all the paraphernalia associated with a large application (such as other processes, dynamically linked libraries, and specific OS features upon which an application relies) is atomically moved with the same containing agent. Hence no application-specific code has to be pre-installed in order to run on a site. An agent only requires SSH access credentials to execute on a Snowbird-enabled site. The SSH credentials are also used to encrypt all communications. Note that an agent can be a tool chain composed of several processes executing simultaneously or sequentially in a pipeline fashion.

Snowbird's use of VM technology offers three significant advantages over existing approaches to code mobility. First, applications do not have to be written in a specific language, be built using specific libraries, or run on a particular OS. Second, legacy applications do not have to be modified, recompiled, or relinked to use Snowbird. This greatly

End Points		RTTs (ms)			
		Min	Mean	Max	c
Berkeley –	Canberra	174.0	174.7	176.0	79.9
Berkeley –	New York	85.0	85.0	85.0	27.4
Berkeley –	Trondheim	197.0	197.0	197.0	55.6
Pittsburgh –	Ottawa	44.0	44.1	62.0	4.3
Pittsburgh –	Hong-Kong	217.0	223.1	393.0	85.9
Pittsburgh –	Dublin	115.0	115.7	116.0	42.0
Pittsburgh –	Seattle	83.0	83.9	84.0	22.9

Table 3.1: Internet2 Round Trip Times

simplifies real-world deployments that use proprietary rather than open-source applications. Third, migration is transparent and seamless to the user, beyond the obviously desirable effects of improved interactive or computational performance.

A second factor influencing Snowbird's design is our desire to support applications at an Internet scale, particularly those using remote datasets over WAN links. It is end-to-end latency, not bandwidth, that is the greater challenge in this context. Table 3.1 shows recent round-trip time (RTT) values for a representative sample of Internet2 sites [130]. The theoretical minimum RTT values imposed by speed-of-light propagation, shown in the last column c, are already problematic. Unfortunately, technologies such as firewalls, deep packet inspection engines, and overlay networks further exacerbate the problem, causing the minimum observed RTT values to far exceed the substantial propagation delays. Although bandwidths will continue to improve over time, RTT is unlikely to improve dramatically. These performance trends align well with the design of Snowbird; the critical resource in VM migration is network bandwidth, while the critical resource for crisp interaction is RTT.

A third factor influencing Snowbird's design is the peer-to-peer (P2P) relationship between migration sites that is implicit in Figure 3.1. Since Snowbird can migrate to any Internet site for which it possesses SSH credentials, there is no notion of clients or servers. Solely for purposes of system administration, Snowbird associates a *home*

Command	Description
Life Cycle Commands	
createagent [-f config] [-i image] <agentname>	Create the agent called <agentname> with localhost as its home. Use default VM settings if <config> is not specified. Populate agent's virtual disk with <image>, if specified. This is a heavyweight operation.
launch <agentname>	Start execution of specified agent at its home. Involves full powerup of the VM, and bootup of guest OS.
kill <agentname>	Kill specified agent at current execution site. The agent will have to be launched again; it cannot be suspended after a kill.
purge <agentname> [hostname]	Permanently delete all traces of agent at <hostname> after synchronizing agent state with home. If no <hostname> is specified, apply to all hosts.
Migration Commands	
suspend <agentname>	Suspend execution of agent at current site.
resume <agentname> [hostname]	Resume execution of agent on specified host. If no host is specified, the agent resumes where it was last suspended.
suspend-resume <agentname> <hostname>	Perform live migration between agent's current machine and hostname.
Maintenance Commands	
addhost <agentname> <hostname>	Make <hostname> a potential destination for the specified agent.
sync <agentname> [hostname]	Ensure that the home of the specified agent has a complete and up-to-date copy of its entire state. Synchronize state at [hostname], if specified.
movehome <agentname> <newhome>	Change the home of the specified agent. The agent cannot be active during this operation.
listhosts <agentname>	Display information about the hosts currently available to an agent.

Arguments between angle brackets (< >) are mandatory, while those between square brackets ([ ]) are optional.

Figure 3.2: Snowbird Commands

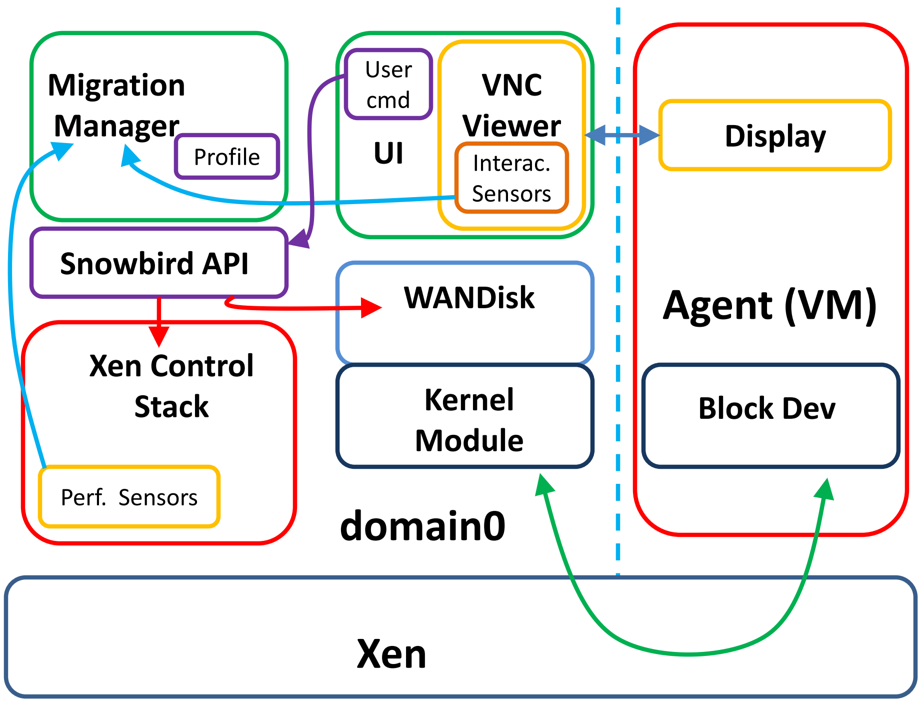


Figure 3.3: Snowbird Architecture for the Xen VMM

host with each agent. This is typically a user's desktop machine or some other nearby computer where the user spends most of her time interacting with the agent. The home host acts as the authoritative machine on which the commands shown in Figure 3.2 are issued. The command line interface for Snowbird includes commands for managing an agent's life cycle, for controlling agent migration, and for system administration. Migration control commands are typically used by the migration manager described in Section 3.2.2. However, they are available for explicit user control, if desired.

Figure 3.3 depicts the different components that make up the Snowbird system. The figure shows the current prototype of the system, which is an instance of Snowbird implemented on top of the Xen VMM. Sections 3.2.1 to 3.2.4 present more details on the different components of the Snowbird architecture depicted in the figure. With the exception of the WANDisk kernel module, all these components are user-space daemons (such as the migration manager), or modifications to existing software running inside a

VM (such as the display implemented via a VNC server). Section 3.2.1 expands upon our use of VM technology. Section 3.2.2 describes the interaction-aware migration manager and the sensors it uses. Section 3.2.3 describes the use of hardware-accelerated graphics by VM applications in the context of Snowbird's graphical user interface. Section 3.2.4 presents Snowbird's wide-area, peer-to-peer storage subsystem for sharing persistent VM state at the block level.

3.2.1 Choice of VM Technology

The current version of Snowbird is based on the Xen 3.0.1 VMM. We chose Xen because its open-source nature makes it attractive for experimentation. However, our design is sufficiently modular that using a different VMM such as VMware Workstation will only require modest changes. Most Snowbird components are daemons that run in the host kernel or domain, and hence no changes to the underlying VMM are needed.

Snowbird uses VM migration [165, 163] to dynamically relocate the agent from a source to a target host. To migrate an agent, its VM is first suspended on the source. The suspended VM image, typically a few hundred MBs of metadata and serialized memory contents, is then transferred to the target, where VM execution is resumed. Snowbird uses *live-migration* [35] to allow a user to continue interacting with the application during agent relocation. This mechanism makes migration appear seamless, by iteratively pre-copying the VM's memory to the target while the VM continues to execute on the source host. When the amount of prefetched VM memory reaches a critical threshold, a brief pause is sufficient to transfer control.

Modern VMMs allow the creation of VMs with multiple virtual CPUs regardless of the underlying number of available physical cores. By configuring their containing VMs as SMPs, agents are able to transparently leverage the increased computing power of potential multiprocessor migration targets.

3.2.2 Interaction-Aware Migration Manager

While users can explicitly control migration decisions using the commands in Figure 3.2, Snowbird provides system-controlled agent relocation as one of its key features. In other words, the decision to migrate, the choice of migration site, and the collection of information upon which to base these decisions can all happen under the covers in a manner that is transparent to the user and to the agent.

A key feature of Snowbird is that it accounts for the quality of the application's interactive response when making its migration decisions. This is in stark contrast to the large body of related work on automated process migration policies [19], which concentrates on computationally-intensive applications devoid of interactions. Snowbird uses an *interaction-aware migration manager* module that bases its decisions on three sources: *interactivity sensors* that extract relevant data from the Snowbird user interface; *performance sensors* that extract their data from the VMM; and *migration profiles* that express the migration policy as transitions of a finite state machine triggered by sensor readings. Snowbird's clean separation between policy and mechanism simplifies the use of different profiles and sensors.

Interactivity sensors

The interaction sensor is built into Snowbird's graphical user interface. As we will described in more detail in Section 3.2.3, Snowbird executes a VNC viewer in the user's desktop exporting the display output of the agent VM. This VNC viewer is augmented to collect a stream of time-stamped events corresponding to keyboard/mouse inputs and screen updates, as shown in Figure 3.4. The intensity of the user's interactive demand and the quality of the agent's response can both be inferred from this stream.

Our measure of *interaction intensity* is the number of input events per unit of time. Our measure of *interactive response quality* is the number of frames per second triggered by an input event. This metric can be derived by assuming that all screen updates are

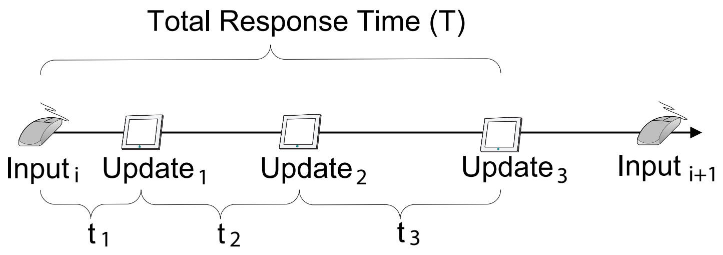


Figure 3.4: Interaction Intensity and Smoothness

causally related to the most recent input event. The frames per second (FPS) triggered by that input event is thus the number of related screen updates divided by the time from the event to the last of those updates. The FPS metric reflects both the smoothness and the swiftness of an interactive response. Remote interaction usually relies on non-work-conserving thin-client algorithms such as VNC [154], that under adverse network conditions skip frames to "catch up" with the output. Skipping frames in this manner results in jerky on-screen tracking of mouse and keyboard inputs that can be annoying and distracting. For work-conserving thin-client algorithms like X, a low FPS rating means that the same amount of screen updates happened in more time, resulting instead in a sluggish response. We thus quantify the quality of the interactive response of an event window as the average FPS yielded by all the inputs in that window. High interaction intensity combined with a low-quality response is the cue used by the migration manager to trigger a remote-to-local transition.

Performance Sensors

Snowbird provides performance sensors for *CPU utilization* and *network utilization*. These sensors periodically poll the VMM for an agent's share of CPU time, and the number of bytes transmitted on its network interfaces, respectively. The poll interval is configurable with a default value of one second.

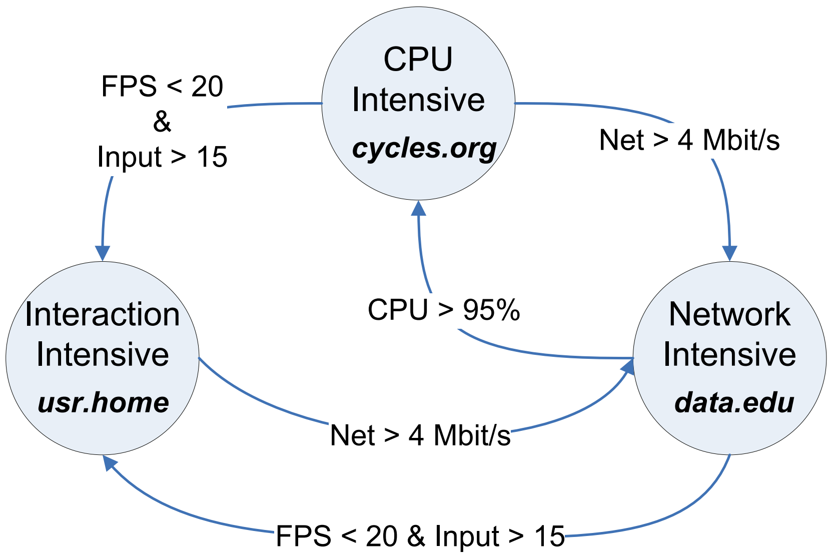


Figure 3.5: Example Partial FSM

Migration Profiles

A migration profile defines a finite state machine (FSM) that is used to model the agent's behaviour. An agent can thus be considered as an entity defined by the VM containing the application, and the profile expressing the application's behavioural patterns. As shown in Figure 3.5, each state in this machine characterizes a particular level of resource demand and/or interaction. Profile rules define when and how sensor readings should trigger state transitions. The profile also specifies the amount of past sensor information that should be averaged to evaluate the rules, which defaults to ten seconds. Each state defines an optimal execution site. The mapping of application profile-defined FSM states to hosts is dependent on the infrastructure available to each user. While the figure presents an examples of a typical FSM derived from the three sensors we implemented, profile writers are free to generate more complex FSMs using more sensors and states.

Profile creation involves characterization of an agent's resource usage and may be performed by application developers or by third-parties such as user groups, administrators, or technically adept users. In the absence of an application-specific profile, the migration manager uses a generic profile that identifies typical crunch and cognitive phases.

The default profile is shown in Figure 3.5, with most of its values being fairly intuitive and conservative. We use the long-established 20 FPS threshold [6] to trigger interactive response-based migrations. The input threshold of 15 inputs per window of ten seconds is derived from observations of the average input event generation rate in our experiments. We were able to use this generic application profile for all the experiments described in Section 3.3.5.

An interesting possibility is to augment the Snowbird migration manager with many of the features developed by the process migration community in the scope of automated migration, such as selecting destination sites based on their current load [19]. One relevant concern in our environment is the handling of applications with overlapping crunch and cognitive phases, which could compromise the agent's stability by "thrashing" between the two states. The straightforward solution we have implemented is to specify a priority favouring interactive performance when conflicting migration rules are simultaneously triggered. Another solution would be to invoke hysteresis mechanisms [146] to prevent the migration manager from adopting this erratic behaviour.

3.2.3 Hardware-Accelerated Graphical Interface

The graphical user interface for an agent has to comply with two requirements. First, a user should be able to interact seamlessly with an agent, regardless of its current location. Second, many of the applications targeted by Snowbird (such as scientific visualization and digital animation), require the use of 3D graphics acceleration hardware, a feature absent from most virtualized execution environments.

To meet these requirements, Snowbird uses VMGL, which is described in depth in Chapter 4 of this thesis. VMGL includes an enhanced thin client interface based on VNC [154], and provides agents with access to 3D graphics hardware acceleration. When the agent is running on a remote host, the thin client protocol is used to communicate screen updates and user inputs (i.e., keystrokes and mouse) over the network. When the

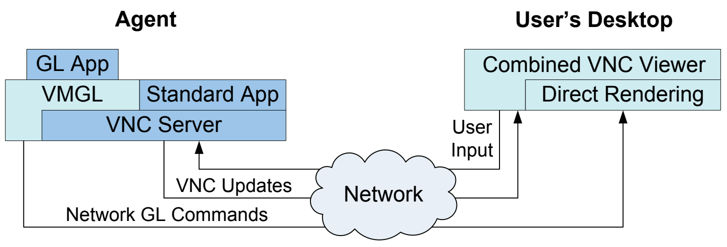


Figure 3.6: 3D Support in Snowbird

agent runs on the user's desktop, the network becomes a loopback connection. Interaction is never interrupted during agent relocation because network connections persist through live-migrations: for relocations within the same L2 subnet, a gratuitous ARP-reply binds the agent's IP address to the new physical host. Relocations across subnets can be supported with VPN tunnels or L2 proxies like VNETs [174].

VMGL provides applications running in a VM access to 3D graphics hardware acceleration by virtualizing the OpenGL API. This cross-platform API for 3D applications is supported by all major graphics hardware vendors. We use library preloading to masquerade as the system's native GL driver and intercept all GL calls made by an application. GL primitives are then forwarded over the network, using the WireGL protocol [28], to a remote rendering module where they are rendered directly by 3D graphics acceleration hardware. Although this setup allows complete flexibility, we expect the rendering module to execute in the user desktop's administrative VM, physically co-located with the agent VM during cognitive phases.

Figure 3.6 shows how we use VMGL within Snowbird. GL primitives bypass the VNC server and are rendered using 3D hardware on the user's desktop. Updates from non-3D APIs (e.g. Xlib) used by standard applications are rendered by the VNC server on its virtual framebuffer and shipped to the viewer. A modified VNC viewer composes both streams and offers a combined image to the user. Input events are handled entirely by the thin client protocol. Similar mechanisms can be used to support the Direct3D rendering API for Windows VMs [197].

3.2.4 The WANDisk Storage System

VM migration mechanisms only transfer memory and processor state; they do not transfer VM disk state, which is typically one to three orders of magnitude larger (many GBs). Therefore, each VM disk operation after migration usually involves network access to the source host. While this is standard practice on the LAN environments that are typical of VM deployments in data centers (SANs, Parallax [121], distributed file systems like Lustre [26]), it is unacceptable for the high-latency WAN environments in which we envision Snowbird being used. A distributed storage mechanism is thus needed to take advantage of read and update locality in disk references. Furthermore, while several WAN-optimized distributed storage choices were available to us, none of them satisfied two key characteristics of the Snowbird deployment model. First, there are multiple symmetric hosts on which an agent might run, thus precluding storage systems that are limited to a single replica support (DRBD [152]), and making systems that centralize data transfers on a single server undesirable due to the inability to directly transfer state between endpoints. Such centralized systems include NFS [162], AFS [36], Coda [164], and VM disk mechanisms used by the Collective [163] and Internet Suspend/Resume [165]. Second, in this P2P-like model there is no need to maintain complex multiple-writer synchronization protocols [128], since the agent executes, and modifies its underlying disk state, in a single host at a time.

We have therefore implemented a distributed storage system called WANDisk, that provides efficient WAN access to multiple replicas of an agent's virtual disk. To provide flexibility in the choice of migration site, WANDisk follows a P2P approach where any Internet host can maintain a persistent replica of the agent's state. To reduce data transfers, WANDisk relies on the persistence of the replicas, which are created on demand as new migration sites are identified. WANDisk's replica control mechanism uses two techniques for optimizing the efficiency of agent migration. First, lazy synchronization is used to avoid unnecessary data transfers to inactive migration sites or for unused

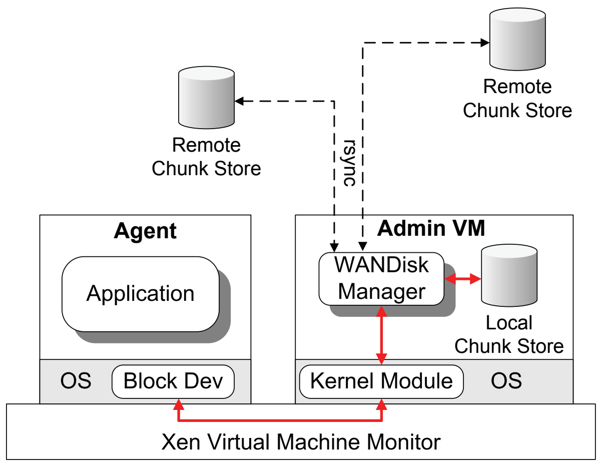


Figure 3.7: WANDisk Storage System Architecture

parts of a virtual disk. Second, differential transfers are used between replicas to reduce synchronization overhead.

Figure 3.7 shows the two-tiered WANDisk architecture, which consists of a *kernel module* and a user-space *disk manager*, both operating within Xen's administrative VM. The kernel module presents a pseudo block device that is mapped to an agent's virtual block device. All agent-originated block requests are handled by the pseudo block device and redirected into the user-space disk manager.

The disk manager partitions the agent's virtual disk into *chunks* and uses a *chunk table* to keep track of versioning and ownership information. Chunk size is configurable at agent creation time; we use a chunk size of 128 KB in our experiments, which has been found to work well in practice [165]. As the agent modifies blocks in its virtual block device, the mapped chunk's version number is incremented, and its ownership transferred to the host where the agent is executing. Each host thus "owns" the chunks which the agent modified while executing there. Before the agent accesses any of those chunks at a different host, the chunk table will point WANDisk to the location of the freshest copy. The chunk table is thus the only piece of metadata necessary for the correct execution of WANDisk, and becomes a crucial addition to an agent's migratable state. To account for this, we have modified live migration in Xen to include the chunk table; however, actual chunk transfers are not involved in the critical path of agent migration. WANDisk fetches chunks exclusively on-demand, using the rsync algorithm [188] to perform efficient differential data transfer.

The heavyweight `sync` command shown in Figure 3.2 is available for bringing any replica up to date under explicit user control. This command may be used for performance or reliability reasons. The command blocks until the replica at the specified migration site is both complete and up to date. At this point, agent execution can continue at that site even if it is disconnected from other replicas.

3.3 Usage Experience and Experimental Evaluation

We have gained hands-on usage experience with Snowbird by applying it to four bimodal applications from distinct application domains. None of these applications were written by us, and none had to be modified for use with Snowbird. Two of the applications (*Maya* and *ADF*) are commercial closed-source products whose success in the marketplace confirms their importance. The other two applications (*QuakeViz* and *Kmenc15*) have open source user communities. Section 3.3.1 describes these applications in more detail.

We found that using these applications with Snowbird was straightforward. Installing each as an agent was no more complex or time-consuming than installing it on a native machine. The only extra step was the creation of an application profile for the migration manager. Our generic application profile proved to be adequate for these four applications, but we recognize that some customization effort may be needed in the case of other applications.

This positive qualitative experience leads to a number of quantitative questions. What performance overheads does Snowbird incur? How much does it improve task completion time in the crunch phase, and crispness of interaction in the cognitive phase? How close is Snowbird's performance to that achievable through optimal partitioning (which is

	Crunch phase (minutes for one processor)	Cognitive phase (minutes)	Parallel mode
Maya	96	29	Process fork
QuakeViz	107	23	Pthreads
ADF	125	26	PVM
Kmenc15	43	15	Process fork

Table 3.2: Application Characteristics

necessarily application-specific)?

The rest of this Section describes our answers to these and related questions. Section 3.3.1 begins by describing the four applications and the benchmarks based on them. Section 3.3.2 then describes our approach to balancing realism and good experimental control in the cognitive phase, even in the face of unpredictable user behaviour. Sections 3.3.3 and 3.3.4 describe our experimental setup. Finally, Section 3.3.5 presents our results.

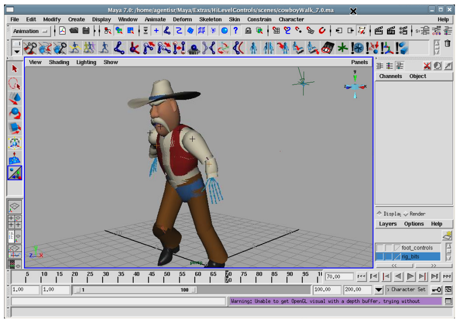
3.3.1 Application Benchmarks

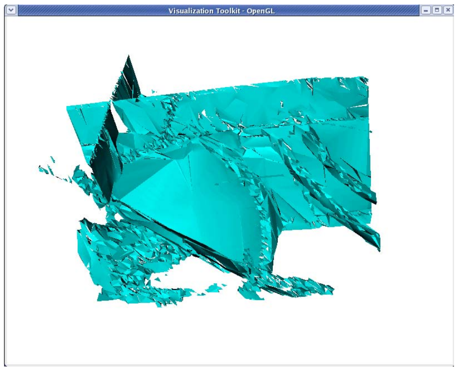
To demonstrate Snowbird's broad applicability, we experimented with applications that are representative of the domains of professional 3D animation, amateur video production, and scientific computing, and include both open source as well as commercial closed source products. For each application, we designed a representative benchmark that consists of a crunch and a cognitive phase. Application characteristics are summarized in Table 3.2, and screenshots can be found in Figure 3.8.

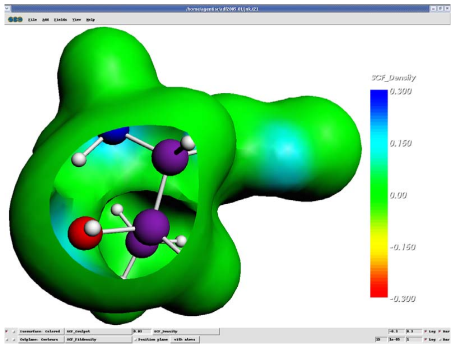
Maya

(Digital Animation, closed source)

Maya is a commercial closed source high-end 3D graphics animation package used for character modeling, animation, digital effects, and production-quality rendering [18] . It is an industry standard employed in several major motion pictures, such as "*Lord of*


(a) Maya: Character modeling


(b) QuakeViz: Ground motion isosurface


(d) ADF: Energy density for an amino acid molecule

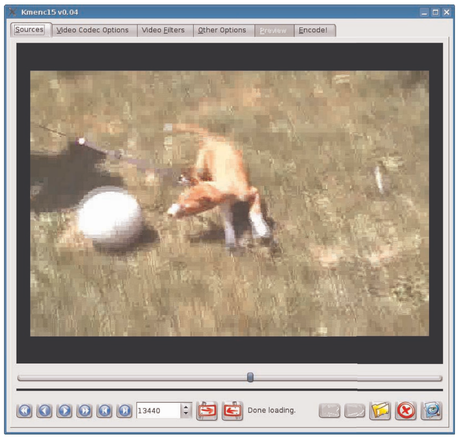

(c) Kmenc15: Video Editing

Figure 3.8: Application Screenshots

the Rings," and "*War of the Worlds*." Our benchmark encompasses the typical work involved in completing an animation project. During the 29-minute cognitive phase, a digital character is loaded, a number of intermediate positions are generated by tweaking the character's skeleton and joints, and the animation pattern is scripted. The user periodically visualizes a low-fidelity preview of the animation, which Maya generates using graphics hardware acceleration. The crunch phase consists of rendering a photo-realistic version of each frame in the animation. This is a highly parallelizable CPU-intensive process that does not use graphics hardware. Maya allows the crunch phase to be initiated on a remote compute server, thus providing a case of application-specific partitioning against which to compare Snowbird.

QuakeViz

(Earth Sciences, open source)

This is an interactive earthquake simulation visualizer, and the only benchmark that accesses a remote dataset. Our benchmark consists of the visualization of a 1.9 GB volumetric dataset depicting 12 seconds of ground motion around a seismic source in the Los Angeles Basin [7]. In our experiments, this dataset is kept on a storage server co-located with the remote compute server and accessed via NFS – resulting in LAN-quality access from the compute server but WAN-quality access from the user desktop. During the crunch phase, QuakeViz mines the dataset to extract ground motion isosurfaces: surfaces inside the volume for which all points are moving in the same direction and at the same speed. The result is a set of triangular meshes depicting isosurfaces at successive time steps. Transformations such as smoothing and normals calculation are applied to these meshes to generate a more visually appealing result. In the cognitive phase, the isosurface meshes are rendered on the screen, and the user studies the seismic reaction by moving forwards or backwards in time and zooming, rotating, or panning the isosurfaces. Our benchmark explores 30 different time-steps during its 23-minute long cognitive phase.

ADF

(Quantum Chemistry, closed source)

This is a commercial closed-source tool, used by scientists and engineers to model and explore properties of molecular structures [180]. In the ADF benchmark, the crunch phase consists of performing a geometry optimization of the threonine amino-acid molecule, using the Self-Consistent Field (SCF) calculation method. ADF distributes this intensive calculation to multiple CPUs using the PVM [173] library, providing a second case of application-specific partitioning against which to compare Snowbird. The SCF calculation generates results that are visualized in a subsequent cognitive phase, such as isosurfaces for the Coulomb potential, occupied electron orbitals, and cut-planes of kinetic energy density and other properties. Analysis of these properties through rotation, zooming, or panning, are examples of the actions performed during the 26 minute-long cognitive phase.

Kmenc15

(Video Editing, open source)

Kmenc15 is an open-source digital editor for amateur video post production [98]. Users can cut and paste portions of video and audio, and apply artistic effects such as blurring or fadeouts. Kmenc15 can process and produce videos in a variety of standard formats. This benchmark does not exploit graphics hardware acceleration. In the 15-minute cognitive phase of our benchmark, we load a 210 MB video of a group picnic and split it into four episodes. We then edit each episode by cropping and re-arranging portions of the recording and adding filters and effects. The crunch phase converts the four edited episodes to the MPEG-4 format. Kmenc15 converts the four episodes in parallel, exploiting available multiprocessing power.

3.3.2 Interactive Session Replay

One of the challenges in evaluating interactive performance is the reliable replay of user sessions. To address this problem, we developed *VNC-Redux*, a tool based on the VNC protocol that records and replays interactive user sessions. During the session record phase, VNC-Redux generates a timestamped trace of all user keyboard and mouse input. In addition, before every mouse button click or release, VNC-Redux also records a snapshot of the screen area around the mouse pointer. During replay, the events in the trace are replayed at the appropriate times. To ensure consistent replay, before replaying mouse button events the screen state is compared against the previously captured screen snapshot: if sufficient discrepancies are detected, the session must be reinitialized and the replay restarted. Screen synchronization succeeds because VNC, like most other thin client protocols, is non work-conserving and can skip intermediate frame updates on slow connections. This results in the client always reaching a stable and similar (albeit not always identical) state for a given input. Therefore, given an identical initial application state, the entire recorded interactive session can be reliably replayed.

Unfortunately, the simple screen synchronization algorithms used by other replay tools [208] do not work well in high-latency environments. These algorithms typically perform a strict per-pixel comparison with a threshold that specifies the maximum number of pixel mismatches allowed. Something as simple as a mouse button release being delayed by a few milliseconds due to network jitter can cause a 3D object's position to be offset by a small amount. This offset causes the algorithm to detect a large number of pixel mismatches, stalling replay.

To address this problem, we developed an algorithm based on Manhattan distances to estimate image "closeness". For two pixels in the RGB colour space, the Manhattan distance is the sum of the absolute differences of the corresponding R, G, and B values. If a pixel's Manhattan distance from the original pixel captured during record is greater than a given distance threshold, it is classified as a pixel mismatch. If the total number of pixel mismatches are greater than a pixel difference threshold, the screenshots being compared are declared to be different. By successfully replaying an interactive trace over emulated 100ms and 66ms links, our experiments indicate that this improved matching algorithm works well over high latency networks.

3.3.3 Experimental Configurations

We investigate four configurations for executing bimodal applications:

- *Local Execution:* The application executes exclusively in an unvirtualized environment on a typical desktop-class machine. During interactive phases, 3D graphics are rendered using locally available hardware acceleration. This represents the best scenario for cognitive phases, but the worst case scenario for crunch phases.

- *Remote Execution:* The application executes exclusively in an unvirtualized environment on an SMP compute server located behind a WAN link and close to external datasets. This represents the best scenario for crunch phases. As the user interacts with the application over a WAN link using a standard VNC thin client, 3D rendering on the remote server is software based, representing the worst case scenario for the cognitive phases.

- *Partitioned:* The application executes in an unvirtualized environment on the desktop-class machine, but is able to ship intensive computation to the remote compute server in an application-specific manner. This execution mode combines the best of remote and local execution, but is fully dependent on application support and requires multiple installations of the application. Only Maya and ADF provide this mode of execution.

- *Snowbird:* Snowbird is used to dynamically switch between local and remote execution modes, independently of application support or the lack of it. Both the user's desktop and remote compute server run the Snowbird infrastructure, as depicted in Figure 3.3: the Xen VMM, WANDisk, the hardware-accelerated agent GUI, and the migration manager. All benchmarks are initiated in an agent running at the user's desktop, with the WANDisk state at all hosts synchronized to a pristine original state. The single generic application profile is used for all of our experiments.

By running the complete benchmark in each of the Remote and Local modes, we obtain two sets of results. First, a measure of what is clearly undesirable: running the crunch phase on an underpowered configuration (Local), and interacting with an application executing behind a WAN link (Remote). By comparing against these results we quantify the benefits of Snowbird in terms of reduced completion time for the crunch phase and improved interactive performance for the cognitive phase.

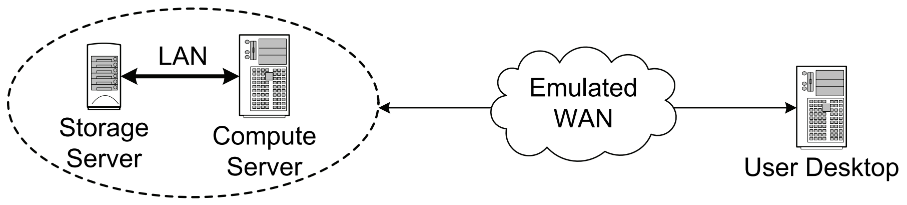


Figure 3.9: Experimental Testbed

Conversely, the execution of the crunch and cognitive phases on the Remote and Local configurations, respectively, represents the ideal application partitioning. This provides an upper bound on the performance of any manual partitioning, as each phase is executed in the most advantageous location and no cost for communication overhead or added computational complexity is included. We compare Snowbird's performance against these sets of results to quantify its overhead.

Finally, we can compare Snowbird to manual application partitioning for those applications that provide that option. While we expect manual application partitioning to be very close to optimal performance for both application phases, we also expect Snowbird to provide similar crunch and interactive performance.

3.3.4 Experimental WAN Testbed

Figure 3.9 shows our experimental testbed, consisting of a user desktop, which is a 3.6 GHz Intel Pentium IV equipped with an ATI Radeon X600 graphics card, and a compute server, which is a dual-threaded dual-core 3.6 GHz Intel Xeon. The desktop and server communicate through a NetEm-emulated WAN link with a bandwidth of 100 Mbps and RTTs of 33, 66, and 100 ms. These RTTs are conservative underestimates of the values observed between US and Europe, as shown in Table 3.1. We use Linux version 2.6.12, with the paravirtualization extensions enabled for the Snowbird experiments. Agent VMs are configured as 512 MB SMP hosts, allowing them to fully utilize the computing power of the compute server's multiple cores. Snowbird uses the WAN-optimized HPN-SSH [149] protocol for data transfers.

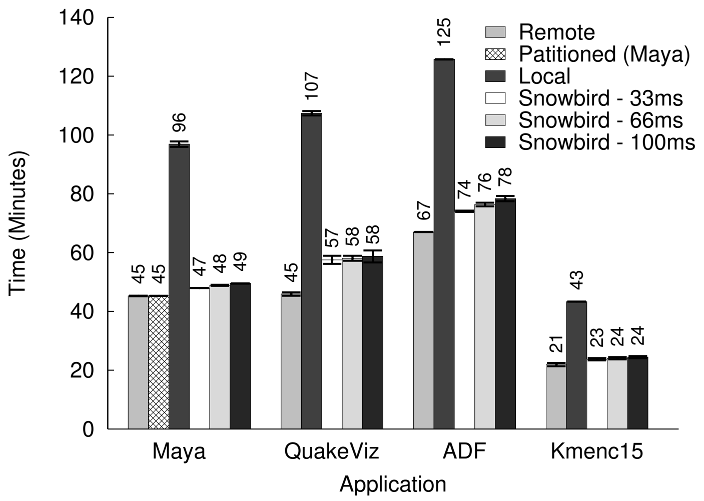


Figure 3.10: Crunch Phase Completion Time

3.3.5 Results

This Section presents the results of our experiments with the four benchmarks introduced in Section 3.3.1. All benchmarks include a cognitive and a crunch phase. In Maya and Kmenc15, the cognitive phase precedes the crunch phase, whereas in QuakeViz and ADF, the cognitive phase follows the crunch phase. Maya and ADF are the only applications we used that provide a partitioned execution mode. Unfortunately, the partitioned execution mode of ADF badly underperformed in our WAN testbed: with a 33 ms RTT, crunch phase completion time expands to roughly six times as much as in thin client mode. The vendor-supplied partitioning is designed for tightly-connected cluster computing and hence uses a very “chatty” synchronization protocol. This is an example of Snowbird overcoming the negative effects of an application-specific partitioning scheme that was not designed for WANs.

Crunch Phase

Figure 3.10 shows the total completion time of the crunch phase for the benchmarks and configurations investigated. Each result is the mean of five trials; error bars show the observed standard deviations. For reasons explained earlier, partitioned execution results are not presented for ADF. As Figure 3.10 shows, Snowbird outperforms local execution by a significant margin. Since the impact of RTT on crunch phase performance is very small, we only show it for Snowbird. The crunch phases of all the benchmarks are CPU intensive and benefit from the increased computational power of the multiprocessor server. QuakeViz also takes advantage of the lower latency and increased bandwidth to its dataset, located on the compute server. More specifically, at 33 ms RTT, Snowbird approximately halves the length of the crunch phase for all applications, and comes within 4 to 28% of the ideal performance of the remote configuration. For Maya, it comes within 4 to 9% of the performance obtained through vendor-supplied partitioning.

Table 3.3 shows how long it takes the migration manager to detect the transition into the crunch phase, how long it takes to migrate the agent to the remote compute server, and the amount of time the application is actually paused during the migration. Each result in this table is the mean of five trials, and the largest standard deviations observed for Detect, Migrate, and Pause are 22%, 4%, and 7% of the corresponding means, respectively. As Table 3.3 shows, the maximum time taken by the migration manager is 14 seconds. Even with the worst-case latency of 100 ms, agent migration never takes more than 70 seconds to complete. In all cases, the agent spends less than 1.5 minutes on the user's desktop after it enters a crunch phase, which amounts to less than 5% of the total benchmark time. The table also shows that the maximum time for which an agent would appear to be unresponsive to user input during migration is six seconds or less. While much higher than downtimes reported for local area VM migration [35], our downtime is due to transferring the last batch of VM memory pages over a WAN link and synchronizing this with the transfer of WANDisk metadata (namely the chunk table). This is an order of magnitude smaller than the best value attainable without live migration (512 MB of VM RAM at 100 Mbps $\simeq$ 41 s).

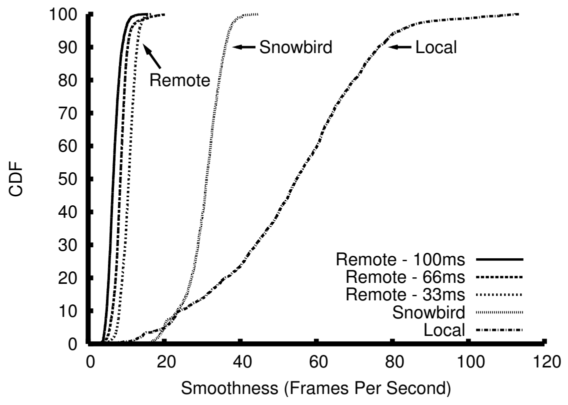
	Time (seconds)								
	Latency = 33 ms			Latency = 66 ms			Latency = 100 ms		
Application	Detect	Migrate	Pause	Detect	Migrate	Pause	Detect	Migrate	Pause
Maya	10.8	51.9	3.5	10.8	53.5	4.7	11.5	58.2	5.6
QuakeViz	8.1	49.9	3.5	8.1	49.9	5.0	8.1	55.6	6.3
ADF	12.5	62.0	4.9	11.5	62.0	6.2	13.1	64.9	6.7
Kmenc15	8.1	51.8	4.7	9.1	54.0	5.7	8.4	59.5	6.7

Table 3.3: Crunch Phase Migration Time

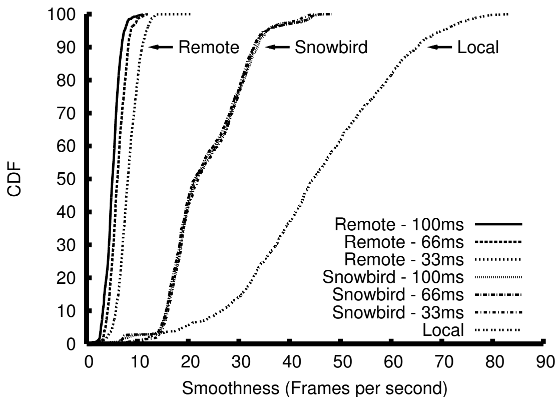
Cognitive Phase

Figure 3.11 shows the Cumulative Distribution Functions (CDFs) of the number of FPS per interaction for each of our four benchmarks under three configurations: local, remote, and Snowbird. Plots to the right indicate better performance than plots to the left. We show results for different network RTTs for the remote and Snowbird configurations. The cognitive phases for QuakeViz and ADF start on the remote compute server soon after the crunch phase terminates. The migration manager detects this transition and migrates back to the user's desktop. In contrast, the cognitive phase of Maya and Kmenc15 start with the agent already running on the user's desktop. We do not include results for Maya and ADF's partitioned mode, as they are practically identical to local interaction.

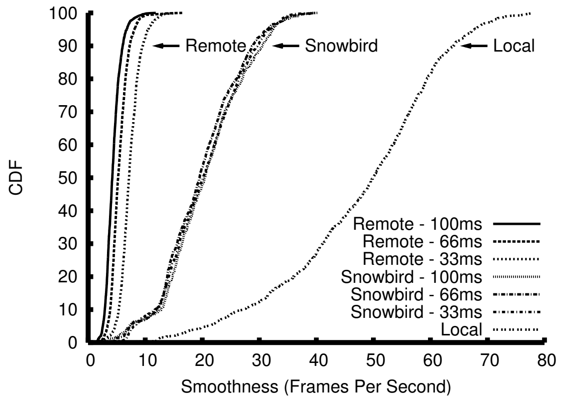
Our results show that Snowbird delivers a much better cognitive performance than remote interaction. More importantly, the median number of FPS delivered by Snowbird is above Airey's previously mentioned 20 FPS threshold needed for crisp interactivity [6]. In general, Snowbird's quantitative interactive performance is between 2.7 to 4.8 times better than that delivered by a thin client, with the interactive response in thin client mode rarely exceeding 10 FPS. Even though the agent has to migrate from the compute server to the user's desktop, Snowbird's cognitive performance tends to be independent of the WAN latency. Further, the network latency has a negligible impact on both the time taken before the decision to migrate is made and the time required to migrate the agent; we omit the migration time results for cognitive phases as they are very similar to those in Table 3.3.



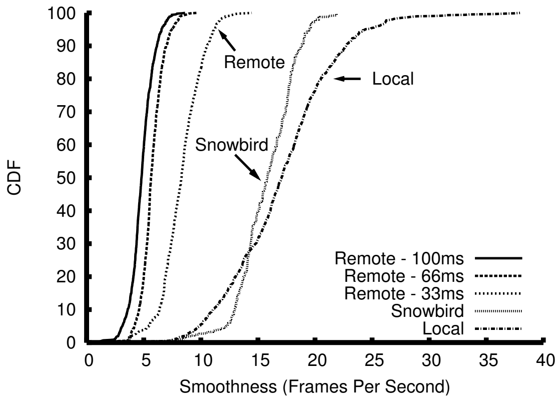
(a) Maya



(b) Quake



(c) ADF



(d) Kmenc15

Figure 3.11: Interactive Response

The results also show that the FPS delivered by Snowbird is not as high as in unvirtualized local interaction. Local execution experiments delivered anywhere between 1.1 to 2.6 times more FPS in the median case. This is essentially an artifact of the early implementation of VMGL used for these experiments – particularly since over 90% of the cognitive phase executes on the local side, in general. As we will show in Section 4.4, a streamlined version of VMGL is capable of delivering between 87% to 100% of the FPS achievable in an unvirtualized configuration, for applications such as 3D games that are far more demanding of the graphics processor. Although purely subjective, we note that despite the early and under-performing version of VMGL used here, once the agent migrates to the local host the user experience delivered by Snowbird is indistinguishable from that of the native configuration for all of the benchmarks.

Summary

Our results confirm that Snowbird offers significant benefits for bimodal applications. Without any application modifications, relinking, or binary rewriting, bimodal applications are able to improve crunch performance through remote infrastructure. This improvement is achieved without compromising cognitive performance. Even when an application vendor supplies a partitioned mode of operation for cluster computing, Snowbird is able to offer comparable performance (within 4%), and in one case greatly exceed its benefit over WANs.

3.4 Implementation Limitations and Future Extensions

The Snowbird prototype has certain limitations in functionality and performance. Some of these limitations arise from our goal of rapidly creating an experimental prototype rather than a robust, complete and efficient product. Other limitations have deeper roots in the design of Snowbird, and will therefore require more effort to overcome.

One limitation is that parallel Snowbird applications execute in a single SMP virtual machine. While the current trend of aggressively scaling processors to a hundred or more cores favours our design, some applications might be inherently designed to use multiple machines in a large cluster. Exploring possibilities to allow VMs "arriving" at a cluster to leverage multiple networked physical machines is one of the seed ideas that eventually evolved into the SnowFlock project we present in Chapter 5.

A second limitation arises from our use of system-level virtualization. At startup, an application might configure itself to take advantage of vendor-specific extensions to the x86 instruction set architecture, such as AMD's 3DNow! or Intel's SSE. Upon migration to different hardware, the application will crash when it attempts to execute an unsup-

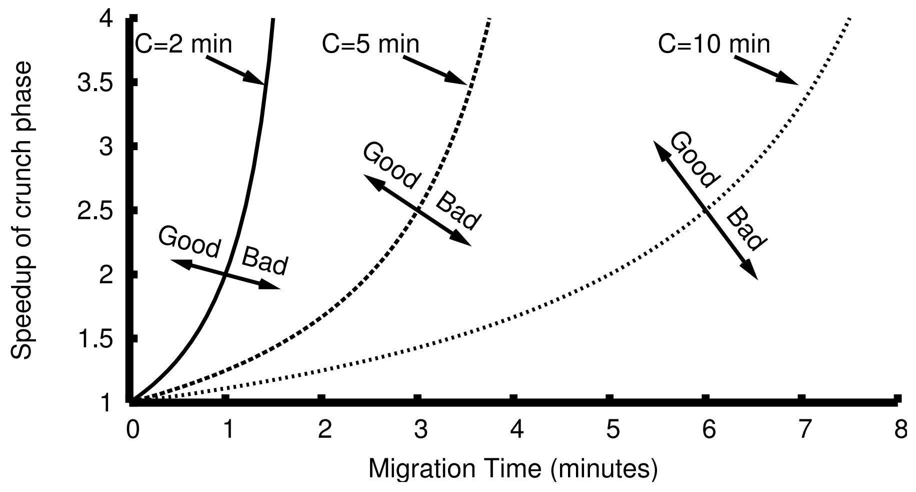


Figure 3.12: When Is Snowbird Useful?

ported instruction. One possible solution is to use dynamic binary rewriting. Another approach is to extend Snowbird so that it never attempts migration to an incompatible destination. These approaches are in line with current standard practice to tackle this problem [4].

A third limitation is that Snowbird assumes a distinct separation of crunch and cognitive phases. Applications that consistently overlap these phases will not benefit from Snowbird. More generally, Snowbird is beneficial only when its agility of adaptation exceeds the rate of change of application behaviour, and when remote execution provides sufficient improvement to overcome the cost of migration. Figure 3.12 illustrates this tradeoff for crunch phases. The horizontal axis shows migration time in minutes, which depends on the quality of the Snowbird implementation, the network conditions and the amount of VM state to transfer. For the current prototype that state is limited to the memory image plus a subset of the virtual disk fetched on-demand by WANDisk. This measure of system agility includes both the swiftness with which migration can be triggered, and the efficiency with which it can be completed. The vertical axis shows the crunch speedup when executing remotely, which depends on the application and the available remote resources. Each curve plots the relation $speedup = C/(C - migration\_time)$ for three hypothetical applications, where C is the crunch phase completion time when executing locally. Above each curve, Snowbird is beneficial; the expected performance gain exceeds the migration cost. Below each curve, Snowbird becomes harmful, as its migration cost eclipses any potential performance gains.

The simple model shown in Figure 3.12 illustrates how improving migration time broadens the set of applications for which Snowbird is applicable. For a given speedup, workloads with smaller crunch time benefit as migration time decreases. And for a given crunch time, swifter migration reduces the constraints on the quality of the remote resources needed. Conversely, high migration times limit the applicability of Snowbird to applications with long crunch phases, or to remote platforms capable of yielding very high speedups. In the current prototype, detection and change of modality occur in roughly 10 seconds, while the migration that follows typically takes about 60 seconds plus lazy WANDisk chunk fetches. Mapping these values to Figure 3.12 to extract a conservative estimate indicates that crunch phases below ten minutes and speedups below a factor 2 will probably not show benefits with the current prototype. We note that VM migration latency over the wide area could be greatly improved with some of the techniques used for efficient VM state replication in SnowFlock, explained in Section 5.4

It should be noted that a complementary attribute of agility is *stability,* which characterizes the ability of the implementation to avoid frivolous migrations that may lead to thrashing. It is well known from control theory that agility and stability are two sides of the same coin, and have to be considered together in the design of an adaptive system. Improvements to Snowbird's agility may necessitate more sophisticated mechanisms for stability.

Chapter 4

Virtualized Graphics Acceleration with VMGL

This chapter covers in more detail the VMGL [103] subsystem of Snowbird. While originating as a means to provide crisp interactive performance to virtualized bimodal applications, VMGL grew into an ambitious project with the goals of providing graphics hardware acceleration to any OpenGL application, regardless of virtual machine monitor, guest operating system, or graphics hardware model.

4.1 Rationale

Virtual machine monitor (VMM) technology has been put to many innovative uses, including mobile computing [30, 100, 165], system management [32, 163], intrusion detection [48], and grid computing [57]. However, the difficulty of virtualizing graphical processing units (GPUs) has limited the use of virtual machines (VMs) for running interactive applications. The performance acceleration provided by GPUs is critical to high-quality visualization in many applications, such as computer games, movie production software, computer-aided design tools for engineering and architecture, computer-aided medical diagnosis, and scientific applications such as protein modeling for drug synthesis.

Software rendering is the prevalent option for virtualized execution, but for this class of applications it is unacceptably slow.

Virtualizing GPUs is difficult for a number of reasons. First, the hardware interface to a GPU is proprietary, and many technical details are closely held as trade secrets. Hence, it is often difficult to obtain the technical specifications necessary to virtualize a GPU. Second, because the hardware interface is not public, GPU vendors make significant changes in the interface as their product lines evolve. Trying to virtualize across such a wide range of interfaces can result in a weak lowest common denominator. Third, the software needed to integrate a GPU into an operating system is typically included with the hardware as a closed-source device driver. In a virtualized environment, the driver is unusable for other guest operating systems. For reasons mentioned earlier, the technical details necessary to create a new driver for other guests are typically not available. In summary, virtualization of a hardware component presumes the existence of a standard interface such as the x86 instruction set, or the IDE and SCSI interfaces to disks; GPUs lack such a standard.

We propose a solution that is strongly influenced by how applications actually use GPUs. Many of the virtualization challenges discussed in the previous paragraph would also complicate the authoring of applications. For example, the absence of a stable GPU interface would require frequent application changes to track hardware. Additionally, the large diversity of technical specifications across GPUs and the difficulty of obtaining them publicly would severely restrict the market size of a specific application implementation. The graphics community avoids these problems through the use of higher-level APIs that abstract away the specifics of GPUs. Practically all applications that use GPUs today are written using one or both of two major APIs: *OpenGL* [96] and *Direct3D* [123]. Of these two, OpenGL is the only cross-platform API supported on all major operating systems. For each supported operating system, a GPU vendor distributes a closed-source driver and OpenGL library. The job of tracking frequent interface changes to GPUs is thus delegated to the GPU vendors, who are best positioned to perform this task. Although OpenGL is a software interface, it has become a *de facto* GPU interface. We therefore make it the virtualization interface.

VMGL is a virtualized OpenGL implementation that offers hardware accelerated rendering capabilities to applications running inside a VM. VMGL runs the vendor-supplied GPU driver and OpenGL library in the VMM host: the administrative VM for a hypervisor like Xen, or the hosting OS for a VMM like VMware Workstation. The host runs a vendor-supported operating system and has direct access to the GPU. Using a GL network transport, VMGL exports the OpenGL library in the host to applications running in other VMs. When those applications issue OpenGL commands, the commands are transported to the GPU-enabled host and executed there. VMGL thus preserves complete application transparency, while no source code modification or binary rewriting is necessary. VMGL also supports suspend and resume of VMs running graphics accelerated applications, thus supporting several novel applications of VM technology [30, 100, 163]. Further, VMGL allows suspended VMs to be migrated to hosts with different underlying GPU hardware. VMGL is not critically dependent on a specific VMM or guest operating system, and is easily ported across them.

We evaluate VMGL for diverse VMMs and guests. The VMMs include Xen on VT and non-VT hardware, and VMware Workstation. The guests include Linux with and without paravirtualization, FreeBSD and Solaris. In experiments with four graphics-intensive applications, including one that is closed source, the observed graphics performance of VMGL comes within 14% or better of native performance, and outperforms software rendering by two orders of magnitude. Although this approach incurs the performance overhead of cross-VM communication, our experimental evaluation demonstrates that this overhead is modest. Moreover, our results also show that multi-core hardware, which is increasingly common, can help in reducing the performance overhead.

As we look toward the future, one important application of VMGL could be found in providing access to the computing power of GPUs to an emerging class of GPU-based scientific applications [22, 62]. The highly parallel and efficient architecture of GPUs has proved tremendously useful in producing high-performance solutions to several scientific problems. Algorithms that solve these problems using GPU processing are written mainly in OpenGL [71]. Hence, we believe that scientific applications running in virtualized environments, like those proposed for the Grid [57], will be able to leverage VMGL for improved performance.

The remaining Sections in this chapter describe the design, implementation and experimental validation of our VMGL prototype for OpenGL virtualization on X11-based systems. We begin in Section 4.2 with an overview of GPUs, OpenGL, and existing techniques to virtualize graphics output. Section 4.3 then describes the detailed design and implementation of VMGL. Section 4.4 presents the experimental validation of VMGL. Finally, Section 4.5 comments on the body of work on hardware-accelerated graphics virtualization that came after VMGL, before concluding this Chapter in Section 4.6.

4.2 Background

In this Section, we provide an introduction to graphics hardware acceleration and OpenGL, the most commonly used cross-platform 3D API. We also describe how X11-based applications leverage hardware acceleration capabilities. We then describe related work addressing virtualization of graphic output. Since VMGL predates most work aimed at virtualizing graphics hardware acceleration, we focus on projects that provide graphics API interposition and manipulation.

4.2.1 Hardware Acceleration

Almost all modern computers today include a Graphics Processing Unit (GPU), a dedicated processor used for graphics rendering. GPUs have become increasingly popular as general purpose CPUs have been unable to keep up with the demands of the mathematically intensive algorithms used for transforming on-screen 3D objects, or applying visual effects such as shading, textures, and lighting. GPUs are composed of a large number of graphics pipelines (16–112 for modern GPUs) operating in parallel. For floating point operations, GPUs can deliver an order of magnitude better performance than modern x86 CPUs [62].

Modern GPUs range from dedicated graphics cards to integrated chipsets. As individual hardware implementations might provide different functionality, 3D graphics APIs have arisen to isolate the programmer from the hardware. The most popular APIs are OpenGL, an open and cross-platform specification, and Direct3D, a closed specification from Microsoft specific to their Windows platform. We describe OpenGL below in more detail.

4.2.2 OpenGL Primer

OpenGL is a standard specification that defines a platform-independent API for 3D graphics. The OpenGL API supports application portability by isolating developers from having to program for different hardware implementations.

Vendors implement the OpenGL API in the form of a dynamically loadable library that can exploit the acceleration features of their graphics hardware. All OpenGL implementations must provide the full functionality specified by the standard. If hardware support is unavailable for certain functions, it must be implemented in software. This isolates the programmer from having to determine available features at runtime. However, the OpenGL specification does allow for vendor-specific extensions; applications can only determine the availability of these extensions at runtime.

The OpenGL calls issued by an application modify the *OpenGL state machine*, a graphics pipeline that converts drawing primitives such as points, lines, and polygons into pixels. An *OpenGL context* encapsulates the current state of the OpenGL state machine. While an application may have multiple OpenGL contexts, only one context may be rendered on a window at a given time. OpenGL is strictly a rendering API and does not contain support for user input or windowing commands. To allow OpenGL contexts to interact with the window manager, applications use *glue* layers such as GLX for X11-based systems, WGL for Microsoft Windows, and AGL for the Macintosh.

Today, OpenGL is the only pervasive cross-platform API for 3D applications. The competing proprietary API, Microsoft's Direct3D, only supports the Windows operating systems. OpenGL implementations are available for Linux, Windows, Unix-based systems, and even embedded systems. Bindings exist for a large number of programming languages including C, C++, C#, Java, Perl, and Python.

4.2.3 X11 Hardware Acceleration

GLX, the OpenGL extension to the X Window System, provides an API that allows X11-based applications to send OpenGL commands to the X server. Depending on hardware availability, these commands will either be sent to a hardware-accelerated GPU or rendered in software using the Mesa OpenGL implementation [120]. As GLX serializes OpenGL commands over the X11 wire protocol, it is able to support both local and remote clients. Remote clients can only perform non-accelerated rendering.

Using GLX can lead to a significant overhead as all data has to be routed through the X server. In response, the Direct Rendering Infrastructure (DRI) was created to allow for safe direct access to the GPU from an application's address space, while still relying on Mesa for a software fallback. The Direct Rendering Manager (DRM), a kernel module, controls the GPU hardware resources and mediates concurrent access by different applications (including the X server). While the DRI provides a direct path for OpenGL commands, GLX must still be used for interactions with the X window server.

4.2.4 Graphics Virtualization

The most straightforward solution to provide 3D acceleration to a VM is to grant it direct access to the GPU hardware. This scheme provides excellent performance to a single VM, but impedes sharing and multiplexing of the device. One interesting variant of this architecture is to designate a *driver VM* [108] as owner of the GPU. The driver VM would thus occupy the same role as the host in a classical VMM architecture, with the advantage of isolating potential driver malfunctions in a separate protection domain. However, in the absence of an IOMMU [24], granting hardware access rights to a VM will weaken the safety and isolation properties of VM technology. A rogue VM with direct hardware access would be able to initiate DMA to and from memory owned by other VMs running on the same machine. Further, a VM with direct hardware access cannot be safely suspended or migrated to a different machine without driver support.

Aside from Blink [78], and preliminary attempts by VMware, VMGL predates most work virtualizing graphics hardware acceleration. We will study Blink alongside relevant projects that arose after VMGL in Section 4.5.

VMGL is based on notions familiar to the area of thin clients and graphics API interposition and manipulation over the network. One relevant form of OpenGL API interposition is AIGLX, which stands for Accelerated Indirect GLX [56], and has been developed to provide accelerated GLX rendering for remote clients. While originally designed to enable OpenGL-accelerated compositing window managers, it could be used as an alternative transport for VMGL. Since AIGLX lacks the transport optimizations used by WireGL, we believe it would severely constrain applicability with its greater bandwidth utilization.

A number of projects for remote visualization of scientific data have tried to optimize remote OpenGL rendering. Some, like Visapult [25], Cactus [68], and SciRun [136], require their applications to be written to a particular interface and are therefore useful only when application source code is available. Other systems [170, 193] render data using remote GPUs and ship the resulting images using slow or lossy thin client protocols such as X11 or VNC.

In summary, VMGL is fundamentally different from traditional thin client protocols (VNC, RDP, ICA) which rely on efficient encoding of 2D pixel maps scraped from the screen. By shipping operations, VMGL is much better suited to supporting rich 3D applications generating dozens of highly detailed frames per second based on spatial transformations of textures stored in memory.

4.3 Design and Implementation

VMGL offers hardware accelerated rendering capabilities to applications running inside a VM. VMGL virtualizes the OpenGL API v1.5, providing access to most modern features exposed by 3D graphics hardware, including vertex and pixel shaders. VMGL also provides VM suspend and resume capabilities. The current VMGL implementation supports Xen and VMware VMMs, ATI, Nvidia, and Intel GPUs, and X11-based guest operating systems like Linux, FreeBSD, and OpenSolaris. VMGL is implemented in userspace to maintain VMM and guest OS agnosticism, and its design is organized around two main architectural features:

- **Virtualizing the OpenGL API** removes any need for application modifications or relinking, guarantees portability to different guest operating systems, and guarantees compatibility with graphics hardware from different manufacturers.

- **Use of a Network Transport** guarantees applicability across VMMs and even for different types of VMs supported by the same VMM.

VMGL is publicly available open-source software [106]. For the remainder of this discussion, the term *host* refers to the administrative VM in Xen, or the underlying OS for hosted VMMs like VMware Workstation. The term *guest* refers to a VM or domain.

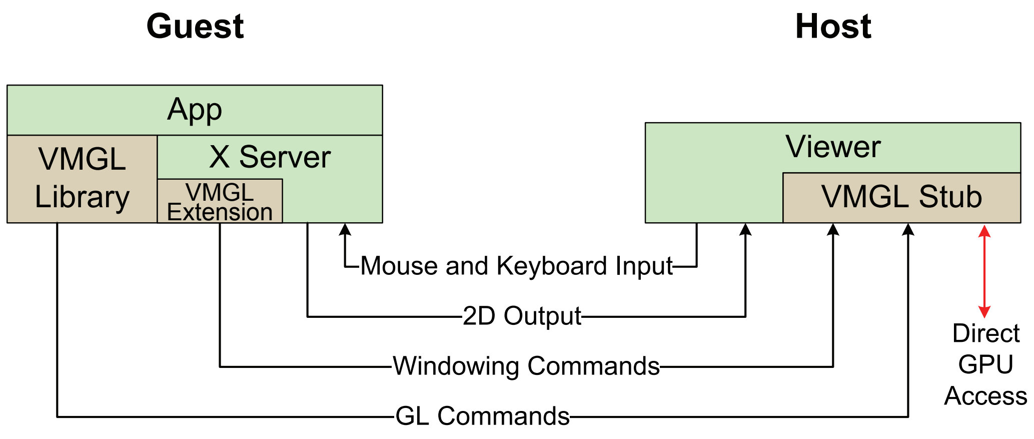


Figure 4.1: VMGL Architecture

4.3.1 Architecture

Figure 4.1 shows the VMGL architecture, which consists of three user-space modules: the VMGL library, the VMGL stub, and the VMGL X server extension. The figure also shows an application running on the guest VM and a viewer which runs on the host and handles user input and the guest's visual output.

Applications running inside guests use the VMGL library as a replacement for standard or vendor-specific OpenGL implementations. Upon application startup, the VMGL library creates a VMGL stub on the host to act as a sink for OpenGL commands. The VMGL stub links against the OpenGL library available on the host to obtain direct rendering capabilities on behalf of the virtualized application. When the application inside the guest issues GL commands, the VMGL library forwards those commands to the VMGL stub using a network transport over a loopback connection. Each application uses a different VMGL stub, and each stub executes as a separate process in the host, thus leveraging the address space protection guarantees offered by the vendor OpenGL implementation.

4.3.2 OpenGL Transport

Figure 4.1 also illustrates the use of viewer software, typically based on VNC [154], that displays the 2D output generated by a guest, and captures user input and relays it to the guest. The guest 2D output is generated by an X server drawing to a virtual 2D framebuffer. In the absence of a virtual framebuffer, 2D output is generated by a VNC X server. We modified a VNC viewer to interact with VMGL stubs; the VNC viewer modifications are minimal as we offload most of the functionality onto the stubs. This allowed us to easily add support for the alternative viewers used by Xen and other VMMs like QEMU [23].

To compose the viewer 2D output and the VMGL stub's 3D GL output, we augment the guest's X server with an extension. The VMGL library uses this extension to register windows bound to OpenGL contexts. The extension monitors changes to the size, position and visibility of OpenGL-enabled windows, and forwards those changes to the VMGL stub. The stub applies this information on its GL graphical output by clipping it to remove sectors that are not currently visible in the guest's desktop, resizing it, and finally superimposing it on the viewer window at the appropriate relative coordinates. The VMGL X extension is a loadable module that can be added to an existing X server configuration. The extension operates at the common device independent layer shared by all variants of X servers, ensuring support for the X11 and VNC servers used inside guests.

The rest of this Section first describes WireGL, the OpenGL network transport used by VMGL. We then describe VMGL's suspend and resume implementation, and discuss driver changes necessary to obtain direct rendering capabilities in a Xen configuration. Finally, we discuss limitations of the current implementation.

The standard OpenGL transport for remote rendering is GLX, previously described in Section 4.2.3. When used over network connections, GLX has two important disadvantages. First, it cannot provide a direct rendering path from the application to the graphics card. This is solved in VMGL by interposing a GL stub that channels GL commands into a direct rendering context. Second, GLX involves costly network round-trips for each and every OpenGL command being invoked. VMGL avoids this cost by leveraging the WireGL protocol [28, 85]. WireGL optimizes the forwarding of OpenGL commands by only transmitting changes to screen-visible state, and by aggregating multiple OpenGL commands in a single network transmission.

WireGL applies the changes to OpenGL state requested by the application to a local cache. Dirty cache contents are flushed lazily as needed. This enables smart discarding or postponing of ineffectual state changing commands. For example, if *glTexSubImage* is used to modify a texture that is currently not visible, no network packets will be sent until the modified area becomes visible.

WireGL further optimizes network utilization by reordering and buffering commands until a commit point arrives. Geometry commands are buffered in queues. Whenever possible, commands are merged in these queues. For example, consecutive *glRotate* and *glTranslate* calls are collapsed into a single matrix modification command. When the application issues state changing or flushing commands (like *glFlush* or *glXSwapBuffers*), the buffered block of geometry modifications is sent, along with outstanding state changes associated to that geometry.

4.3.3 Suspend and Resume Functionality

VMGL provides support for VM suspend and resume, enabling user sessions to be interrupted or moved between computers [30, 100, 163]. Upon VM resume, VMGL presents the same graphic state that the user observed before suspending while retaining hardware acceleration capabilities.

VMGL uses a *shadow driver* [176] approach to support guest suspend and resume. While the guest is running, VMGL snoops on the GL commands it forwards to keep track of the entire OpenGL state of an application. Upon resume, VMGL instantiates a new stub on the host, and the stub is initialized by synchronizing it with the application OpenGL state stored by VMGL. While the upper bound on the size of the OpenGL state kept by VMGL is in principle determined by the GPU RAM size, our experiments in Section 4.4.6 demonstrate that it is much smaller in practice.

VMGL keeps state for all the OpenGL contexts managed by the application, and all the windows currently bound to those contexts. For each window we track the visual properties and the bindings to the VMGL X extension. For each context, we store state belonging to three categories:

- **Global Context State:** Including the current matrix stack, clip planes, light sources, fog settings, visual properties, etc.
- **Texture State:** Including pixel data and parameters such as border colour or wrap coordinates. This information is kept for each texture associated to a context.
- **Display Lists:** A display list contains a series of OpenGL calls that are stored in GPU memory in a compact format, to optimize their execution as a single atomic operation at later times. For each display list associated to a context we keep a verbatim “unrolling” of its sequence of OpenGL calls.

Like the rest of VMGL, the code implementing OpenGL state restore resides in userspace, thus retaining OS and VMM independence. Furthermore, OpenGL state is independent of its representation in GPU memory by a particular vendor. Therefore, VMGL can suspend and resume applications across physical hosts equipped with GPUs from different vendors. OpenGL vendor extensions can present an obstacle to cross-vendor migration – we discuss this topic at the beginning of Section 4.3.5.

4.3.4 Porting GPU Drivers For Xen

VMMs such as VMware Workstation run an unmodified OS as the host, thus enabling the VMGL stubs to readily take advantage of hardware-specific drivers for direct rendering.

However, this is not the case for Xen. Its administrative VM, also known as domain 0, is itself a VM running a paravirtualized kernel, causing incompatibilities with closed-source drivers to arise.

Xen's architecture prevents virtual machines from modifying the memory page tables through direct MMU manipulation. Paravirtualized kernels in Xen need to invoke the `mmu_update` and `update_va_mapping` hypercalls to have Xen perform a batch of page table modifications on its behalf. Before manipulating the hardware MMU, Xen will sanity-check the requested changes to prevent unauthorized access to the memory of another virtual machine. Transferring MMU-manipulation responsibilities to the hypervisor has introduced another level of indirection in memory addressing: physical frame numbers in a domain kernel are mapped by Xen into *machine frame numbers*, the actual memory frame numbers handled by the MMU.

To enable direct rendering functionality, OpenGL implementations need to communicate with the graphics card. This is typically achieved by memory mapping a character device, which results in the kernel remapping GPU DMA areas into the GL library's address space. In the absence of IOMMU hardware support for virtualized DMA addressing [24], Xen needs to interpose on these operations, translate them to machine frame numbers, and sanitize them. Drivers included in the Linux kernel distribution and using the Direct Rendering Manager described in Section 4.2 (e.g. Intel's), use functions that Xen paravirtualizes to provide the proper DMA addressing. Unfortunately, this is not the case with the proprietary closed-source drivers of Nvidia and ATI cards, which for that reason do not work natively with Xen paravirtualized kernels.

Luckily, these drivers are wrapped by an open-source component that is recompiled to match the specifics of the current kernel. As long as all DMA mapping functions are contained in the open-source component, the proprietary driver can be adjusted to run in domain 0. We have ported the fglrx driver version 8.29.6 for an ATI Radeon X600 PCI-Express card. By changing the DMA mapping macro to use a paravirtualization-aware function, we were able to use the driver in domain 0. Similar modifications to the proprietary Nvidia driver version 1.0-8756 also provide direct rendering functionality for domain 0 [138].

4.3.5 Limitations

VMGL currently supports 59 OpenGL v1.5 extensions, including vertex programs, fragment programs, and 13 vendor-specific extensions. We are constantly working to extend VMGL support to more GL extensions. For instance, the Unreal Tournament 2004 benchmark used in the next Section demanded the implementation of a number of extensions including `GL_EXT_bgra`. The coverage provided by the current set of extensions supported by VMGL is extensive, as indicated by VMGL's widespread use for gaming and scientific visualization frontends. A current source of interest in VMGL is in adapting it as an enhancer of the SciRun [136] distributed scientific visualization engine. However, vendor-specific extensions could represent a source of incompatibility if a VMGL-enabled guest is resumed on a new physical host with a different GPU from the one available where it was last suspended. If the GPU at the resume site does not support some of the vendor-specific extensions in use by an application, we will have to temporarily map their functionality to supported variants, and possibly suffer a performance hit. An alternative solution is to altogether disable vendor-specific extensions, at the expense of sacrificing functionality in some cases.

VMGL currently does not support Windows or MacOS guests. We have not yet developed the necessary hooks into the windowing systems to provide functionality similar to that of our X server extension, although we do not anticipate any major obstacles in this area. Finally, the Direct3D API used by some Windows applications can be supported through Direct3D to OpenGL translation layers, such as WineD3D [38].

Our work so far has focused on portability across VMMs and guest operating systems. We have therefore avoided all performance optimizations that might compromise portability. By carefully relaxing this constraint, we anticipate being able to improve VMGL's performance. As we will show in Section 4.4, for very demanding applications at high levels of concurrency the total bandwidth between application VMs and the OpenGL stubs becomes the performance bottleneck. A shared-memory rather than network transport implementation could relieve this bottleneck. By implementing this optimization in a way that preserves the external interfaces of VMGL, we could enable VMM-specific and guest-specific code to be introduced with minimal negative impact on portability. The network transport would always remain a fallback for environments without support for a shared-memory transport.

4.4 Evaluation

Our evaluation of VMGL addresses the following questions:

Performance. How does VMGL compare to software rendering alternatives, such as the Mesa OpenGL library [120]? How close does it come to providing the performance observed with unvirtualized graphics acceleration?

Portability. Can VMGL be used with different VMMs? Can VMGL be used with different VM types supported by the same VMM? Can VMGL be used with different guest operating systems?

Suspend and Resume. What is the latency for resuming a suspended OpenGL application? What is the size of an application's OpenGL suspended state? Can we migrate suspended OpenGL applications across GPUs from different vendors?

Sensitivity to Resolution. What is the effect of rendering resolution on VMGL performance?

Sensitivity to Multiple Processors. How sensitive is VMGL to processing power? Can it take advantage of multi-core CPUs?

Scalability. How well does VMGL scale to support multiple VMs performing 3D drawing concurrently? A proposed use for VMs is the deployment of virtual appliances [194]. It is expected that users will run multiple virtual appliances in a single physical platform, with perhaps several appliances doing 3D rendering simultaneously.

4.4.1 Benchmarks

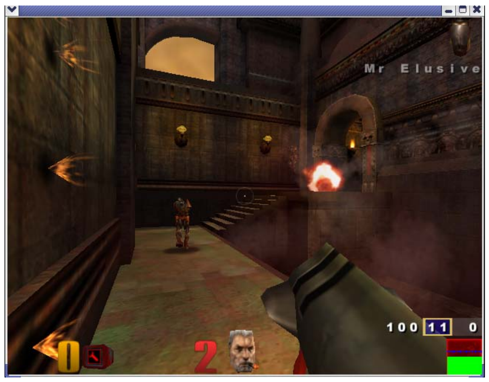
Application	Release Date
Quake 3	Dec, 1999
Wolfenstein: Enemy Territory	May, 2003
Unreal Tournament 2004	Mar, 2004
Mplayer	Jun, 2006

Table 4.1: Application Benchmarks

Table 4.1 summarizes the four benchmarks we use in the evaluation of VMGL. We focus our evaluation on computer games and entertainment as these classes of applications have effectively become the driving force in the development of consumer graphics applications and hardware [153]. The four applications benchmarked are as follows:

- **Quake 3**: Quake III Arena [89] (Figure 4.2 (a)), was first released in December, 1999. Quake 3 employs an extensive array of OpenGL drawing techniques [203], including shader scripts; volumetric textures, fog and lighting; vertex animation; Gouraud shading; spline-based curved-surfaces, and others. This set of features has enabled Quake 3, despite its relative age, to remain a popular application for benchmarking 3D performance [126, 187]. Quake 3 was open-sourced in 2005.

- **Enemy**: Wolfenstein Enemy Territory [169] (Figure 4.2 (b)) was released in May of 2003. The game is a third-generation successor to the Quake 3 engine, including enhancements such as skeletal animation and substantially increased texture and scenic detail. Enemy Territory's logic was open-sourced in 2004.



(a) Quake 3 Arena

(b) Enemy Territory

(c) Unreal Tournament 2004

(d) Mplayer

Figure 4.2: Benchmark screenshots

- **Unreal**: Unreal Tournament 2004 [54] (Figure 4.2 (c)) has a modern graphics engine [191] that exploits a variety of features such as vertex lighting, projective texturing, sprite or mesh particle systems, distance fog, texture animation and modulation, portal effects, and vertex or static meshes. Like Quake 3, Unreal is also heavily favoured by the industry as a *de facto* benchmark for 3D graphics performance [13, 187]. Unlike Quake 3 and Enemy, this application is closed source.

- **Mplayer**: Mplayer [127] (Figure 4.2 (d)) is a popular open source media player available for all major operating systems. It supports a number of different video codecs, and a number of output drivers, including texture-driven OpenGL output.

For the first three benchmarks, we replayed publicly available demos for 68 seconds (Quake 3 [89]), 145 seconds (Enemy [204]), and 121 seconds (Unreal [190]), respectively. For the Mplayer benchmark we replayed the first 121 seconds of a video clip encoded at two different resolutions.

4.4.2 Experimental Setup

We tested VMGL with two virtual machine monitors, Xen 3.0.3 [21] and VMware[1] Workstation 5.5.3 [196]. All experiments were run on a 2.4 GHz Intel Core2 machine with two single-threaded cores, VT hardware virtualization extensions, and 2 GB of RAM. For most experiments, we employed a Radeon X600 PCI-Express ATI graphics card; for a set of suspend and resume experiments we used an Intel 945G PCI-Express Graphics card. Regardless of the VMM, the machine ran the Fedora Core 5 Linux distribution with the 2.6.16.29 kernel in 32 bit mode (paravirtualization extensions added for use with Xen), and X.Org version 7.0 with the fglrx proprietary ATI driver version 8.29.6, or the DRI driver based on Mesa version 6.4.2 for the Intel card. All virtual machines were configured with the same kernel (modulo paravirtualization extensions for Xen), same

[1]We thank VMware for granting permission to publish the results of our evaluation.

distribution, 512 MB of RAM, and no swap. We ran each benchmark in three different configurations:

- **Native**: an unvirtualized environment with direct access to hardware and native OpenGL drivers. VMGL was not used. This represents the upper bound on achievable performance for our experimental setup.

- **Guest + Mesa Software Rendering**: a virtualized guest using software rendering provided by the Mesa OpenGL library. No hardware rendering facilities were used. This is the commonly available configuration for current users of 3D applications in virtualized environments. The Unreal benchmark refuses to run in the absence of hardware acceleration.

- **Guest + VMGL**: a virtualized guest using VMGL to provide 3D hardware acceleration. This configuration is depicted by Figure 4.1.

In each of these configurations, all benchmarks were executed at two different resolutions:

- **High Resolution**: The resolution was set to 1280x1024 except for Mplayer, which had a resolution of 1280x720 (the closest NTSC aspect ratio).

- **Low Resolution**: The resolution was set to 640x480 except for Mplayer, which had a resolution of 640x352 (the closest NTSC aspect ratio).

We quantify graphics rendering performance in terms of framerate or Frames per Second (FPS), a standard metric used for the evaluation of 3D graphics [13, 126, 187]. We also measure VMGL's resource utilization in terms of CPU load and network usage. All data points reported throughout the rest of this Section are the average of five runs. All bar charts have standard deviation error bars.

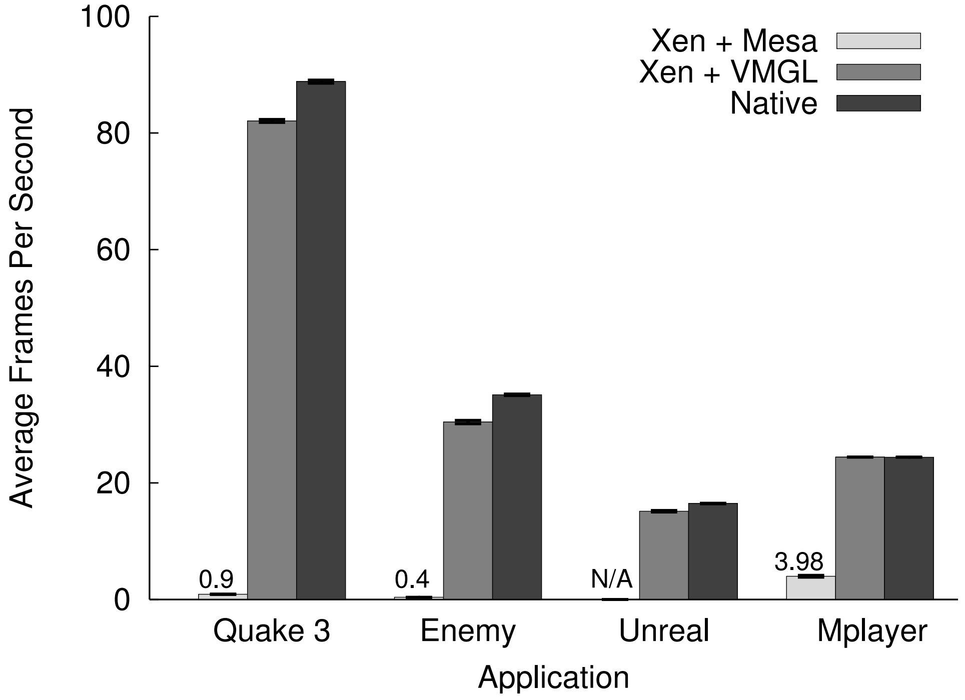


Figure 4.3: VMGL performance – Average FPS, high resolution.

4.4.3 Performance

Figure 4.3 shows the results from running the benchmarks under three configurations: native, Xen paravirtualized guest with VMGL, and Xen paravirtualized guest with Mesa software rendering. All benchmarks in the figure are run in high resolution mode.

First, we observe that VMGL's performance is two orders of magnitude better than software rendering. On the contrary, the number of FPS delivered by Mesa ranges from 0.4 to 4. From the user's perspective, this low framerate renders the applications unusable.

Next, we observe that VMGL's performance approximates that of the native configuration, with the performance drop ranging from 14% for the Enemy benchmark to virtually no loss for the Mplayer and Unreal benchmarks. In our subjective experience, the user experience delivered by VMGL is indistinguishable from that of the native configuration for all of the benchmarks.

Figure 4.3 reports a global average metric. To further understand VMGL's perfor-

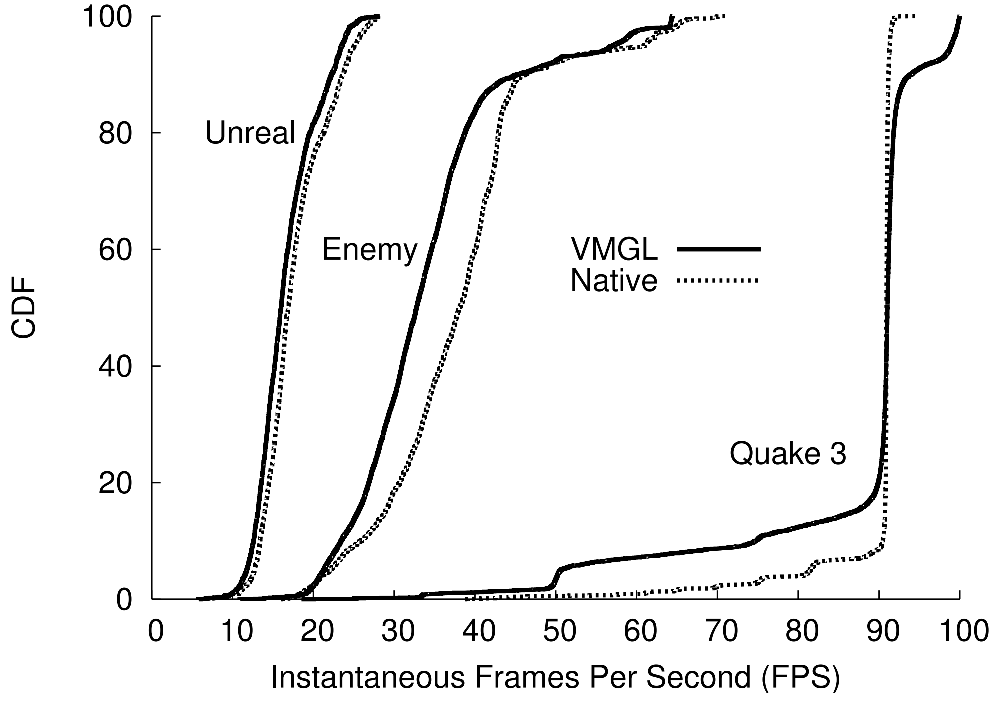


Figure 4.4: VMGL performance – FPS variability, high resolution.

mance we need to compare its variations in framerate and peak FPS values against those observed under native execution. Crisp interaction with highly detailed graphics applications not only demands a high framerate, but also a uniform experience without jitter [60]. Figure 4.4 plots a cumulative distribution function for the instantaneous FPS across all five trials on each benchmark. Plots to the right indicate better performance than plots to the left; the more vertical a plot is, the smaller variability in framerate. We exclude Mesa results given their very low quality; we also exclude the Mplayer benchmark as it presents a constant framerate of 25 FPS across the remaining configurations. VMGL results closely follow the behaviour of their native execution counterparts. The variability in frame rates is consistent with that observed under native execution. Differences in the framerate distribution and peak FPS values with respect to native execution are minimal.

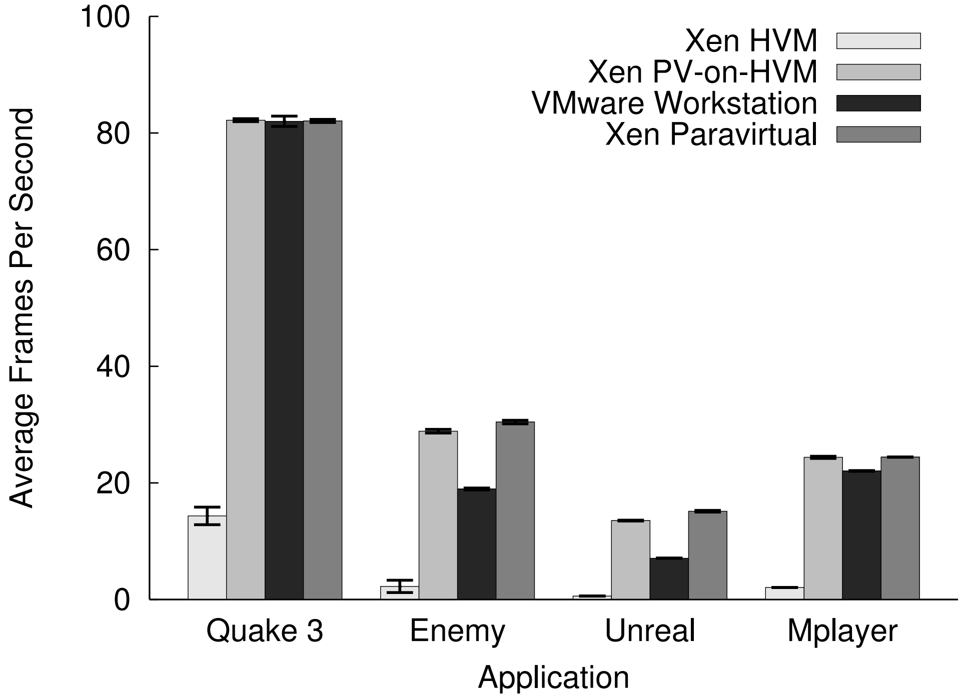


Figure 4.5: VMM portability – High resolution.

4.4.4 VMM Portability

Figure 4.5 shows VMGL's performance for one VMware Workstation and three Xen configurations: *Xen HVM* leverages Intel's VT extensions to run an unmodified Linux kernel as a guest, and emulates network I/O using code derived from the QEMU emulator [23]; *Xen PV-on-HVM* is similar to Xen HVM, but a loadable kernel module provides the guest with Xen-aware paravirtualized network functionality; *VMware Workstation* runs an unmodified Linux kernel as the guest OS and uses VMware Tools for proprietary network virtualization; Finally, *Xen Paravirtual* is the same Xen paravirtualized guest configuration as that of the *Xen + VMGL* bars of Figure 4.3.

As expected, Figure 4.5 shows that the quality of network virtualization is a fundamental factor affecting VMGL's performance. Without paravirtualized extensions, a Xen HVM presents very low FPS ratings. The PV-on-HVM configuration provides almost identical performance to that of Xen paravirtualized guests. VMware Workstation's similar use of virtualization-aware drivers on an otherwise unmodified OS also yields an

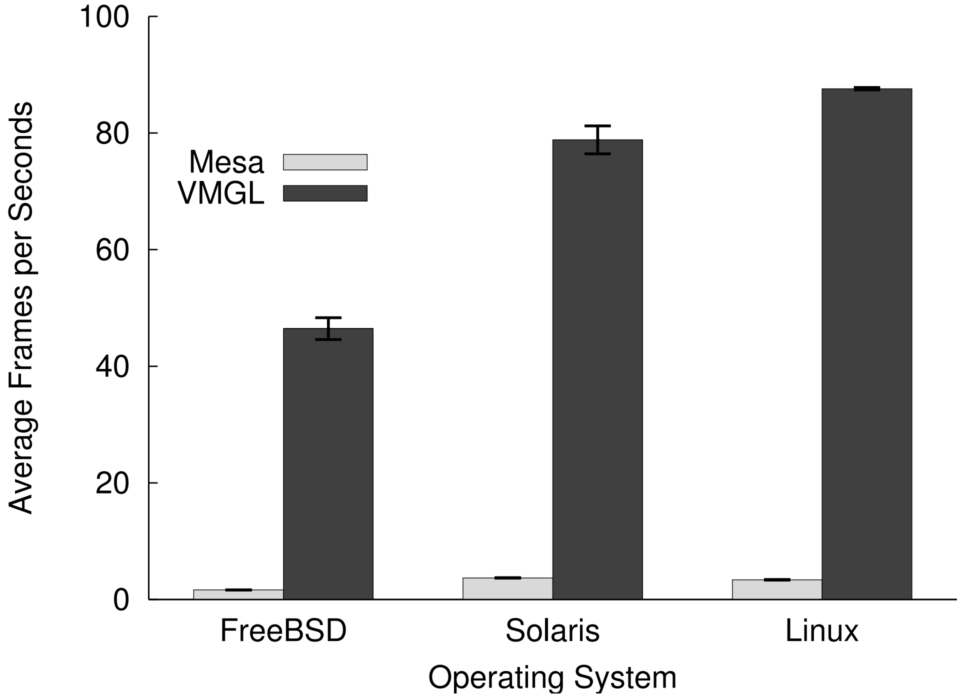


Figure 4.6: Guest OS portability – Quake 3, VMware Workstation guests.

order of magnitude better performance than a pure Xen HVM.

4.4.5 Portability Across Guest Operating System

VMGL userspace design and its implementation in standard programming languages makes it easy to port across operating systems. In particular, we have ported VMGL to FreeBSD release 6.1 and OpenSolaris 10 release 06/06. The source code logic remained unmodified. All necessary changes had to do with accounting for differences in the OS development environment, such as header inclusion, library linking, and tools used in the build process.

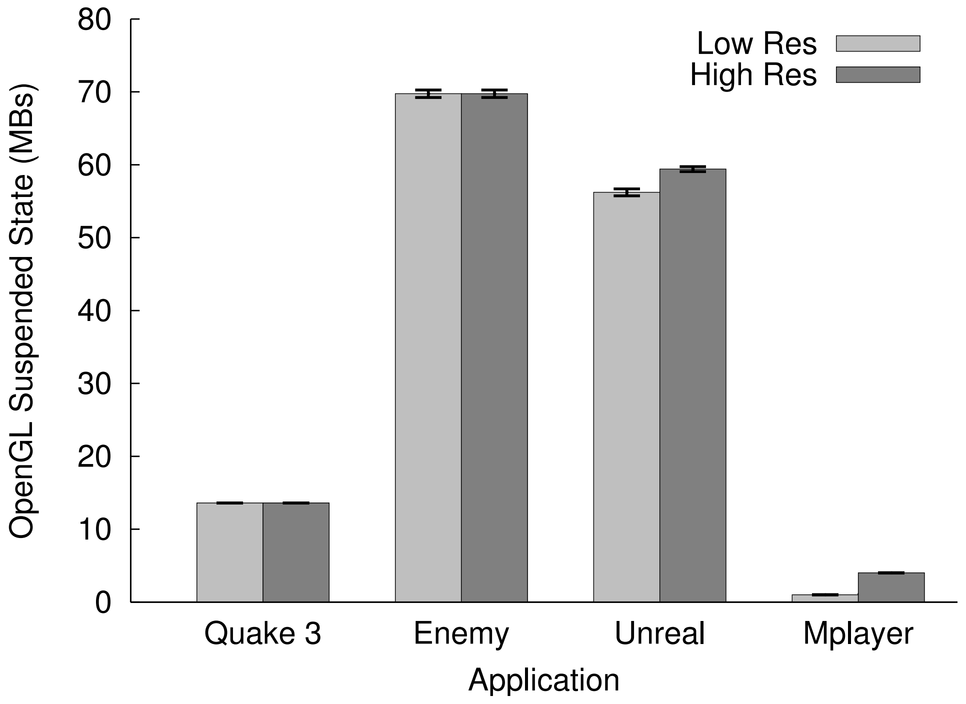
To test our VMGL port for these two operating systems, we configured them as VMware Workstation guests running the open-source Quake 3 port ioquake3 (Quake 3's authors did not port the application to OpenSolaris or FreeBSD). Figure 4.6 compares the performance of Mesa software rendering and VMGL accelerated rendering for each OS, including Linux. While FreeBSD did not perform in general as well as OpenSolaris, in both cases VMGL conserves its notable performance advantage over software rendering. While we did not configure our experimental machine to natively run FreeBSD or OpenSolaris, we are confident VMGL will show a trend similar to that shown with Linux and maintain performance on par with an unvirtualized configuration.

4.4.6 Suspend and Resume

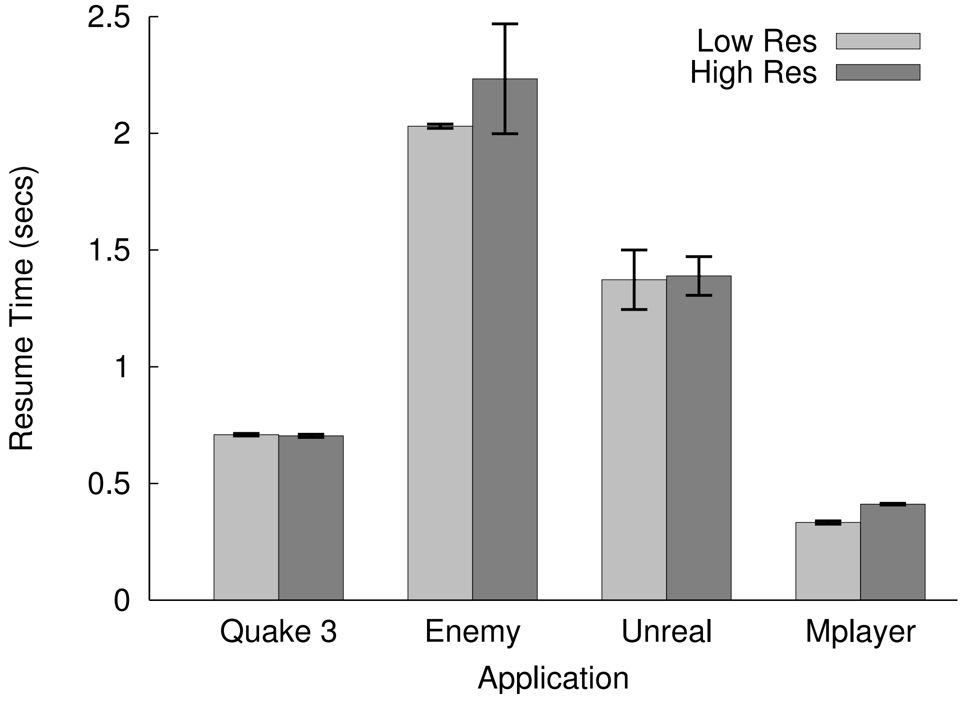
To measure the performance of VMGL's suspend and resume code, we suspended a guest running the benchmarks at five different and arbitrary points in time. We then resumed the guest and verified successful resumption of the OpenGL application. We measured the size of the OpenGL state necessary to synchronize the GL stub to the current application state, and the time it took to perform the entire resume operation. We did not observe any noticeable effect of the suspend and resume code on the application's framerate performance. The results of these experiments are displayed in Figure 4.7. This Figure displays results for both application resolutions obtained with Xen paravirtualized guests; similar results were obtained with VMware Workstation guests.

The resume time (Figure 4.7 (b)) is strongly dependent on the size of the suspended OpenGL state (Figure 4.7 (a)), which can be as large as 70 MB for the Enemy benchmark. Nevertheless, the latency for reinstating the suspended OpenGL state on a new VMGL stub never exceeded 2.5 seconds. Regardless of the suspend point, the size of Mplayer's state is always the same, as this state is almost exclusively composed of the texture corresponding to the current frame. Since the frame is twice as big on each dimension, the state is four times larger in high resolution mode than in low resolution mode. Finally, we were surprised to note that the size of Quake 3's OpenGL state is also invariant with respect to the suspend point. We conjecture that Quake 3 pre-allocates the entire OpenGL state for a new environment before allowing interaction.

We performed a second set of experiments in which we suspended and resumed a guest across two different hosts: our experimental machine and a similar physical host using



(a) Size of suspended OpenGL state



(b) Resume time

Figure 4.7: Suspend and resume – Xen Paravirtual guest.

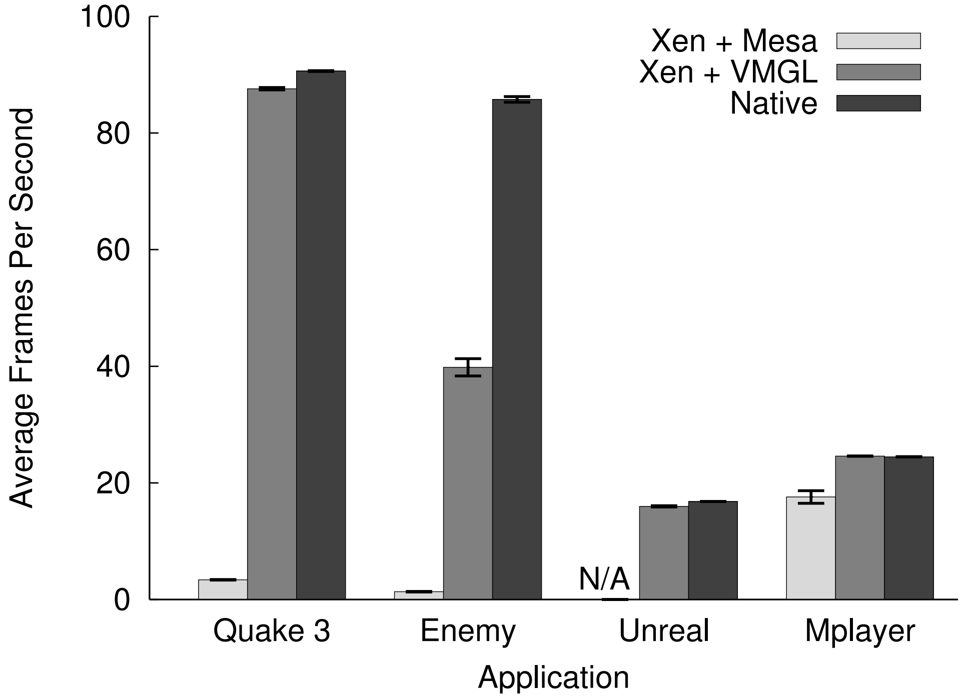


Figure 4.8: Benchmarks in low resolution mode.

an Intel 945G GPU. The tests completed successfully with similar latency and state size results. In VMGL, we had to disable five extensions provided by the ATI card but not by Intel's (including `GL_ARB_occlusion_query`, for example), and four extensions available in the Intel card but not in ATI's (including `GL_NV_texture_rectangle`).

4.4.7 Sensitivity to Screen Resolution

OpenGL drawing primitives use a normalized coordinate system, and rely on the hardware capabilities of the graphics card to scale the geometry to match the current screen resolution. This implies that the higher the resolution, the busier the GPU and therefore the less noticeable the VMGL command marshaling overhead becomes. The slightly counter-intuitive consequence is that it is preferable to run applications under VMGL at higher resolutions, something which is desirable anyway.

Figure 4.8 shows the results from running the benchmarks at low resolution (640x480, Mplayer runs at 640x352) for three configurations: Xen with Mesa, Xen with VMGL, and native. The first three benchmarks generate the same stream of OpenGL commands as in the high resolution experiments (Figure 4.3), and rely on automatic scaling. Mplayer is different, as each frame is generated by synthesizing an appropriately sized texture from the input video data, and therefore it does not involve any hardware scaling. The increased pressure on the VMGL transport is evident for the Enemy benchmark, presenting a performance drop with respect to the unvirtualized baseline to approximately half the rate of frames per second. However, for the remaining three benchmarks the performance of Xen+VMGL closely matches that of the native configuration. Software rendering is still unable to provide reasonable performance, perhaps with the exception of the Mplayer benchmark achieving 17.6 average FPS due to the smaller sized frames.

For the remainder of this evaluation, we concentrate on low-resolution experiments as they bias the results *against* VMGL.

4.4.8 Sensitivity to Multi-Core Processing

To determine the benefits that VMGL derives from multi-core processing, we also executed all of our application benchmarks after disabling one of the two cores in our experimental machine. These results, presented in Figure 4.9, show a performance drop for Enemy and Unreal, the more modern applications. There is no significant difference for the older applications.

We analyze the benefits arising from a multi-core setup using Unreal as an example. Figures 4.10(a) to 4.10(c) show the differences in resource usage for the single and multi-core cases. The increased CPU utilization possible with dual-core parallelism (Figure 4.10(b)) results in a higher rate of OpenGL commands pushed per second through the VMGL transport (Figure 4.10(c)). The consequence is a higher framerate in the dual-core case (Figure 4.10(a)). Unreal's behaviour seems to be work-conserving: rather than dropping frames at a low framerate, it takes longer to complete the demo.

The presence of two cores leads to increased resource utilization for a number of rea-

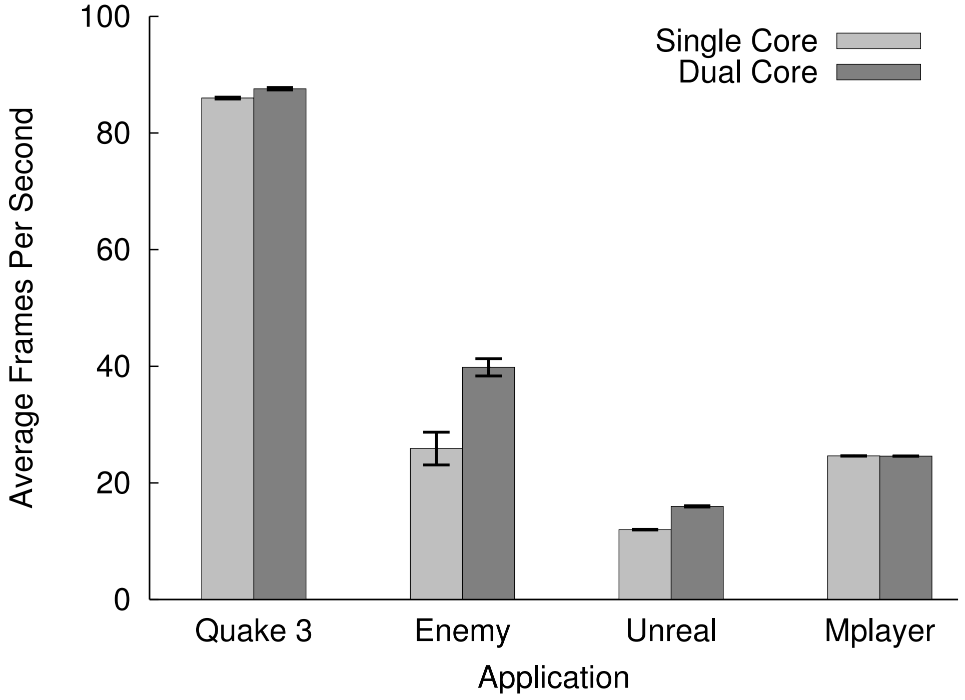


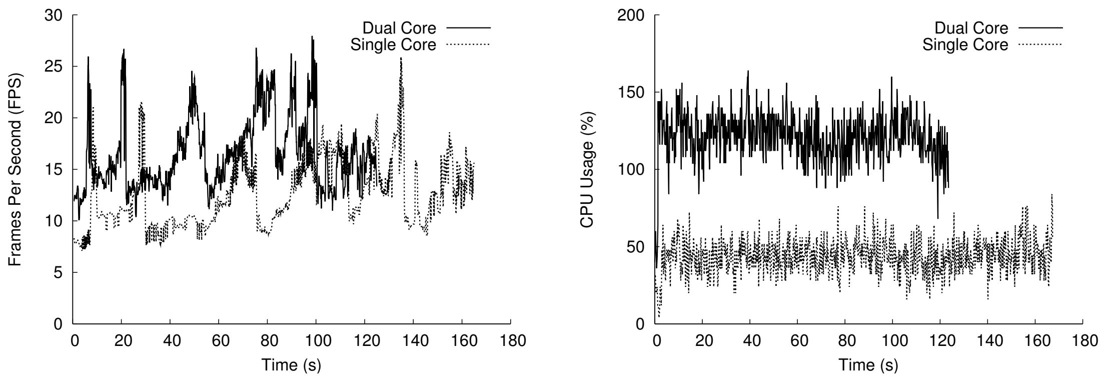
Figure 4.9: CPU sensitivity – Xen paravirtual + VMGL, low resolution.

sons. First, multiple cores allow concurrent execution for the two networking stacks: in the guest where the application executes and in the host where the viewer resides. It also allows for parallelizing the marshaling and unmarshaling cycles of OpenGL commands by VMGL. The availability of two cores also ameliorates the VMM's overhead of constantly needing to context switch between the two VMs, and to switch to the hypervisor to handle the interrupts generated by the bridged networking setup, a previously documented overhead [119, 118].

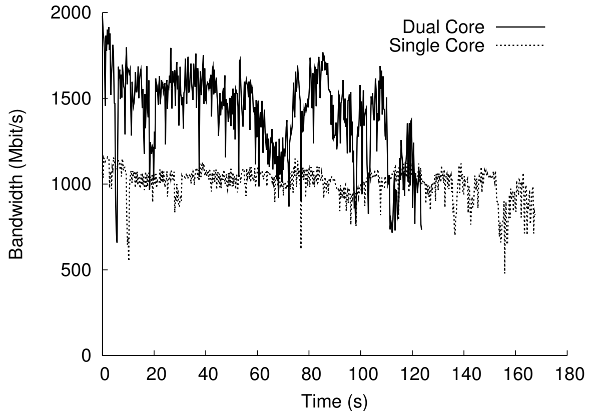
4.4.9 Concurrent Guests

To examine VMGL's ability to support concurrent guests, we compare the performance of two instances of an application executing concurrently in an unvirtualized configuration, to the performance of two instances executing in two separate Xen paravirtual guests.

Figure 4.11 (a) presents the average per-instance FPS results for the concurrent execution of two instances, compared to the average FPS results for a single instance (taken



(a) Instantaneous frames per second (b) CPU usage

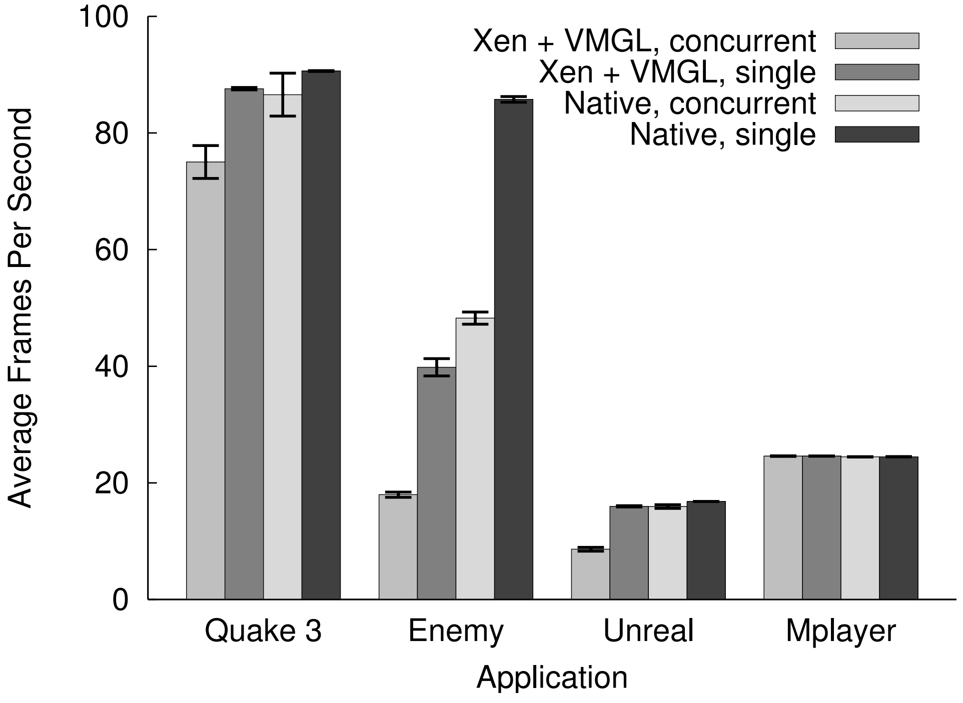


(c) Network usage

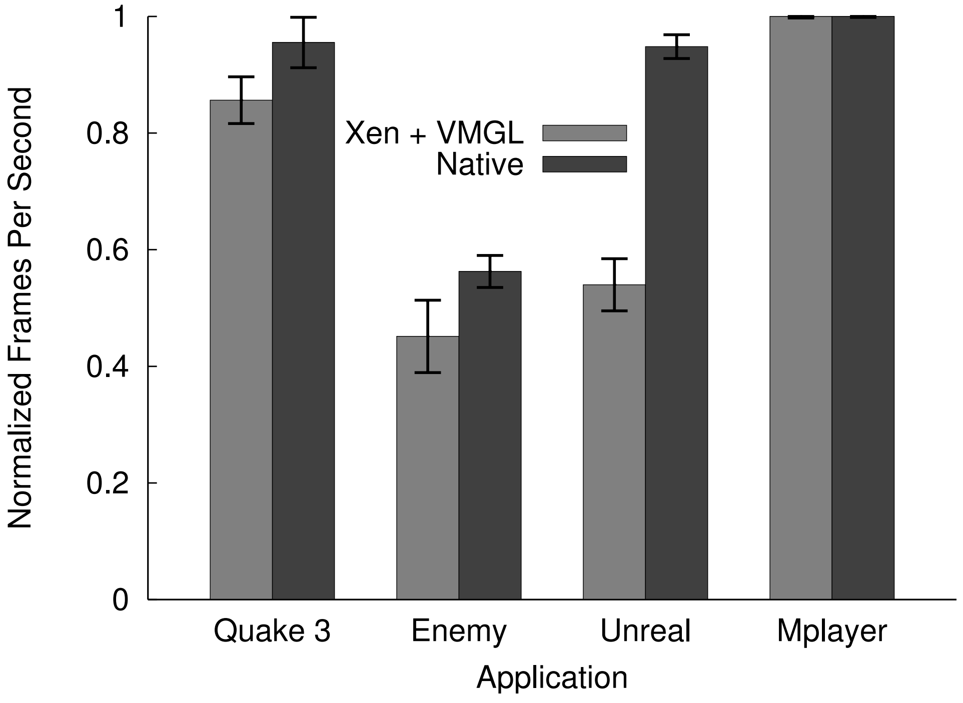
Figure 4.10: Unreal instantaneous FPS, and CPU and network usage on dual- vs. single-core configurations, low resolution. CPU utilization includes all components depicted in Figure 4.1. With a single-core, the benchmark takes longer to complete due to the reduced framerate.

from Figure 4.8). Figure 4.11 (b) normalizes the concurrent execution results against the single-instance results (i.e. *Xen + VMGL, concurrent* divided by *Xen + VMGL, single*). The purpose of normalization is to observe the "natural" scalability inherent to the application: simultaneous instances may compete for CPU and memory resources. The additional drop in normalized FPS for the VMGL configurations reflects the overheads of GL marshaling and context-switching VMs.

The Mplayer benchmark, more representative of a multiple virtual appliance scenario, presents excellent scalability results. We observe decreasing VMGL scalability as the application becomes more heavyweight and places a larger demand on the GL transport: 10% additional overhead for Quake 3, 20% for Enemy, and 43% for Unreal. Figure 4.10(c)



(a) Raw per-instance FPS



(b) Normalized FPS

Figure 4.11: Concurrent guests with Xen paravirtual: FPS for running two simultaneous instances of each benchmark at low resolution.

indicates that the bandwidth demands of a single instance of Unreal can peak at almost 2 Gbps. Extreme configurations with multiple high-end applications rendering concurrently may impose an aggregate bandwidth demand on VMGL of several Gbps. Under such circumstances, a VMM-specific shared memory transport may be preferable.

4.5 Work Related to VMGL

The VMGL project addresses the intersection of virtualization and specialized hardware acceleration, predating most of the work done in the area. In this Section we address work that surfaced between the publication of VMGL and the writing of this thesis.

Several vendors have adopted API virtualization as a method to provide hardware graphics acceleration to VMs. Parallels virtualizes Direct3D [135], while VirtualBox has recently added support for OpenGL virtualization [192]. The best known work in this field is VMware's virtualization of Direct3D through a hybrid approach [47]. VMware provides a virtual graphics card with an API that closely matches that of Direct3D. VMware's Direct3D library is thus a simple pass-through mechanism, while the bulk of the virtualization work remains in the virtual card internals. The promise of this architecture is in having better control over the memory buffers and DMA areas used by the actual GPU hardware. This could be understood as a weakness of VMGL for applications that aggressively use shader algorithms and rich vertex and texture buffers.

Concurrently with VMGL, the Blink [78] system for the Xen hypervisor strived to provide virtualization of OpenGL primitives for VMs. Blink multiplexes graphical content onto a virtual GPU, with an emphasis on safety in the face of multiple untrusted clients. Blink is VMM-specific as it uses a shared memory transport and does not support suspending or migrating a VM.

In [167], a hybrid approach similar to VMware's is proposed, using the nascent Gallium3D [72] architecture. Gallium abstracts a common set of operations that most GPUs are bound to support. In this way, 3D drivers can be architected as a common set of operations with a set of device-specific backends. While the jury's out on the ability of such an architecture to survive in a space dominated by vendor lock-in, the work offers the potential to substantially simplify the development of multiple virtualization strategies for different backends and frontends. Instead of developing each component in the cartesian product space of guest OS vs. backing device, frontends for each OS can be produced independently of the backends for each device. At this point we must note that VMGL transparently supports any backing device, and that the porting effort to different guest OSs has been trivial – for X11-based systems, admittedly.

An important area of future work for VMGL is the possibility to provide access to GPGPU models of computation, in which a GPU is leveraged as a heterogeneous computing element providing a large number of parallel arithmetic pipelines. Although unexplored, VMGL could potentially support this model, as GPGPU code is synthesized into OpenGL shader programs. However, the gVIM work [76] transposes the idea of API remoting to the CUDA environment, the API provided for NVIDIA to develop GPGPU codes. This work is strikingly similar to VMGL in that it relies on API remoting to leverage GPU hardware. One important problem for GPGPU computation is the managing of memory buffers for the card to efficiently consume data and produce results, avoiding multiple unnecessary stages of copying in between user-space, kernel-space, and the internal GPU buffers. gVIM does not address this problem in earnest, which could become an important bottleneck for generalized high-performance computation (as well as for speedy high-detail graphics rendering).

4.6 Future Directions and Conclusion

GPUs are critical to high-quality visualization in many application domains. Running such applications in VMM environments is difficult for a number of reasons, all relating to the fact that the GPU hardware interface is proprietary rather than standardized. VMGL virtualizes the OpenGL software interface, recognizing its widespread use in graphics-intensive applications. By virtualizing at the API level, VMGL is able to support multiple guest OSs and to provide suspend and resume capabilities across GPUs from different vendors. Our experiments confirm excellent rendering performance with VMGL, coming within 14% or better of native hardware accelerated performance measured in frames per second. This is two orders of magnitude better than software rendering, which is the commonly available alternative today for graphics-intensive applications in virtualized environments.

Our results also show that the resource demands of VMGL align well with the emerging trend of multi-core processors. In other words, there is natural and easy-to-exploit parallelism in the VMGL architecture. Our work thus reveals an opportunity for three emerging trends (virtualization, growing use of GPUs by applications, and multi-core processing) to evolve in a mutually supportive way.

VMGL is an open source software project [106] that has been downloaded over ten thousand times. Its widespread adoption is due to its timeliness, its ease of use, and the coverage it provides in terms of supporting a very large and useful set of OpenGL primitives, and supporting multiple VMMs.

As outlined in Sections 4.1 and 4.3.5, VMGL presents ample space for future work, partly due to some of its limitations. First, VMGL does not straight-forwardly support non-X11 guests such as Windows or MacOS. As shown in Section 4.5, graphics API virtualization for these operating systems has been made available recently, following the same principles employed by VMGL. Second, VMGL's goal of supporting multiple VMMs imposed the adoption of a VMM-agnostic transport such as the network-based one currently in use. Performance of the system will be highly improved by adding VMM-specific shared memory transports. A shared memory transport will eliminate the data copying and processing burden of running two parallel TCP/IP stacks, one in the host and another one in the guest. Third, we believe VMGL can be extremely useful to high-performance computing programmers wishing to leverage the power of graphical processors in a GPGPU setting coupled with virtualization. Finally, the suitability of VMGL as a thin client protocol may be of interest in certain high-bandwidth low-latency scenarios such as an enterprise LAN. With applications aggressively adopting 3D-based GUIs, a thin client protocol expressing graphics transformations as 3D API commands will be likely to perform much better than current protocols, which essentially screen-scrape and transmit pixel maps.

VMGL allows applications packaged as x86 VMs to take advantage of GPU hardware in a manner that is VMM and vendor independent. This property makes VMGL a key enabler to the vision in this thesis, by allowing modern graphics-intensive applications to seamlessly execute on rich edge clients that serve primarily as displays and local caches of state. VMGL complements the capabilities of Snowbird in allowing applications to move between the core and edges of the network without sacrificing the crisp quality of graphics-intensive end-user interaction.

Chapter 5

Scale Flexibility with SnowFlock

In this chapter we describe how we tackle the problem of scale flexibility in clouds of virtualized machines with the SnowFlock project [105, 104].

5.1 Introduction

Cloud computing is transforming the computing landscape by shifting the hardware and staffing costs of managing a computational center to third parties such as Yahoo! [206] or Amazon [10]. Small organizations and individuals are now able to deploy world-scale services: all they need to pay is the marginal cost of actual resource usage. Virtual machine (VM) technology is widely adopted as an enabler of cloud computing. Virtualization provides many benefits, including security, performance isolation, ease of management, and flexibility of running in a user-customized environment.

A major advantage of cloud computing is the ability to use a variable number of physical machines and VM instances depending on the needs of the problem. For example a task may need only a single CPU during some phases of execution but may be capable of leveraging *hundreds* of CPUs at other times. While current cloud APIs allow for the instantiation of new VMs, their lack of agility fails to provide users with the full potential of the cloud model. Instantiating new VMs is a slow operation (typically taking "minutes" [10]), and the new VMs originate either as fresh boots or replicas of a template VM, unaware of the current application state. This forces cloud users into employing ad hoc solutions that require considerable developer effort to explicitly propagate application state and waste resources by pre-provisioning worker VMs that remain mostly idle. Moreover, idle VMs are likely to be consolidated and swapped out [171, 205], incurring costly migration delays before they can be used.

We introduce VM fork, an abstraction that simplifies development and deployment of cloud applications that dynamically change their execution footprint. VM fork allows for the *rapid* ($<$ 1 second) instantiation of *stateful* computing elements in a cloud environment. While VM fork is similar in spirit to the familiar UNIX process fork, in that the child VMs receive a copy of all of the state generated by the parent VM prior to forking, it is different in three fundamental ways. First, our VM fork primitive allows for the forked copies to be instantiated on a set of different physical machines, enabling the task to take advantage of large compute clusters. Second, we have made our primitive *parallel*, enabling the creation of multiple child VMs with a single call. Finally, our VM fork replicates all of the processes and threads of the originating VM. This enables effective replication of multiple cooperating processes, e.g. a customized LAMP (Linux/Apache/MySql/Php) stack. Previous work [198], while capable of replicating VMs, is limited to cloning VMs with limited functionality and a short life span to individual replicas executing in the same host.

VM fork enables the trivial implementation of several useful and well-known patterns that are based on stateful replication, e.g., inheriting initialized data structures when spawning new workers. Pseudocode for four of these is illustrated in Figure 5.1: sandboxing of untrusted code, enabling parallel computation, instantiating new worker nodes to handle increased load (e.g. due to flash crowds), and opportunistically utilizing unused cycles with short tasks. All four patterns not only exploit fork's ability to create *stateful* workers, but also exploit its ability to *instantaneously* create workers.

(a) Sandboxing

```
state = trusted_code()
ID = VM_fork(1)
if ID.isChild():
 untrusted_code(state)
 VM_exit()
else:
 VM_wait(ID)
```

(b) Parallel Computation

```
ID = VM_fork(N)
if ID.isChild():
 parallel_work(data[ID])
 VM_exit()
else:
 VM_wait(ID)
```

(c) Load Handling

```
while(1):
 if load.isHigh():
  ID = VM_fork(1)
  if ID.isChild():
   while(1):
    accept_work()
 elif load.isLow():
  VM_kill(randomID)
```

(d) Opportunistic Job

```
while(1):
 N = available_slots()
 ID = VM_fork(N)
 if ID.isChild():
  work_a_little(data[ID])
  VM_exit()
 VM_wait(ID)
```

Figure 5.1: Four programming patterns based on fork's stateful cloning. Forked VMs use data structures initialized by the parent, such as `data` in case (b). Note the implicit fork semantics of instantaneous clone creation.

SnowFlock, our implementation of the VM fork abstraction, provides swift parallel stateful VM cloning with little runtime overhead and frugal consumption of cloud I/O resources, leading to good scalability. SnowFlock takes advantage of several key techniques. First, SnowFlock utilizes *lazy state replication* to minimize the amount of state propagated to the child VMs. Lazy state replication allows for extremely fast instantiation of clones by initially copying the minimal necessary VM data, and transmitting only the fraction of the parent's state that clones actually need. Second, a set of *avoidance heuristics* eliminate substantial superfluous memory transfers for the common case of clones allocating new private state. Finally, exploiting the likelihood of child VMs to execute very similar code paths and access common data structures, we use a *multicast distribution* technique for VM state that provides scalability and prefetching.

We evaluated SnowFlock by focusing on a demanding instance of Figure 5.1 (b): interactive parallel computation, in which a VM forks multiple workers in order to carry out a short-lived, computationally-intensive parallel job. We have conducted experiments with applications from bioinformatics, quantitative finance, rendering, and parallel compilation. These applications are deployed as Internet services [159, 55] that leverage mass parallelism to provide interactive (tens of seconds) response times to complex queries such as finding candidates similar to a gene, predicting the outcome of stock options, rendering an animation, etc. On experiments conducted with 128 processors, SnowFlock achieves speedups coming within 7% or better of optimal execution, and offers sub-second VM fork irrespective of the number of clones. SnowFlock is an order of magnitude faster and sends two orders of magnitude less state than VM fork based on suspend/resume or migration.

5.2 VM Fork: Usage and Deployment Model

The VM fork abstraction lets an application take advantage of cloud resources by forking multiple copies of its VM, that then execute independently on different physical hosts. VM fork preserves the isolation and ease of software development associated with VMs, while greatly reducing the performance overhead of creating a collection of identical VMs on a number of physical machines.

The semantics of VM fork are similar to those of the familiar process fork: a *parent* VM issues a fork call which creates a number of clones, or *child* VMs. Each of the forked VMs proceeds with an identical view of the system, save for a unique identifier (*vmid*) which allows them to be distinguished from one another and from the parent. However, each forked VM has its own independent copy of the operating system and virtual disk, and state updates are not propagated between VMs.

A key feature of our usage model is the ephemeral nature of children. Forked VMs are transient entities whose memory image and virtual disk are discarded once they exit. Any application-specific state or values they generate (e.g., a result of computation on a portion of a large dataset) must be explicitly communicated to the parent VM, for example by message passing or via a distributed file system.

The semantics of VM fork include integration with a dedicated, isolated virtual network connecting child VMs with their parent. Upon VM fork, each child is configured with a new IP address based on its vmid, and it is placed on the parent's virtual network. Child VMs cannot communicate with hosts outside this virtual network. Two aspects of our design deserve further comment. First, the user must be conscious of the IP reconfiguration semantics; for instance, network shares must be (re)mounted after cloning. Second, we provide a NAT layer to allow the clones to connect to certain external IP addresses. Our NAT performs firewalling and throttling, and only allows external inbound connections to the parent VM. This is useful to implement for example a web-based frontend, or allow access to a dataset provided by another party.

VM fork has to be used with care as it replicates all the processes and threads of the parent VM. Conflicts may arise if multiple processes within the same VM simultaneously invoke VM forking. For instance, forking a VM that contains a full desktop distribution with multiple productivity applications running concurrently is certainly not the intended use of VM fork. We envision instead that VM fork will be used in VMs that have been carefully customized to run a single application or perform a specific task, such as serving a web page. The application has to be cognizant of the VM fork semantics, e.g., only the "main" process calls VM fork in a multi-process application.

VM fork is concerned with implementing a fast and scalable cloning interface for VMs. How those clones VMs are mapped to underlying physical resources, scheduled, load balanced, etc, is up to resource management software. There is an abundance of commercial resource management software that is particularly adept at following workplans, observing user quotas, enforcing daily schedules, accommodating for downtimes, etc. All of these activities exceed the aim of VM fork, which aims to treat the resource management software as an independent and modular entity indicating where VM clones should be spawned.

We also note that VM Fork is not a parallel programming paradigm. VM Fork is meant to efficiently expand a single VM into a set or cluster of identical VMs. This is desirable because it is easy to reason about identical VMs, and it is straightforward to adapt the notion of an instantaneous cluster of identical VMs to multiple frameworks that require scaling of computation. VM fork thus easily accommodates frameworks that provide parallel programming facilities, or sandboxing, or scaling of web servers, to name a few examples. However, VM fork does not provide in itself facilities for parallel programming, such as shared memory regions, locks, monitors, semaphores, etc. There is great potential for future work in exploring the synergy between VM fork and parallel programming libraries. We present some of this work, which is outside of the scope of this thesis, in Section 5.4.2.

Finally, VM fork is not a replacement for any storage strategy. There is a great wealth of knowledge and research effort invested into how to best distribute data to compute nodes for data-intensive parallel applications. As a result of the high variability in application data patterns and deployment scenarios, many high performance storage solutions exist. VM fork does not intend to supersede or replace any of this work. VM fork is concerned with automatically and efficiently creating multiple replicas of a VM, understanding the VM as the unit that encapsulates the code and execution. We thus envision a deployment model in which datasets are not copied into the disk of a VM but rather mounted externally. VM fork will thus clone VMs to execute on data which is served by a proper big-data storage mechanism: PVFS [31], pNFS [143], Hadoop FS [182], etc. We further reflect on the interesting challenges arising from this separation of responsibility in Sections 5.4.7 and 5.6.

5.3 Challenges and Design Rationale

Performance is the greatest challenge to realizing the full potential of the VM fork paradigm. VM fork must *swiftly* replicate the state of a VM to many hosts simulta-

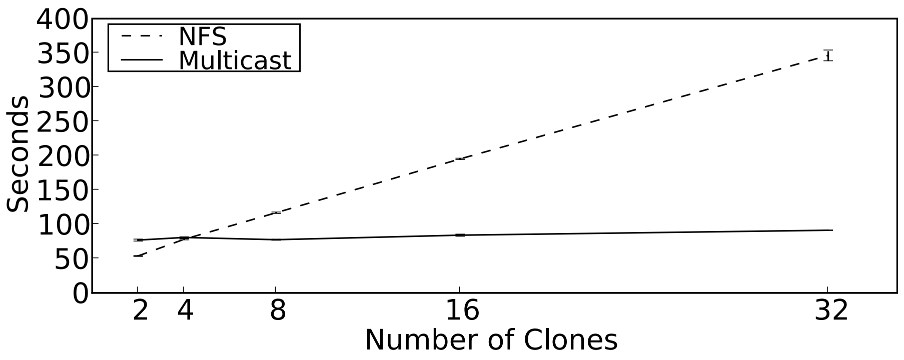


Figure 5.2: Latency for forking a 1GB VM by suspending and distributing the image over NFS and multicast.

neously. This is a heavyweight operation as VM instances can easily occupy GBs of RAM. While one could implement VM fork using existing VM suspend/resume functionality, the wholesale copying of a VM to multiple hosts is far too taxing, and decreases overall system scalability by clogging the network with gigabytes of data.

Figure 5.2 illustrates this by plotting the cost of suspending and resuming a 1GB VM to an increasing number of hosts over NFS (see Section 5.5 for details on the testbed). As expected, there is a direct relationship between I/O involved and fork latency, with latency growing to the order of hundreds of seconds. Moreover, contention caused by the simultaneous requests by all children turns the source host into a hot spot. Despite shorter downtime, live migration [35, 195], a popular mechanism for consolidating VMs in clouds [171, 205], is fundamentally the same algorithm plus extra rounds of copying, thus taking longer to replicate VMs.

A second approximation to solving the problem of VM fork latency uses our multicast library (see Section 5.4.6) to leverage parallelism in the network hardware. Multicast delivers state simultaneously to all hosts. Scalability in Figure 5.2 is vastly improved. However, overhead is still in the range of minutes and several gigabytes of VM state are still transmitted through the network fabric. To move beyond this, *we must substantially reduce the total amount of VM state pushed over the network.*

Our fast VM fork implementation is based on the following four insights: (i) it is possible to start executing a child VM on a remote site by initially replicating only minimal state; (ii) children will typically access only a fraction of the original memory image of the parent; (iii) it is common for children to allocate memory after forking; and (iv) children often execute similar code and access common data structures.

The first two insights led to the design of *VM Descriptors*, a lightweight mechanism which instantiates a new forked VM with only the critical metadata needed to start execution on a remote site, and *Memory-On-Demand*, a mechanism whereby clones lazily fetch portions of VM state over the network as it is accessed. Our experience is that it is possible to start a child VM by shipping only 0.1% of the state of the parent, and that children tend to only require a fraction of the original memory image of the parent. Further, it is common for children to allocate memory after forking, e.g., to read portions of a remote dataset or allocate local storage. This leads to fetching of pages from the parent that will be immediately overwritten. We observe that by augmenting the guest OS with *avoidance heuristics*, memory allocation can be handled locally by avoiding fetching pages that will be immediately recycled, thus exploiting our third insight. We show in Section 5.5 that this optimization can reduce communication drastically to a mere 40MBs for application footprints of 1GB ($\simeq$4%). Although these observations are based on our work with parallel workloads, they are likely to hold in other domains where a parent node spawns children as workers that execute limited tasks, e.g., load handling in web services.

Compared to ballooning [199], memory-on-demand is a non-intrusive approach that reduces state transfer without altering the behaviour of the guest OS. Ballooning a VM down to the easily manageable footprint that our design achieves would trigger swapping and lead to abrupt termination of processes. Another non-intrusive approach for minimizing memory usage is copy-on-write, used by Potemkin [198]. However, copy-on-write limits Potemkin to cloning VMs within the same host whereas we fork VMs across physical hosts. Further, Potemkin does not provide runtime stateful cloning, since all new

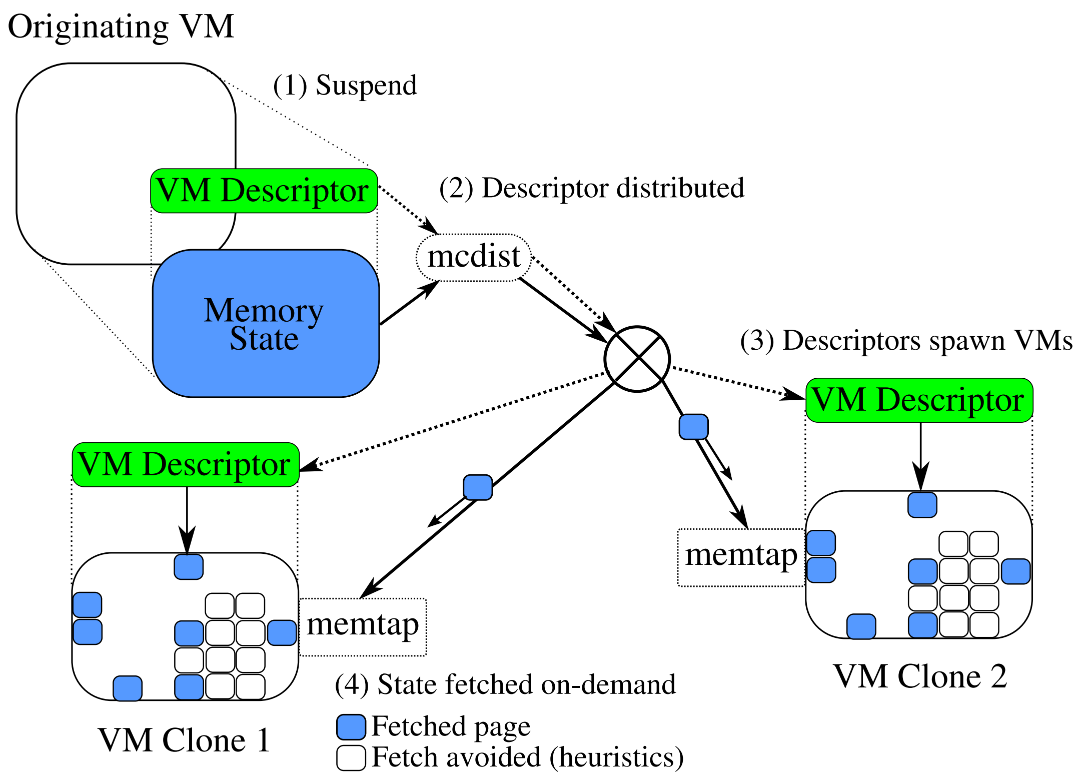
VMs are copies of a frozen template.

To take advantage of the fourth insight of high correlation across memory accesses of the children, and to prevent the parent from becoming a hot-spot, we *multicast* replies to memory page requests. Multicast provides scalability and prefetching: it may service a page request from any number of children with a single response, simultaneously prefetching the page for all children that did not yet request it. Our design is based on the observation that the multicast protocol does not need to provide atomicity, ordering guarantees, or reliable delivery to prefetching receivers in order to be effective. Children operate independently and individually ensure delivery of needed pages; a single child waiting for a page does not prevent others from making progress.

Lazy state replication and multicast are implemented within the Virtual Machine Monitor (VMM) in a manner transparent to the guest OS. Our avoidance heuristics improve performance by adding VM-fork awareness to the guest. Uncooperative guests can still use VM fork, with reduced efficiency depending on application memory access patterns.

5.4 SnowFlock Implementation

SnowFlock is our implementation of the VM fork primitive. SnowFlock is implemented as a combination of modifications to the Xen 3.0.3 VMM and daemons that run in Xen's domain 0 (the administrative and privileged VM). The SnowFlock daemons form a distributed system that controls the life-cycle of VMs, by orchestrating their cloning and deallocation. SnowFlock defers policy decisions, such as resource accounting and the allocation of VMs to physical hosts, to suitable cluster management software (i.e. Usher [117] or Moab [37]) via a plug-in architecture. SnowFlock currently supports allocation management with Platform EGO [142] and Sun Grid Engine [64]. Throughout this Chapter we use a simple internal resource manager which tracks memory and CPU



Condensed VM descriptors are distributed to cluster hosts to spawn VM replicas. Memtap populates the VM replica state on demand, using multicast distribution. Avoidance heuristics reduce the number of necessary fetches.

Figure 5.3: **SnowFlock VM Replication Architecture**

allocations on each physical host.

SnowFlock uses four mechanisms to fork a VM, as illustrated in Figure 5.3. First, the parent VM is temporarily suspended to produce a *VM descriptor*: a small file that contains VM metadata and guest kernel memory management data. The VM descriptor is then distributed to other physical hosts to spawn new VMs; the entire operation is complete in sub-second time. Second, our memory-on-demand mechanism, *memtap*, lazily fetches additional VM memory state as execution proceeds. Third, the *avoidance heuristics* leverage the cooperation of the guest kernel to substantially reduce the amount of memory that needs to be fetched on demand. Finally, our multicast distribution system *mcdist* is used to deliver VM state simultaneously and efficiently, as well as to provide

- **sf_request_ticket (n, hierarchical)**: Requests an allocation for n clones. If hierarchical is true, process fork will follow VM fork, to occupy the cores of SMP cloned VMs. Returns a ticket describing an allocation for $m \leq n$ clones.
- **sf_clone(ticket)**: Clones, using the allocation in ticket. Returns the clone ID, $0 \leq ID \leq m$.
- **sf_exit()**: For children ($1 \leq ID \leq m$), terminates the child.
- **sf_join(ticket)**: For the parent ($ID = 0$), blocks until all children in the ticket reach their sf_exit call. At that point all children are terminated and the ticket is discarded.
- **sf_kill(ticket)**: Parent only, discards the ticket and immediately kills all associated children.

Table 5.1: **The SnowFlock VM Fork API**

implicit pre-fetching.

The next Section describes the SnowFlock API. We then introduce work that makes users oblivious to this API by hiding it within MPI, a well-known parallel computing middleware. While this work is not strictly part of this thesis, it is highly related to SnowFlock and introduces a way of achieving completely transparent and efficient on-the-fly application footprint scaling. We continue by describing in detail each of the four SnowFlock mechanisms. The impact of each of these mechanisms is measured in detail through microbenchmarks in Section 5.5.3. We finish this Section by discussing the specifics of the virtual I/O devices of a SnowFlock VM, namely the virtual disk and network isolation implementations.

5.4.1 API

Table 5.1 describes the SnowFlock API. VM fork in SnowFlock consists of two stages. First, the application uses **sf_request_ticket** to place a reservation for the desired number of clones. Such a reservation is obtained by interfacing with the resource management software to obtain a suitable set of machines on top of which the owner of the VM can use to run clones on top of. To optimize for common use cases in SMP hardware, VM fork can be followed by process replication: the set of cloned VMs span multiple hosts, while the processes within each VM span the physical underlying cores. This behaviour is optionally available if the **hierarchical** flag is set. Due to user quotas, current load, and other policies, the cluster management system may allocate fewer VMs than requested. In this case the application has the option to re-balance the computation to account for the smaller allocation. In the second stage, we fork the VM across the hosts provided by the cluster management system with the **sf_clone** call. When a child VM finishes its part of the computation, it executes an **sf_exit** operation which terminates the clone. A parent VM can wait for its children to terminate with **sf_join**, or force their termination with **sf_kill**.

The API calls from Table 5.1 are available to applications via a SnowFlock client library, with C and Python bindings. The client library marshals API calls and communicates them to the SnowFlock daemon running on domain 0 over the XenStore, a shared memory inter-VM messaging interface.

5.4.2 Simplifying Adoption

While the SnowFlock API is simple and flexible, it nonetheless demands modification of existing code bases. Our SnowFlock-friendly implementation of the widely used Message Passing Interface (MPI) library allows a vast corpus of unmodified parallel applications to use SnowFlock's capabilities. Based on mpich2 [16], our implementation replaces the task-launching subsystem of the library by one which invokes **sf_clone** to create the desired number of clones. Appropriately parameterized worker processes are started on each clone. Each worker uses unmodified MPI routines from then on, until the point of application termination, when the VMs are joined.

The end result of this work is providing full transparency while still realizing the benefits of cloud computing in general, and SnowFlock in particular. The user operates a single VM within which she has complete control. She uses completely unmodified applications (MPI-based ones in this case), and is never burdened with administration or any other logistic related to maintaining additional machines. However, the user can leverage, instantaneously and with great efficiency, additional hardware resources to dynamically increase the application's footprint. This is achieved by the user leveraging her unmodified applications, and never becoming directly involved with the SnowFlock API.

More details of this work can be found in our HPCVirt paper [137]. Future work plans include performing a similar adaptation of the MapReduce toolkit [42].

5.4.3 VM Descriptors

We observe that a VM suspend and resume cycle can be distilled to the minimal operations required to begin execution of a VM replica on a separate physical host. We thus modify the standard Xen suspend and resume process to yield *VM Descriptors*, condensed VM images that allow for swift clone spawning.

Construction of a VM descriptor starts by spawning a thread in the VM kernel that quiesces its I/O devices, deactivates all but one of the virtual processors (VCPUs), and issues a hypercall suspending the VM from execution. When the hypercall succeeds, a privileged process in domain 0 maps the suspended VM memory to populate the descriptor. Unlike a regular Xen suspended VM image, the descriptor only contains metadata describing the VM and its virtual devices, a few memory pages shared between the VM and the Xen hypervisor and control tools, the registers of the remaining VCPU, and the contents of the guest kernel memory management data structures.

The most important memory management structures are the page tables of the VM, which make up the bulk of a VM descriptor. In the x86 architecture each process has a separate page table, although there is a high-degree of inter-process page table sharing, particularly for kernel code and data. The cumulative size of a VM descriptor is thus loosely dependent on the number of processes the VM is executing. Additionally, a VM descriptor preserves the Global Descriptor Tables (GDT). These per-processor tables are required by the x86 segmentation hardware, and Xen relies on them to establish privilege separation between the hypervisor and the kernel of a VM.

A page table entry in a Xen paravirtualized VM contains a virtual to "machine" address translation. In Xen parlance, the machine address space is the true physical address space of the host machine, while physical frame numbers refer to the VM's notion of its own contiguous physical memory. A VM keeps an internal physical-to-machine table describing this additional level of indirection, and Xen maintains the reverse machine-to-physical table. When saving page tables to a descriptor, all the valid entries must be made host-independent, i.e. converted to VM-specific physical frame numbers. Certain values in the VCPU registers and in the pages the VM shares with Xen need to be translated as well.

The resulting descriptor is multicast to multiple physical hosts using the mcdist library we describe in Section 5.4.6, and used to spawn a VM replica on each host. The metadata is used to allocate a VM with the appropriate virtual devices and memory footprint. All state saved in the descriptor is loaded: pages shared with Xen, segment descriptors, page tables, and VCPU registers. Physical addresses in page table entries are translated to use the physical-to-machine mapping of the new host. The VM replica resumes execution by returning from its suspend hypercall, and undoing the actions of its suspend thread: enabling the extra VCPUs and reconnecting its virtual I/O devices to the new frontends.

We evaluate the VM descriptor mechanism in Section 5.5.3. In summary, the VM descriptors can be used to suspend a VM and resume a set of 32 replicas in less than a second, with an average descriptor size of roughly one MB for a VM with one GB of RAM.

5.4.4 Memory On Demand

Immediately after being instantiated from a descriptor, the VM will find it is missing state needed to proceed. In fact, the code page containing the very first instruction the VM tries to execute upon resume will be missing. SnowFlock's memory on demand subsystem, *memtap*, handles this situation by lazily populating the VM replica's memory image with state fetched from the originating host, where an immutable copy of the VM's memory from the time of cloning is kept.

Memtap is a user-space process attached to the VM replica that communicates with the hypervisor via a shared memory page and a set of event channels (akin to software interrupt lines), one per VCPU. The hypervisor detects when a missing page will be accessed for the first time by a VCPU, pauses that VCPU and notifies the memtap process with the corresponding event channel. Memtap maps the missing VM page, fetches its contents, and notifies the hypervisor to unpause the VCPU.

To allow the hypervisor to trap memory accesses to pages that have not yet been fetched, we leverage Xen shadow page tables. In shadow page table mode, the x86 register that points to the base of the current page table is replaced by a pointer to an initially empty page table. The shadow page table is filled on demand as faults on its empty entries occur, by copying entries from the real page table. No additional translations or modifications are performed to the copied page table entries; this is commonly referred to as direct paravirtual shadow mode. Shadow page table faults thus indicate that a page of memory is about to be accessed. At this point the hypervisor checks if this is the first access to a page of memory that has not yet been fetched, and, if so, notifies memtap. We also trap hypercalls by the VM kernel requesting explicit page table modifications. Paravirtual kernels have no other means of modifying their own page tables, since otherwise they would be able to address arbitrary memory outside of their sandbox.

On the parent VM, memtap implements a Copy-on-Write policy to serve the memory image to clone VMs. To preserve a copy of the memory image at the time of cloning, while still allowing the parent VM to execute, we use shadow page tables in log-dirty mode. All parent VM memory write attempts are trapped by disabling the writable bit on shadow page table entries. Upon a write fault, the hypervisor duplicates the page and patches the mapping of the memtap server process to point to the duplicate. The parent VM is then allowed to continue execution. By virtue of this COW mechanism used to retain a version of memory as expected by the clones, the total memory accounted to the parent VM grows by the number of pages duplicated by CoW. At worst, the VM's memory footprint will be X times larger, X being the number of clone generations simultaneously existing. In general, this number will be much lower (a similar behaviour was studied in more depth by the Potemkin group [198]).

Memory-on-demand is supported by a single data structure, a bitmap indicating the presence of the VM's memory pages. The bitmap is indexed by physical frame number in the contiguous address space private to a VM, and is initialized by the VM descriptor resume process by setting all bits corresponding to pages constructed from the VM descriptor. The Xen hypervisor reads the bitmap to decide if it needs to alert memtap. When receiving a new page, the memtap process sets the corresponding bit. The guest VM also uses the bitmap when avoidance heuristics are enabled. We will describe these in the next Section.

Certain paravirtual operations need the guest kernel to be aware of the memory-on-demand subsystem. When interacting with virtual I/O devices, the guest kernel hands *page grants* to domain 0. A grant authorizes domain 0 to behave as a DMA unit, by performing direct I/O on VM pages and bypassing the VM's page tables. To prevent domain 0 from reading inconsistent memory contents, we simply touch the target pages before handing the grant, thus triggering the necessary fetches.

Our implementation of memory-on-demand is SMP-safe. The shared bitmap is accessed in a lock-free manner with atomic (`test_and_set`, etc) operations. When a shadow page table write triggers a memory fetch via memtap, we pause the offending VCPU and buffer the write of the shadow page table. Another VCPU using the same page table entry will fault on the still empty shadow entry. Another VCPU using a different page table entry but pointing to the same VM-physical address will also fault on the not-yet-set bitmap entry. In both cases the additional VCPUs are paused and queued as depending on the very first fetch. When memtap notifies completion of the fetch, all pending shadow page table updates are applied, and all queued VCPUs are allowed to proceed with execution.

5.4.5 Avoidance Heuristics

While memory-on-demand guarantees correct VM execution, it may still have a prohibitive performance overhead, as page faults, network transmission, and multiple context switches between the hypervisor, the VM, and memtap are involved. We thus augmented the VM kernel with two fetch-avoidance heuristics that allow us to bypass unnecessary memory fetches, while retaining correctness.

The first heuristic intercepts pages selected by the kernel's page allocator. The kernel page allocator is invoked when a user-space process requests more memory, typically via a `malloc` call (indirectly), or when a kernel subsystem needs more memory. The semantics of these operations imply that the recipient of the selected pages does not care about the pages' previous contents. Thus, if the pages have not yet been fetched, there is no reason to do so. Accordingly, we modified the guest kernel to set the appropriate present bits in the bitmap, entirely avoiding the unnecessary memory fetches.

The second heuristic addresses the case where a virtual I/O device writes to the guest memory. Consider block I/O: the target page is typically a kernel buffer that is being recycled and whose previous contents do not need to be preserved. Again, the guest kernel can set the corresponding present bits and prevent the fetching of memory that will be immediately overwritten.

In Section 5.5.3 we show the substantial improvement that these heuristics have on the performance of SnowFlock for representative applications. In summary, the descriptors and the heuristics can reduce the amount of VM state sent by three orders of magnitude, down to under a hundred MBs when replicating a VM with 1 GB of RAM to 32 hosts.

5.4.6 Multicast Distribution

A multicast distribution system, *mcdist*, was built to efficiently provide data to all cloned virtual machines simultaneously. This multicast distribution system accomplishes two goals that are not served by point-to-point communication. First, data needed by clones will often be prefetched. Once a single clone requests a page, the response will also reach all other clones. Second, the load on the network will be greatly reduced by sending a piece of data to all VM clones with a single operation. This improves scalability of the system, as well as better allowing multiple sets of cloned VMs to co-exist.

To send data to multiple hosts simultaneously we use IP-multicast, which is commonly supported by high-end and off-the-shelf networking hardware. Multiple IP-multicast groups may exist simultaneously within a single local network and mcdist dynamically chooses a unique group in order to eliminate conflicts. Multicast clients[1] subscribe by sending an IGMP protocol message with the multicast group to local routers and switches. The switches then relay each message to a multicast IP address to all switch ports that subscribed via IGMP.

In mcdist, the server is designed to be as minimal as possible, containing only membership management and flow control logic. No ordering guarantees are given by the server and requests are processed on a first-come first-served basis. Ensuring reliability thus falls to receivers, through a timeout-based mechanism.

We highlight Frisbee [81] as a relevant reference when implementing our own multicast distribution mechanism. Frisbee aims to instantiate hundreds of disk images simultaneously, in preparation for an EmuLab [202] experiment. To our knowledge, Frisbee is the first storage substrate to aggressively employ content multicasting through receiver initiated transfers that control the data flow. Frisbee and mcdist differ in their domain-specific aspects: e.g. Frisbee uses filesystem-specific compression, not applicable

[1]In the context of this Section, the memtap processes serving children VMs are mcdist clients.

to memory state; conversely, mcdist's lockstep detection, described below, does not apply to Frisbee's disk distribution.

We begin the description of mcdist by explaining the changes to memtap necessary to support multicast distribution. We then proceed to describe two domain-specific enhancements, lockstep detection and the push mechanism, and finalize with a description of our flow control algorithm.

Memtap Modifications

Our original implementation of memtap used standard TCP networking. A single memtap process would receive only the pages that it had asked for, in exactly the order requested, with only one outstanding request (the current one) per VCPU. Multicast distribution forces memtap to account for asynchronous receipt of data, since a memory page may arrive at any time by virtue of having been requested by another VM clone. This behaviour is exacerbated in push mode, as described below.

On receipt of a page, the memtap daemon executes a hypercall that maps the target page of the VM in its address space. The cost of this hypercall can prove excessive if executed every time page contents arrive asynchronously. Furthermore, in an SMP environment, race conditions may arise when writing the contents of pages not explicitly requested: a VCPU may decide to use any of these pages without fetching them.

Consequently, in multicast mode, memtap batches all asynchronous responses until a threshold is hit, or a page that has been explicitly requested arrives and the mapping hypercall is needed regardless. To avoid races, all VCPUs of the VM are paused. Batched pages not currently mapped in the VM's physical space, or for which the corresponding bitmap entry is already set are discarded. A single hypercall is then invoked to map all remaining pages; we set our batching threshold to 1024 pages, since this is the largest number of pages mappable with a single context switch to the hypervisor. The page contents are copied, bits are set, and the VCPUs un-paused. The impact of this mechanism is evaluated in Section 5.5.3.

Lockstep Detection

Lockstep execution is a term used to describe computing systems executing the same instructions in parallel. Many clones started simultaneously exhibit a very large amount of lockstep execution. For example, shortly after cloning, VM clones generally share the same code path because there is a deterministic sequence of kernel hooks called during resumption of the suspended VM. Large numbers of identical page requests are generated at the same time.

When multiple requests for the same page are received sequentially, requests following the first are ignored, under the assumption that they are not the result of lost packets, but rather of lockstep execution. These requests will be serviced again after a sufficient amount of time, or number of requests, has passed.

Push

Push mode is a simple enhancement to the server which sends data proactively. This is done under the assumption that the memory access patterns of a VM exhibit spatial locality. Our algorithm works as follows: the server maintains a pool of counters; when a request for a page comes to the server, in addition to providing the data for that request, the server adds a counter to the pool. The new counter starts at the requested page plus one. When there is no urgent work to do, the counters in the pool are cycled through. For each counter the current page is sent, and the counter is then incremented. As pages are sent out by the server, through the counters mechanism or due to explicit requests, a corresponding bit is set in a global bitmap. No page is sent twice due to the automated counters, although anything may be explicitly requested any number of times by a client, as in pull mode. Experimentally, we found that using any sufficiently large number of counters (e.g., greater than the number of clients) provides very similar performance.

Flow Control

Sending data at the highest possible rate quickly overwhelms clients, who face a significant cost for mapping pages and writing data. A flow control mechanism was designed for the server which limits and adjusts its sending rate over time. Both server and clients estimate their send and receive rate, respectively, using a weighted average of the number of bytes transmitted each millisecond. Clients provide explicit feedback about their current rate to the server in request messages, and the server maintains an estimate of the *mean* client receive rate. The server increases its rate limit linearly, and when a loss is detected implicitly by a client request for data that has already been sent, the rate is scaled back to a fraction of the client rate estimate. Throughout our evaluation of SnowFlock, we experimentally adjusted the parameters of this algorithm. We used a rate of increase of 10 KB/s every 10 milliseconds, and we scaled back to three quarters of the estimate client rate upon detection of packet loss.

5.4.7 Virtual I/O Devices

Outside of the four techniques addressing fast and scalable VM replication, we need to provide a virtual disk for the cloned VMs, and we must guarantee the necessary network isolation between VMs cloned from different parents.

Virtual Disk

The virtual disks of SnowFlock VMs are implemented with a blocktap[200] driver. Multiple views of the virtual disk are supported by a hierarchy of copy-on-write slices located at the site where the parent VM runs. Each clone operation adds a new CoW slice, rendering the previous state of the disk immutable, and launches a disk server process that exports the view of the disk up to the point of cloning. Children access a sparse local version of the disk, with the state from the time of cloning fetched on-demand from the disk server.

Based on our experience with the memory on demand subsystem, we apply similar techniques for the virtual disk. First, we use multicast to distribute disk state to all clones and exploit similarity in access patterns. Second, we devise heuristics complementary to those used for memory. When the VM writes a chunk of data to disk, the previous contents of that chunk are not fetched. For simplicity and convenience, we chose the page size as the granularity of disk operations. No individual request to the disk subsystem will span more than a page-sized page-aligned chunk. COW operations on the parent VM, data transfer from the disk server to children, and the discard-on-write heuristic are easily and efficiently implemented in this way.

Much like memory, the disk is replicated to clones "one way". In other words, the children and master see the same state up to the point of cloning, but there is no sharing or write-through channels back to the master. If sharing is desired at the file system level, it can be trivially implemented by using a distributed file system like NFS over the virtual private network. More specialized – and convenient – mechanisms, such as a key-value store for clones to indicate completion and results of assigned work, are left for future work.

In most cases the virtual disk is not heavily exercised by a SnowFlock child VM. Most work done by the clones is processor intensive, resulting in little disk activity that does not hit kernel caches. Further, thanks to the fetch avoidance heuristic previously mentioned, writes generally do not result in fetches. Finally, we assume that data-intensive applications will in general obtain their data from shared storage provided by the physical cluster. For all these reasons, our implementation largely exceeds the demands of many realistic tasks and did not cause any noticeable overhead for the experiments in Section 5.5.

We will leave the exploration of interactions with virtual disk infrastructures such as Parallax [121] for future work. Parallax aims to support multiple VM disks while minimizing disk creation and snapshot time. Parallax's design resorts to aggressive Copy-on-Write sharing of multiple VM disk images in a cluster, and achieves this by relying on cluster-wide availability of a single (large) storage volume. may be more suitable.

Network Isolation

In order to prevent interference or eavesdropping between unrelated VMs on the shared network, either malicious or accidental, we implemented an isolation mechanism operating at the level of Ethernet packets, the primitive exposed by Xen virtual network devices. Before being sent on the shared network, the source MAC addresses of packets sent by a SnowFlock VM are rewritten as a special address which is a function of both the parent and child VM identifiers. Simple filtering rules are used by all hosts to ensure that no packets delivered to a VM come from VMs other than its parent or fellow clones. Conversely, when a packet is delivered to a SnowFlock VM, the destination MAC address is rewritten to be as expected by the VM, rendering the entire process transparent. Additionally, a small number of special rewriting rules are required for protocols with payloads containing MAC addresses, such as ARP. Despite this, the overhead imposed by filtering and rewriting is imperceptible and full compatibility at the IP level is maintained.

5.5 Evaluation

In this Section we first examine the high-level performance of SnowFlock through macro-benchmark experiments using representative application benchmarks. We describe the applications in Section 5.5.1 and the experiments in Section 5.5.2. In Section 5.5.3, we turn our attention to a detailed micro evaluation of the different aspects that contribute to SnowFlock's performance and overhead.

All of our experiments were carried out on a cluster of 32 Dell PowerEdge 1950 servers. Each machine had 4 GB of RAM, 4 Intel Xeon 3.2 GHz cores, and dual Broadcom NetXtreme II BCM5708 gigabit network adaptors. All machines were running the SnowFlock prototype based on Xen 3.0.3, with paravirtualized Linux 2.6.16.29 running as the OS for both host and guest VMs. All machines were connected to two daisy-chained Dell PowerConnect 5324 gigabit switches. Unless otherwise noted, all results reported in the following sections are the means of five or more runs, and error bars depict standard deviations. All VMs were configured with 1124 MB of RAM, which is the memory footprint needed by our most memory-intensive application (SHRiMP). All VMs had a plain file-backed root disk of 8 GBs in size.

In several experiments we compare SnowFlock's performance against a "zero-cost fork" baseline. Zero-cost results are obtained with VMs previously allocated, with no cloning or state-fetching overhead, and in an idle state, ready to process the jobs allotted to them. As the name implies, zero-cost results are overly optimistic and not representative of cloud computing environments, in which aggressive consolidation of VMs is the norm and instantiation times are far from instantaneous. The zero-cost VMs are vanilla Xen 3.0.3 domains configured identically to SnowFlock VMs in terms of kernel version, disk contents, RAM, and number of processors.

5.5.1 Applications

To evaluate the generality and performance of SnowFlock, we tested several usage scenarios involving six typical applications from bioinformatics, graphics rendering, financial services, and parallel compilation. We devised workloads for these applications with runtimes of above an hour on a single-processor machine, but which can be substantially reduced to the order of a hundred seconds if provided enough resources. Application experiments are driven by a workflow shell script that clones the VM and launches an application process properly parameterized according to the clone ID. Results are dumped to temporary files which the clones send to the parent before reaching an `sf_exit` call. Once the parent VM successfully completes an `sf_join`, the results are collated. The exception to this technique is ClustalW, where we modify the application code directly.

NCBI BLAST

The NCBI implementation of BLAST[8], the Basic Local Alignment and Search Tool, is perhaps the most popular computational tool used by biologists. BLAST searches a database of *biological sequences*, strings of characters representing DNA or proteins, to find sequences similar to a query. Similarity between two sequences is measured by an *alignment* metric, that is typically similar to edit distance. BLAST is demanding of both computational and I/O resources; gigabytes of sequence data must be read and compared with each query sequence, and parallelization can be achieved by dividing either the queries, the sequence database, or both. We experimented with a BLAST search using 1200 short protein fragments from the sea squirt *Ciona savignyi* to query a 1.5GB portion of NCBI's non-redundant protein database, a consolidated reference of protein sequences from many organisms. VM clones access the database, which is a set of plain text files, via an NFS share, which is a typical setup in bioinformatic labs. Database access is parallelized across VMs, each reading a different segment, while query processing is parallelized across process-level clones within each VM.

SHRiMP

SHRiMP [160] (SHort Read Mapping Package) is a tool for aligning large collections of very short DNA sequences ("reads") against a known genome: e.g. the human genome. This time-consuming task can be easily parallelized by dividing the collection of reads among many processors. While overall similar to BLAST, SHRiMP is designed for dealing with very short queries and very long sequences, and is more memory intensive, requiring from a hundred bytes to a kilobyte of memory for each query. In our experiments we attempted to align 1.9 million 25 letter-long reads, extracted from a *Ciona savignyi* individual using the AB SOLiD sequencing technology, to a 5.2 million letter segment of the known C. savignyi genome.

ClustalW

ClustalW [82] is a popular program for generating a *multiple alignment* of a collection of protein or DNA sequences. Like BLAST, ClustalW is offered as a web service by organizations owning large computational resources [55]. To build a guide tree, ClustalW uses progressive alignment, a greedy heuristic that significantly speeds up the multiple alignment computation, but requires precomputation of pairwise comparisons between all pairs of sequences. The pairwise comparison is computationally intensive and embarrassingly parallel, since each pair of sequences can be aligned independently.

We have modified ClustalW to allow parallelization with SnowFlock: Figure 5.4 shows the integration of API calls into the ClustalW code. After cloning, each child computes the alignment of a set of pairs statically assigned according to the clone ID. The result of each alignment is a similarity score. Simple socket code is used by the children to relay scores to the parent, which then joins the set of children. Replacing VM forking with process forking yields an equivalent parallel program confined to executing within a single machine. Using this implementation we conducted experiments performing guide-tree generation by pairwise alignment of 200 synthetic protein sequences of 1000 amino acids (characters) each.

```
sequences = ReadSequences(InputFile)
ticket = sf_request_ticket(n, hierarchical=true)
m = ticket.allocation
ID = sf_clone(ticket)
for i in sequences:
   for j in sequences[i+1:]:
      if ((i*len(sequences)+j) % m == ID):
         PairwiseAlignment(i, j)
if (ID > 0):
   RelayScores()
   sf_exit()
else:
   PairsMatrix = CollatePairwiseResults()
   sf_join(ticket)
BuildGuideTree(PairsMatrix, OutputFile)
```

Figure 5.4: ClustalW Pseudo-Code Using SnowFlock's VM Fork API

QuantLib

QuantLib [148] is an open source development toolkit widely used in quantitative finance. It provides a vast set of models for stock trading, equity option pricing, risk analysis, etc. Quantitative finance programs are typically single-program-multiple-data (SPMD): a typical task using QuantLib runs a model over a large array of parameters, e.g. stock prices, and is thus easily parallelizable by splitting the input. In our experiments we ran a set of Monte Carlo, binomial and Black-Scholes variant models to assess the risk of a set of equity options. Given a fixed maturity date, we processed 1024 equity options varying the initial and striking prices, and the volatility. The result is the set of probabilities yielded by each model to obtain the desired striking price for each option.

Aqsis – Renderman

Aqsis [15] is an open source implementation of Pixar's RenderMan interface [141], an industry standard widely used in films and television visual effects since 1989. This renderer accepts scene descriptions produced by a modeler and specified in the RenderMan Interface Bitstream (RIB) language. Rendering also belongs to the SPMD class of applications, and is thus easy to parallelize: multiple instances can each perform the same task on different frames of an animation. For our experiments we fed Aqsis a sample RIB script from the book "Advanced RenderMan: Creating CGI for Motion Pictures" [14].

Distcc

Distcc [44] is software which distributes builds of C/C++ programs over the network for parallel compilation. It operates by sending preprocessed code directly to a compiling process on each host and retrieves object file results back to the invoking host. Because the preprocessed code includes all relevant headers, it is not necessary for each compiling host to access header files or libraries; all they need is the same version of the compiler. Distcc is different from our previous benchmark in that it is not embarrassingly parallel:

actions are tightly coordinated by a master host farming out preprocessed files for compilation by workers. In our experiments we compile the Linux kernel (version 2.6.16.29) from `kernel.org`, using gcc version 4.1.2 and the default kernel optimization level (`O2`.)

5.5.2 Macro Evaluation

In our macro evaluation we aim to answer the following three questions:

- How does SnowFlock compare to other methods for instantiating VMs?
- What is the performance of SnowFlock for the representative applications described in Section 5.5.1?
- How scalable is SnowFlock? How does it perform in cloud environments with multiple applications simultaneously and repeatedly forking VMs, even under adverse VM allocation patterns?

Comparison

Table 5.2 illustrates the substantial gains SnowFlock provides in terms of efficient VM cloning and application performance. The table shows results for SHRiMP using 128 processors under three configurations: SnowFlock with all the mechanisms described in Section 5.4, and two versions of Xen's standard suspend/resume that use NFS and multicast to distribute the suspended VM image. The results show that SnowFlock significantly improves execution time with respect to traditional VM management techniques, with gains ranging between a factor of two to a factor of seven. Further, SnowFlock is two orders of magnitude better than traditional VM management techniques in terms of the amount of VM state transmitted. Despite the large memory footprint of the application (1GB), SnowFlock is capable of transmitting, in parallel, little VM state. Experiments with our other application benchmarks present similar results and are therefore not shown.

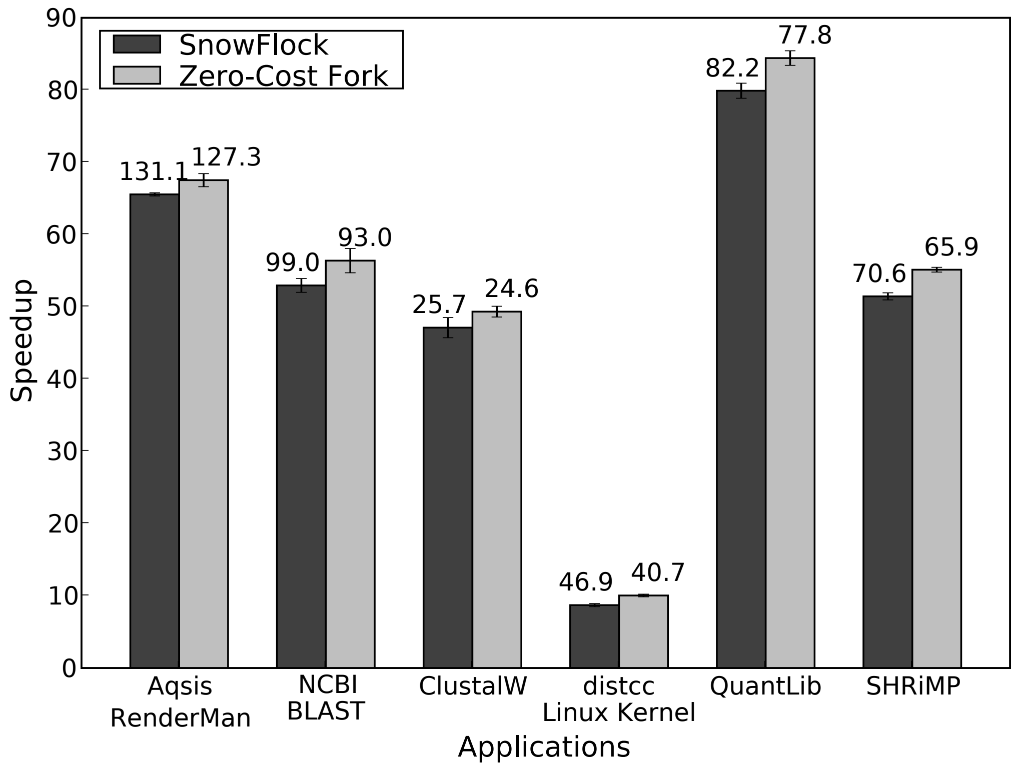
	Benchmark Time (s)	State Sent (MB)
SnowFlock	70.63 ± 0.68	41.79 ± 0.7 (multicast)
Suspend/Resume multicast	157.29 ± 0.97	1124 (multicast)
Suspend/Resume NFS	412.29 ± 11.51	$1124 * 32$ (unicast)

Table 5.2: SnowFlock vs. VM Suspend/Resume. Benchmark time and VM state sent, for SHRiMP with 128 threads.

Application Performance

We tested the performance of SnowFlock with the applications described in Section 5.5.1 and the following features enabled: memory on demand, multicast without push, avoidance heuristics, and hierarchical cloning. For each application we spawned 128 threads of execution (32 4-core SMP VMs on 32 physical hosts) in order to fully utilize our testbed. SnowFlock was tested against an ideal with 128 threads to measure overhead, and against an ideal with a single thread in order to measure speedup.

Figure 5.5 summarizes the results of our application benchmarks. We obtain speedups very close to the ideal, and total time to completion no more than five seconds (or 7%) greater than the ideal. The overheads of VM replication and on-demand state fetching are small. These results demonstrate that the execution time of a parallelizable task can be reduced, given enough resources, to an "interactive range." We note three aspects of application execution. First, ClustalW yields the best results, presenting an overhead with respect to the ideal as low as one second. Second, even though most of the applications are embarrassingly parallel, the achievable speedups are below the maximum. Some children may finish ahead of others, such that "time to completion" effectively measures the time required for the slowest VM to complete. Third, distcc's tight synchronization results in underutilized children and low speedups, although distcc still delivers a time to completion of under a minute.



Applications ran with 128 threads: 32 VMs × 4 cores. **Bars** show **speedup**, measured against a 1 VM × 1 core ideal. **Labels** indicate **time to completion** in seconds.

Figure 5.5: Application Benchmarks

Scale and Agility

We address SnowFlock's capability to support multiple concurrent forking VMs. We launched four VMs such that each simultaneously forked 32 uniprocessor VMs. To stress the system, after completing a parallel task, each parent VM joined and terminated its children and immediately launched another parallel task, repeating this cycle five times. Each parent VM executed a different application. We selected the four applications that exhibited the highest degree of parallelism (and child occupancy): SHRiMP, BLAST, QuantLib, and Aqsis. To further stress the system, we abridged the length of the cyclic parallel task so that each cycle would finish in between 20 and 35 seconds. We employed an "adversarial allocation" in which each task uses 32 processors, one per physical host, so that 128 SnowFlock VMs are active at most times, and each physical host needs to

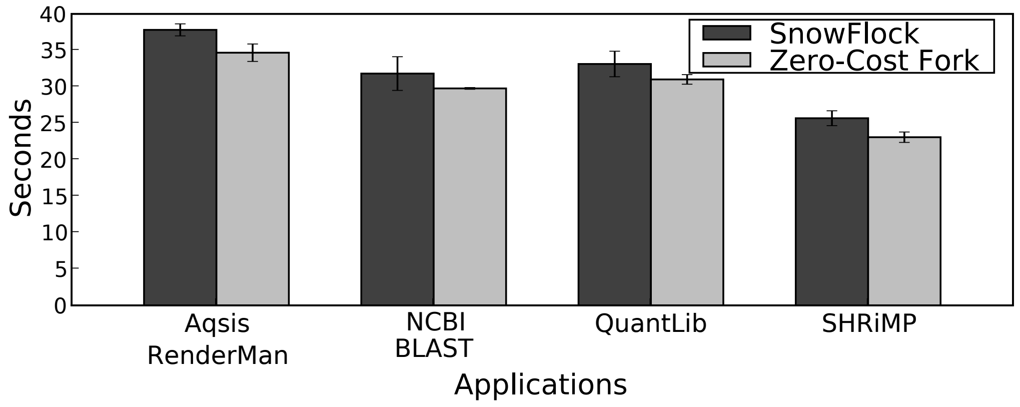


Figure 5.6: Concurrent Execution of Multiple forking VMs. For each task we repeatedly cycled cloning, processing and joining.

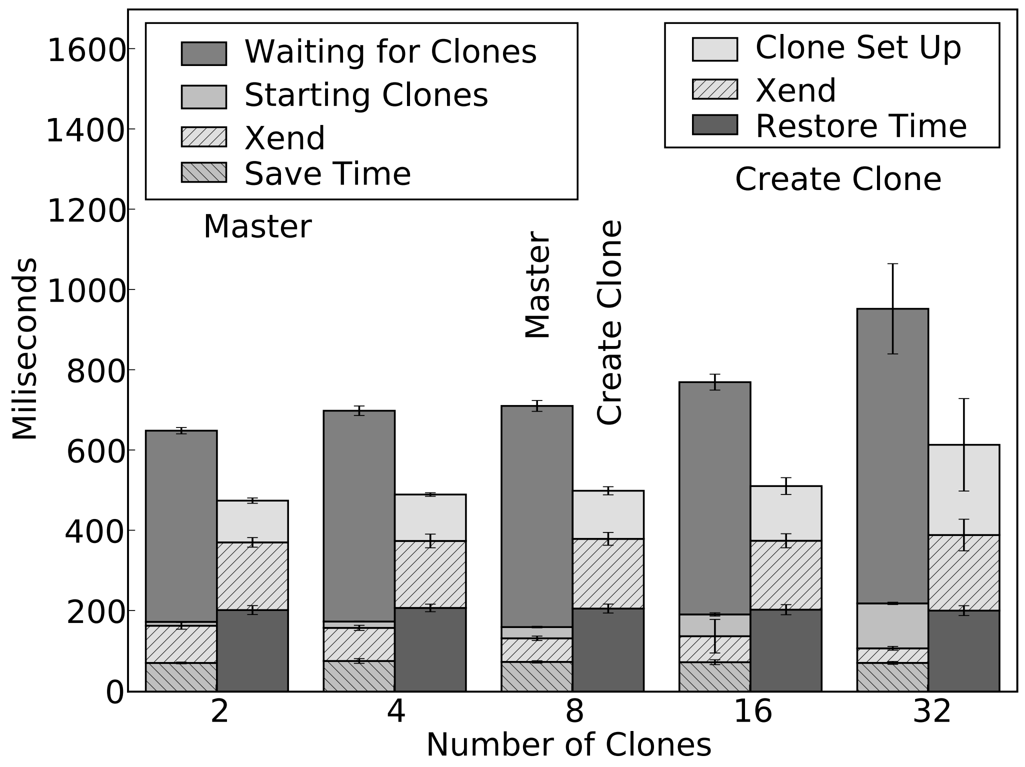
fetch state from four parent VMs. The zero-cost results were obtained with an identical distribution of VMs. Since there is no state-fetching performed in the zero-cost case, the actual allocation of VMs does not affect those results.

The results, shown in Figure 5.6, demonstrate that SnowFlock is capable of withstanding the increased demands of multiple concurrent forking VMs. As we will show in Section 5.5.3, this is mainly due to the small overall number of memory pages sent by the multicast server, when pushing is disabled and heuristics are enabled. The introduction of multiple concurrently forking VMs causes no significant increase in overhead, although outliers with higher time to completion are seen, resulting in wider error bars. These outliers are caused by occasional congestion when receiving simultaneous bursts of VM state for more than one VM; we believe optimizing mcdist will yield more consistent running times.

5.5.3 Micro Evaluation

To better understand the behaviour of SnowFlock, and to isolate the factors which cause the small overhead seen in the previous Section, we performed experiments to answer several performance questions:

- How fast does SnowFlock clone a VM? How scalable is the cloning operation?



"Master" is the global view, while "Create Clone" is the average VM resume time for the n clones. Legend order matches bar stacking from top to bottom.

Figure 5.7: Time To Create Clones

- What are the sources of overhead when SnowFlock fetches memory on demand to a VM?
- How sensitive is the performance of SnowFlock to the use of avoidance heuristics and the choice of networking strategy?

Fast VM Replica Spawning

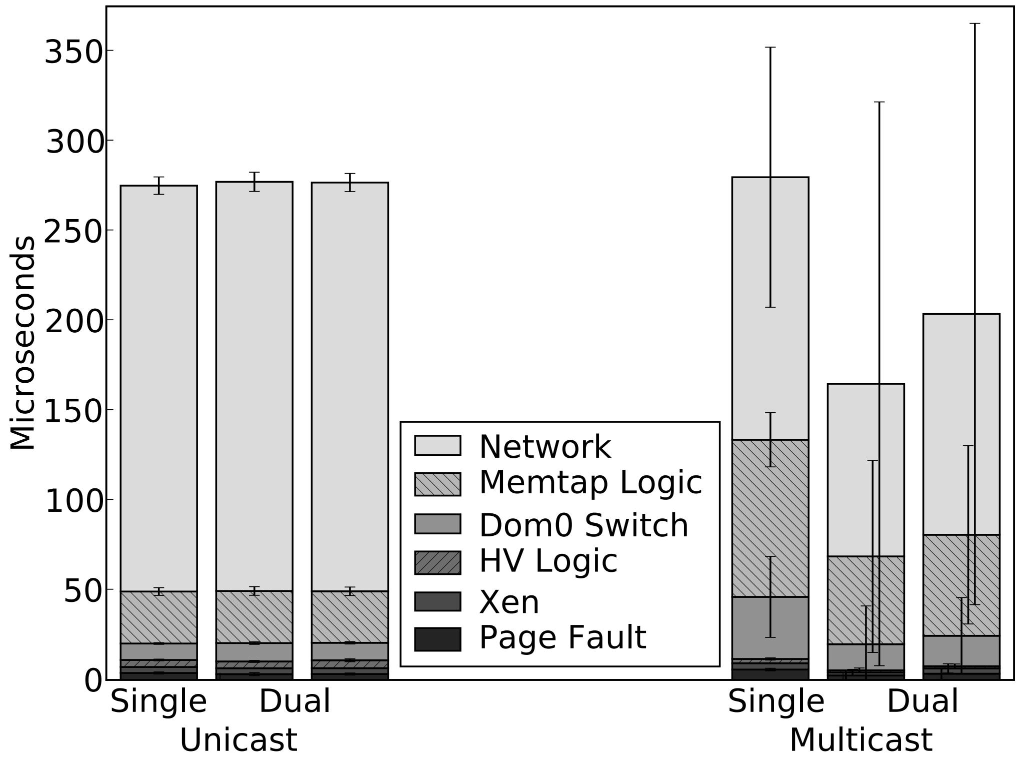
Figure 5.7 shows the time spent replicating a single-processor VM to n VM clones, with each new VM spawned on a different physical host. For each size n we present two bars: the "Master" bar shows the global view of the operation, while the "Create Clone" bar shows the average time spent by all VM resume operations on the n physical hosts. The average size of a VM descriptor for these experiments was 1051 $\pm$ 7 KB.

Recall that in Section 5.4, the process used to replicate a VM was introduced. For the purposes of evaluating the time required for VM cloning, we decompose this process into the following steps corresponding to bars in Figure 5.7: 1) Suspending the running VM and generating a VM descriptor. These are the “Save Time” and “Xend” components in the “Master” bar, with “Xend” standing for unmodified Xen code we leverage; 2) contacting all target physical hosts to trigger VM instantiation (“Starting Clones”); 3) each target host pulls the VM descriptor via multicast (“Clone Set up” in the “Create Clone” bar); 4) spawning each clone VM from the descriptor (“Restore Time” and “Xend”); and 5) waiting to receive notification from all target hosts (“Waiting for Clones”, which roughly corresponds to the total size of the corresponding “Create Clone” bar).

From these results we observe the following: first, VM replication is an inexpensive operation ranging in general from 600 to 800 milliseconds; second, VM replication time is largely independent of the number of replicas being created. Larger numbers of replicas introduce, however, a wider variance in the total time to fulfill the operation. The variance is typically seen in the time to multicast each VM descriptor, and is due in part to a higher likelihood that on some host a scheduling or I/O hiccup might delay the VM resume for longer than the average.

Memory On Demand

To understand the overhead involved in our memory-on-demand subsystem, we devised a microbenchmark in which a VM allocates and fills in a number of memory pages, invokes SnowFlock to have itself replicated and then touches (by reading and writing the first byte) each page in the allocated set. We instrumented the microbenchmark, the Xen hypervisor and memtap to timestamp events during the fetching of each page. The results for multiple microbenchmark runs totalling ten thousand page fetches are displayed in Figure 5.8. The “Single” columns depict the result when a single VM fetches state. The “Dual” columns show the results when two VM clones concurrently fetch state.



Components involved in a SnowFlock page fetch. Legend order matches bar stacking from top to bottom. “Single” bars for a single clone fetching state, “Dual” bars for two clones concurrently fetching state.

Figure 5.8: Page Fault Time

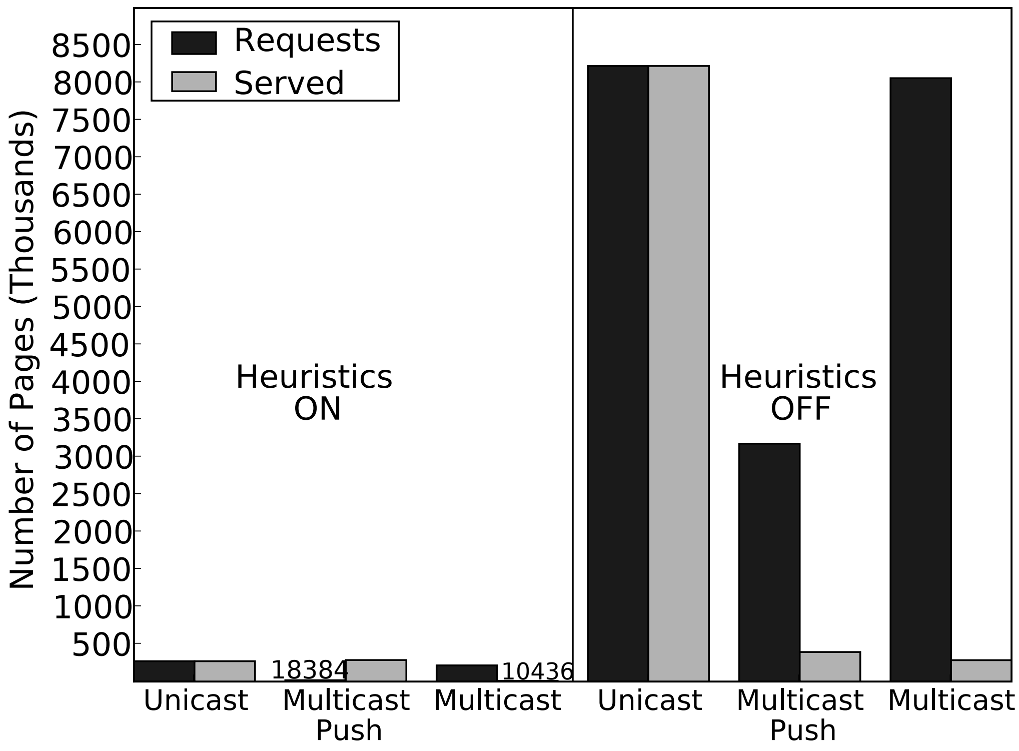
We split a page fetch operation into six components. “Page Fault” indicates the hardware overhead of using the shadow page tables to detect first access to a page after VM resume. “Xen” is the cost of executing the Xen hypervisor shadow page table logic. “HV Logic” is the time consumed by our logic within the hypervisor (bitmap checking and SMP safety.) “Dom0 Switch” is the time spent while context switching to the domain 0 memtap process, while “Memtap Logic” is the time spent by the memtap internals, consisting mainly of mapping the faulting VM page. Finally, “Network” depicts the software and hardware overheads of remotely fetching the page contents over gigabit Ethernet.

The overhead of page fetching is modest, averaging 275 μs with unicast (standard TCP). Our implementation is frugal, and the bulk of the time is spent in the networking stack. With multicast, substantially higher variances are observed in three components: network, memtap logic, and domain 0 switch. As explained in Section 5.4.6, memtap fully pauses the VM and batches multiple page mappings in a single hypercall with multicast, making the average operation more costly. Also, mcdist's logic and flow control are not as optimized as TCP's, and run in user space. The need to perform several system calls results in a high scheduling variability. A final contributor to multicast's variability is the effectively bimodal behaviour caused by implicit prefetching. Sometimes, the page the VM needs may already be present, in which case the logic defaults to a number of simple checks. This is far more evident in the dual case, in which requests by one VM result in prefetching for the other, and explains the high variance and lower overall "Network" averages for the multicast case.

Sensitivity Analysis

In this Section we perform a sensitivity analysis on the performance of SnowFlock, and measure the benefits of the heuristics and multicast distribution. Throughout these experiments we employed SHRiMP as the driving application. The experiment spawns n uniprocessor VM clones, and on each clone it executes a set of reads of the same size against the same genome. N clones perform n times the amount of work as a single VM, and should complete in the same amount of time, yielding an n-fold throughput improvement. We tested twelve experimental combinations by: enabling or disabling the avoidance heuristics; increasing SHRiMP's memory footprint by doubling the number of reads from roughly 512 MB (167116 reads) to roughly 1 GB (334232 reads); and varying the choice of networking substrate between unicast, multicast, and multicast with push.

Figure 5.9 illustrates the memory-on-demand activity for a memory footprint of 1 GB and 32 VM clones. Consistent results were observed for all experiments with smaller numbers of clones, and with the 512 MB footprint. The immediate observation is the substantially beneficial effect of the avoidance heuristics. Nearly all of SHRiMP's memory

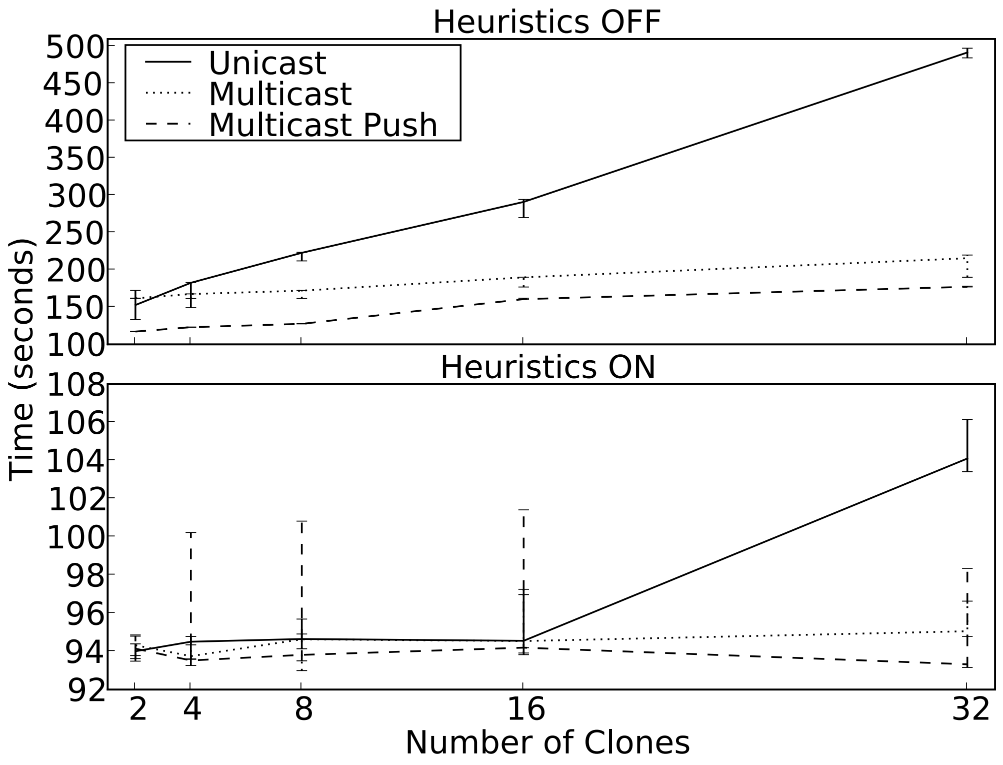


Comparison of aggregate page requests from 32 clones vs. number of pages sent by the memory server.

Figure 5.9: Pages Requested vs. Served

footprint is allocated from scratch when the reads are loaded. The absence of heuristics forces the VMs to request pages they do not really need, inflating the number of requests from all VMs by an order of magnitude. Avoidance heuristics have a major impact in terms of network utilization, as well as in enhancing the scalability of the experiments and the system as a whole. The lower portion of Figure 5.10 shows that the decreased network utilization allows the experiments to scale gracefully, such that even unicast is able to provide good performance, up to 16 clones.

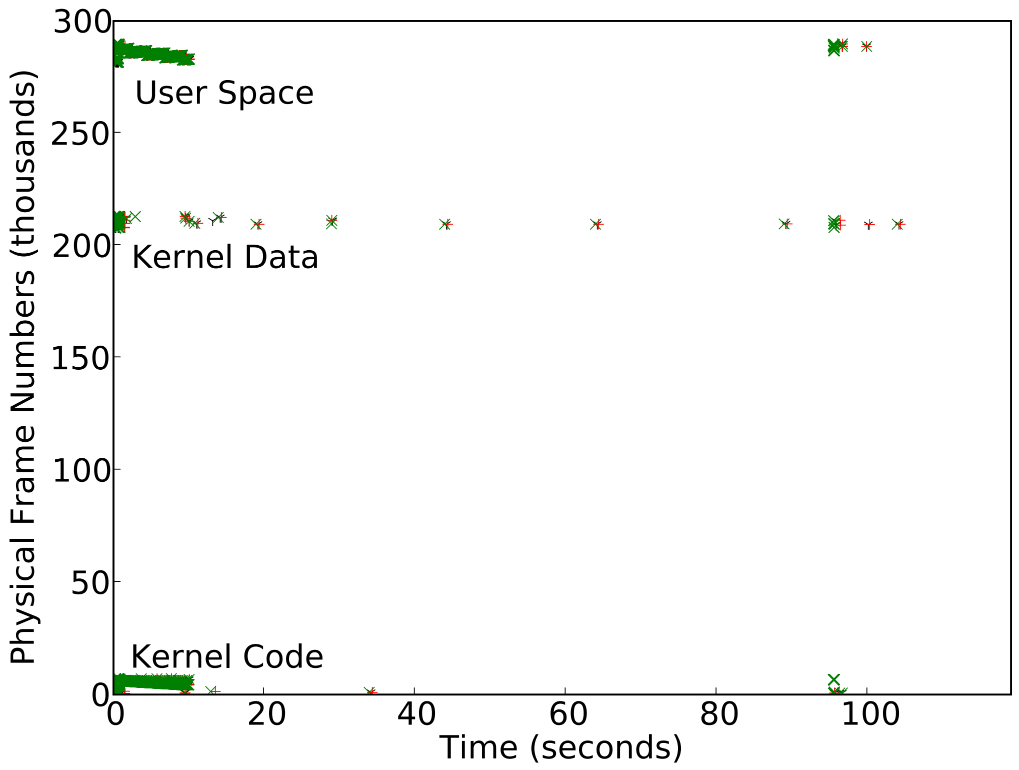
Three aspects of multicast execution are reflected in Figure 5.9. First, even under the extreme pressure of disabled heuristics, the number of pages served is reduced dramatically. This enables a far more graceful scaling than that achieved with unicast, as seen in the upper portion of Figure 5.10. Second, lockstep avoidance works effectively: lock-



Uniprocessor VMs, n clones perform n times the work of 1 VM. Points are medians, error bars show min and max.

Figure 5.10: Time to Completion

step executing VMs issue simultaneous requests that are satisfied by a single response from the server. Hence, a difference of an order of magnitude is evident between the "Requests" and "Served" bars in three of the four multicast experiments. Third, push mode increases the chances of successful prefetching and decreases the overall number of requests from all VMs, at the cost of sending more pages. The taller error bars in Figure 5.10 (up to 8 seconds) of the push experiments, however, reveal the instability caused by the aggressive distribution. Constantly receiving pages keeps memtap busy and not always capable of responding quickly to a page request from the VM, adversely affecting the runtime slightly and causing the VMs to lag behind. Further, multicast with push aggressively transmits the full set of pages comprising the VM memory state, mostly negating the opportunities for network utilization savings that on-demand propagation,

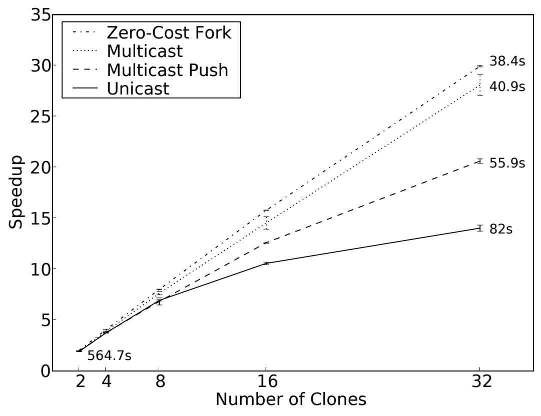


Page requests over time for three randomly selected VMs out of 32 clones, using multicast with avoidance heuristics enabled.

Figure 5.11: Pages Requested vs. Time

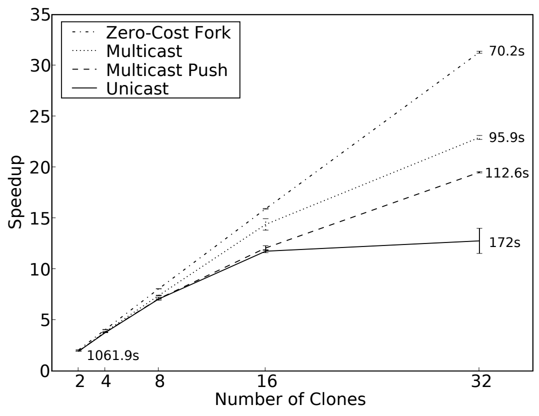
coupled with our avoidance heuristics, creates.

Figure 5.11 shows the requests of three randomly selected VMs as time progresses during the execution of SHRiMP. Avoidance heuristics are enabled, the footprint is 1 GB, and there are 32 VM clones using multicast. Page requests cluster around three areas: kernel code, kernel data, and previously allocated user space code and data. Immediately upon resume, kernel code starts executing, causing the majority of the faults. Faults for pages mapped in user-space are triggered by other processes in the system waking up, and by SHRiMP forcing the eviction of pages to fulfill its memory needs. Some pages selected for this purpose cannot be simply tossed away, triggering fetches of their previous contents. Due to the avoidance heuristics, SHRiMP itself does not directly cause any fetches, and takes over the middle band of physical addresses for the duration of the



Speedup against a single thread, with labels showing absolute times.

(a) Speedup – 256 MB DB



Speedup against a single thread, with labels showing absolute times.

(b) Speedup – 512 MB DB

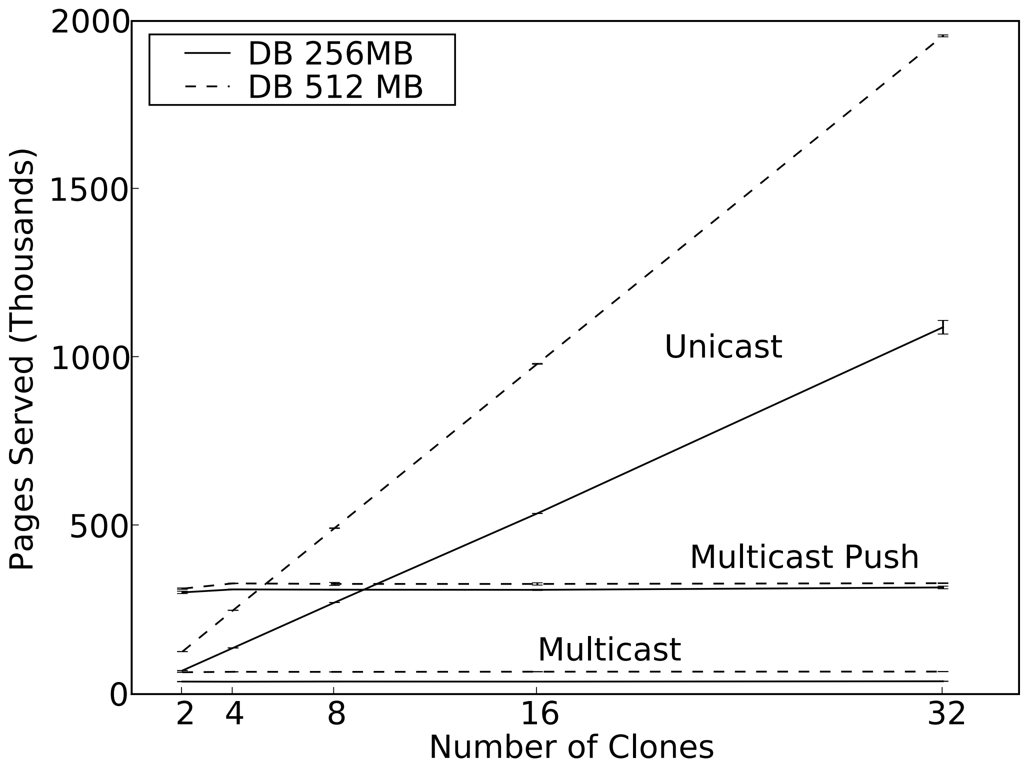
Figure 5.12: Sensitivity Analysis with NCBI BLAST

experiment. Fetches are only performed rarely when kernel threads need data.

Heuristic-adverse Experiment

While the results with SHRiMP are highly encouraging, this application is not representative of workloads in which an important portion of memory state is needed after cloning. In this scenario, the heuristics are unable to ameliorate the load on the memory-on-demand subsystem. To synthesize such behaviour we ran a similar sensitivity analysis using BLAST. We aligned 2360 queries split among an increasing number of n VM clones. The queries were run against a portion of the NCBI genome DB cached in main memory before cloning the VMs. With avoidance heuristics enabled, we varied the networking substrate and the size of the cached DB, from roughly 256 MB to roughly 512 MB.

Figure 5.12 (a) plots the speedups achieved for a DB of 256 MB by cloning a uniprocessor VM to n replicas. Speedups are against a zero-cost ideal with a single VM, and the labels indicate the runtime for 32 VMs. We see an almost linear speedup for multicast, closely tracking the speedup exhibited with ideal execution, while unicast ceases to scale after 16 clones. Multicast push shows better performance than unicast but is unable



Aggregate number of pages sent by the memory server to all clones.

Figure 5.13: Memory Pages Served

to perform as well as multicast, due to memtap and network congestion. With a larger database (Figure 5.12 (b)) even multicast starts to suffer a noticeable overhead, hinting at the limits of the current implementation.

Figure 5.13 shows the number of pages fetched vs. the number of clones for the three networking substrates. The information is consistent with that presented in Figure 5.9. The linear scaling of network utilization by unicast leads directly to poor performance. Multicast is not only the better performing, but also the one imposing the least load on the network, allowing for better co-existence with other sets of SnowFlock cloned VMs. These observations, coupled with the instabilities of push mode shown in the previous Section, led us to choose multicast with no push as SnowFlock's default behaviour for the macro-evaluation in Section 5.5.2.

5.6 Summary

In this Chapter, we introduced the primitive of VM fork and SnowFlock, our Xen-based implementation. Matching the well-understood semantics of stateful worker creation, VM fork provides cloud users and programmers the capacity to instantiate dozens of VMs in different hosts in sub-second time, with little runtime overhead, and frugal use of cloud IO resources. VM fork thus enables the simple implementation and deployment of services based on familiar programming patterns that rely on fork's ability to quickly instantiate stateful workers. While our evaluation focuses on interactive parallel Internet services, SnowFlock has broad applicability to other applications, such as flash crowd handling, execution of untrusted code components, instantaneous testing, etc.

SnowFlock makes use of three key observations. First, that it is possible to drastically reduce the time it takes to clone a VM by copying only the critical state, and fetching the VM's memory image efficiently on-demand. Second, that simple modifications to the guest kernel significantly reduce network traffic by eliminating the transfer of pages that will be overwritten. For our applications, these optimizations can drastically reduce the communication cost for forking a VM to a mere 40 MBs for application footprints of several GBs. Third, the locality of memory accesses across cloned VMs makes it beneficial to distribute VM state using multicast. This allows for the instantiation of a large number of VMs at a (low) cost similar to that of forking a single copy.

In closing, SnowFlock lowers the barrier of entry to cloud computing and opens the door for new cloud applications. VM fork provides a well-understood programming interface with substantial performance improvements; it removes the obstacle of long VM instantiation latencies, and greatly simplifies management of application state. SnowFlock thus places in the hands of users the full potential of the cloud model by simplifying the programming of applications that dynamically change their execution footprint. In particular, SnowFlock is of immediate relevance to users wishing to test and deploy parallel applications in the cloud.

5.6.1 Future directions for SnowFlock

SnowFlock is an active open-source project [189]. Our future work plans involve adapting SnowFlock to big-data applications. We believe there is fertile research ground studying the interactions of VM fork with data parallel APIs such as MapReduce [42]. For example, SnowFlock's transient clones cannot be entrusted with replicating and caching data due to their ephemeral natures. Allowing data replication to be handled by hosts, as opposed to the VMs, endows the VMs with big-data file system agnosticism. In other words, it allows the VMs to remain oblivious to how the big-data replication and distribution policies are actually implemented by the underlying storage stack.

SnowFlock is promising as a potential transforming force in the way clusters or clouds are managed today. A building block of cloud computing is task consolidation, or resource over-subscription, as a means to exploit statistical multiplexing. The underlying rationale for consolidation is that more resources than actually available can be offered. Since simultaneous peak consumption of allotted resources by all tasks happens very rarely, if ever, the overall capacity of the cloud will not be exceeded. Scale flexibility at the degree offered by SnowFlock is very hard to achieve in clouds using consolidation principles. Currently, consolidation of idle VMs is performed primarily by suspending them to secondary storage, or by live-migrating them to highly over-subscribed hosts where time sharing of the processor by idle tasks is not an issue [171, 205]. However, when an idle VM transitions to a stage of activity, sharing due to consolidation becomes harmful to performance. Resuming and or live-migrating a consolidated VM to a host where it can execute with no sharing is costly. These methods are plagued by the same problems as booting from images kept in backend storage: high latency, and lack of scalability, due to an abuse of the network interconnect by pushing GBs of VM state.

Beyond its appealing computing model, the performance characteristics of SnowFlock make it an attractive choice to rethink the current consolidation-based model, and effectively do away with the need for consolidation in many cases. SnowFlock is able to instantiate VMs at a very frugal cost, in terms of time and I/O involved for state transmission. Thus, there is no need to keep idle VMs lurking around the cloud, unless they accomplish a specific task; new generic computing elements can be instantiated quickly and inexpensively. In an ideal SnowFlock cloud, there are no idle machines, since a VM is destroyed after it finishes a compute task, its resources are reclaimed and can be immediately assigned to a new VM. However, a resource of interest that the VM may have accrued throughout its execution is the contents of its buffer cache. These contents can survive VM destruction if administered in collaboration with the host OS. This mechanism is in consonance with the file system agnosticism proposed previously, and is being currently studied by two groups [50, 116].

SnowFlock's objective is performance rather than reliability. While memory-on-demand provides significant performance gains, it imposes a long-lived dependency on a single source of VM state. Another aspect of our future work involves studying how to push VM state in the background to achieve a stronger failure model, without sacrificing the speed of cloning or low runtime overhead.

We view the use of high-speed interconnects, if available, as a viable alternative to multicasting. Huang et al. [84] demonstrate efficient point-to-point VM migration times with RDMA on Infiniband. However, we note that the latency of this method increases linearly when a single server pushes a 256 MB VM to multiple receivers, which is a crucial operation to optimize for use in SnowFlock.

Finally, we wish to explore applying the SnowFlock techniques to wide-area VM migration. This would allow, for example, "just-in-time" cloning of VMs over geographical distances to opportunistically exploit cloud resources. We foresee modifications to memory-on-demand to batch multiple pages on each update, replacement of IP-multicast, and use of content-addressable storage at the destination sites to obtain local copies of frequently used state (e.g. libc).

Chapter 6

Conclusion

In this thesis we have demonstrated that virtualization can be used to relocate tasks, and to dynamically scale the footprint of processing-intensive tasks in a public cloud. These are two fundamental building blocks of a vision of computing that aims to balance the benefits of time-sharing and personal computing. In our vision, users are endowed with specialized rich client devices – portables with graphics hardware – that provide seamless and crisp interactive performance, while applications, configuration and data are pervasively available from centralized well-maintained repositories operating as server farms residing in the network core, or cloud. The building blocks we have demonstrated provide system support for applications to flexibly relocate and resize according to runtime needs. We define this ability as *flexible computing*.

Our methods leverage and exploit a fundamental property of system virtualization: its ability to encapsulate a complete software stack (OS included) and serialize it. A serialized VM only demands three pre-conditions to execute on any piece of hardware: (i) a matching binary interface (ABI), (ii) network access to keep alive open connections, and (iii) matching underlying virtual machine monitor (VMM) software. Because of the vast adoption of the x86 processor architecture, and the TCP/IP protocol stack that provides robust connection-based semantics (with the help of tunnels, VPNs, MobileIP and other technologies), these constraints are not restrictive.

These assertions regarding the ease of deployment of serialized VM images are not absolutely clear cut and have minor caveats, such as the minute but binary-incompatible differences in the instruction set extensions supported by different x86 models and vendors. Setting those caveats aside for a moment allows us to indulge in an exercise in reductionism: portraying the contributions of this thesis as ways of manipulating and propagating serialized VM state. Location flexibility then becomes a version of `mv`, and scale flexibility becomes a (parallel) version of `cp`, operating efficiently on serialized VM images. This overly reductionist point of view integrates under the same framework differential transfer, lazy or on-demand propagation, post-copy, multicast distribution, and heuristics to prune the amount of state propagated. In other words, it integrates the specific techniques developed or used in many parts of this work, and techniques that could be added in future work, such as compression, binary diffing, state recreation via replay, and content-addressable storage. This framework also relates the work done here with many other recent projects that essentially focus on ways to manipulate VM state and study derived applications: Remus [40], Potemkin [198], Difference Engine [75], etc.

The insights presented in this work not only constitute important building blocks of a vision, but also lay out a number of future work directions. Security was singled out in the introduction. We also pointed at a trend towards software specialization matching the underlying hardware specialization. For example, a general purpose OS may become overkill for a device that basically operates as a GUI engine pulling data from the cloud. Such a device might be perfectly served by a web browser architecture with a TCP/IP stack and a javascript engine. Whilst exaggerated, this simplification points out the specialization of roles percolating upwards into the software realm.

Role and device specialization synergizes with an increased adoption of interpreted languages. Interfaces such as Google Gears [63], the Dojo toolkit [45], or Microsoft's Azure cloud [122] drive forward a re-formulation of applications into a model that cleanly accommodates the separation of roles promoted by this thesis: GUI frontends and data backends. Conveniently, these languages provide increased security and reliability by design, preventing buffer overflows and heap mis-management, and virtualize the underlying code execution interface by relying on interpreted bytecode (with potential performance optimizations via JIT-compilation). The latter is particularly powerful in that it removes limitations on VM-based techniques deriving from ABI incompatibility. In other words, it allows relocation and scaling of computation transparently across ARM (portables) and Intel (server) devices.

We seem thus, to have come full circle. In this thesis we have used system virtual machines to flexibly and dynamically scale and relocate applications. Future-telling is a proven way of making mistakes. The jury is out, thus, on whether VM-based techniques will dominate the computing landscape, or will be confined to the support of legacy applications. Will most applications be written in javascript and .NET? Will Intel aggressively move the x86 architecture into the small device space? Regardless, we have shown that virtualization holds appropriate solutions for a world in which applications seamlessly relocate between the edges and the core, and efficiently scale their computing footprint when executing in cloud server farms.

Appendix: Attribution

Except for a few exceptions, the totality of the work and experiments presented in this thesis was implemented and conducted by the author. Said exceptions are a portion of the VNC-Redux prototype, implemented by Niraj Tolia; the majority of the mcdist prototype, implemented by Adin Scannell; a portion of the network filtering subsystem of SnowFlock, implemented by Stephen Rumble; a first cut of the SnowFlock control stack and user-facing API, implemented by Joe Whitney; and the completion of the SnowFlock C user-facing API, carried out by Philip Patchin.

Bibliography

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., and M. W. Young. Mach: A New Kernel Foundation for Unix Development. In *Proc. of the Summer USENIX Conference*, pages 93–112, Atlanta, GA, 1986.

[2] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, San Jose, CA, October 2006.

[3] Advanced Micro Devices. Introducing AMD Virtualization. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_14287,00.html`.

[4] Advanced Micro Devices. Live Migration with AMD-V Extended Migration Technology. `http://developer.amd.com/assets/Live%20Virtual%20Machine%20Migration%20on%20AMD%20processors.pdf`.

[5] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital Dynamic Telepathology - The Virtual Microscope. In *Proc. American Medical Informatics Association (AMIA) Annual Fall Symposium*, Lake Buena Vista, FL, November 1998.

[6] J. M. Airey, J. H. Rohlf, and Jr. F. P. Brooks. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proc.*

SI3D '90: Symposium on Interactive 3D Graphics, pages 41–50, Snowbird, UT, 1990.

[7] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers. In *Proc. ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, November 2003.

[8] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1997.

[9] G. A. Alvarez, E. Borowsky, S. Go, T. R. Romer, R. B. Becker Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[10] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/gp/browse.html?node=201590011`.

[11] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proc. of the USENIX Annual Technical Conference*, pages 307–322, San Diego, CA, June 2000.

[12] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proc. Conference on Virtual Execution Environments (VEE)*, pages 44–54, San Diego, CA, June 2007.

[13] AnandTech. Next-Generation Game Performance with the Unreal Engine: 15-way GPU Shootout. `http://www.anandtech.com/video/showdoc.html?i=1580&p=1`.

[14] A. A. Apodaka and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Academic Press, 2000.

[15] Aqsis Renderer – Freedom to Dream. `http://aqsis.org/`.

[16] Argonne National Laboratory. MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[17] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, September 1989.

[18] Autodesk. Maya. `http://www.autodesk.com/maya`.

[19] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System - Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1993.

[20] R. Baratto, S. Potter, G. Su, and J. Nieh. MobiDesk: Virtual Desktop Computing. In *Proc. 10th Annual ACM International Conference on Mobile Computing and Networking*, pages 1–15, Philadelphia, PA, September 2004.

[21] 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 *Proc. 17th Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[22] G. Baron, C. Sarris, and E. Fiume. Fast and Accurate Time-domain Simulations With Commodity Graphics Hardware. In *Proc. Antennas and Propagation Society International Symposium*, pages 193–196, Washington, DC, July 2005.

[23] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, pages 47–60, Anaheim, CA, April 2005.

[24] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn. The Price of Safety: Evaluating IOMMU Performance. In *Ottawa Linux Symposium*, pages 9–20, Ottawa, Canada, June 2007.

[25] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Proc. ACM/IEEE Conference on Supercomputing*, Dallas, TX, November 2000.

[26] Peter J. Braam. The lustre storage architecture, November 2002. `http://www.lustre.org/docs/lustre.pdf`.

[27] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *Proc.* 12th *Symposium on Operating Systems Principles (SOSP)*, pages 1–11, Copper Mountain, CO, December 1995.

[28] I. Buck, G. Humphreys, and P. Hanrahan. Tracking Graphics State for Networked Rendering. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Interlaken, Switzerland, August 2000.

[29] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proc.* 16th *Symposium on Operating Systems Principles (SOSP)*, pages 143–156, Saint Malo, France, October 1997.

[30] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with Portable SoulPads. In *Proc. 3rd International Conference on Mobile Systems Applications and Services (MobiSys)*, pages 65–78, Seattle, WA, June 2005.

[31] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[32] R. Chandra, C. Sapuntzakis N. Zeldovich, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2005.

[33] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 90–103, Washington, DC, 2003.

[34] P. M. Chen and B. D. Noble. When Virtual is Better Than Real. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.

[35] C. Clark, K. Fraser, S. Hand, J. Gorm Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[36] J. Clarke, M. Kazar, S. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[37] Cluster Resources Inc. Moab Cluster Suite. `http://www.clusterresources.com/pages/products/moab-cluster-suite.php`.

[38] CodeWeavers. WineD3D. `http://wiki.winehq.org/WineD3D`.

[39] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal on resources for Developers*, 25(5):483–490, 1981.

[40] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. 5th Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, April 2008.

[41] K. Czajkowski, M. Thiebaux, and C. Kesselman. Practical Resource Management for Grid-based Visual Exploration. In *Proc. IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 416–426, San Francisco, CA, August 2001.

[42] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, CA, December 2004.

[43] J. Dike. *User Mode Linux*. Prentice Hall, 2006. `http://user-mode-linux.sourceforge.net/`.

[44] distcc: a Fast, Free Distributed C/C++ Compiler. `http://distcc.samba.org/`.

[45] Dojo, the Javascript Toolkit. `http://www.dojotoolkit.org/`.

[46] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, 21(8):1–27, August 1991.

[47] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. In *Proc. Workshop on I/O Virtualization (WIOV)*, San Diego, CA, December 2008.

[48] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. In

Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 211–224, Boston, MA, December 2002.

[49] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 63–72, Santa FE, NM, 1988.

[50] M. Abd el Malek, M. Wachs, J. Cipar, K. Sanghi, G. R. Ganger, G. A. Gibson, and M. K. Reiter. File System Virtual Appliances: Portable File System Implementations. Technical Report CMU-PDL-09-102, Carnegie Mellon University, May 2009.

[51] W. Emeneker and D. Stanzione. Dynamic Virtual Clustering. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*, pages 84–90, Austin, TX, September 2007.

[52] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. 15th Symposium on Operating System Principles (SOSP)*, pages 251–266, Copper Mountain, CO, December 1995.

[53] Enomalism Virtualized Management Dashboard. `http://www.enomalism.com/`.

[54] Epic Games. Unreal Tournament. `http://www.unrealtournament.com/`.

[55] European Bioinformatics Institute - ClustalW2. `http://www.ebi.ac.uk/Tools/clustalw2/index.html`.

[56] Fedora Project. Rendering project/AIGLX. `http://fedoraproject.org/wiki/RenderingProject/aiglx`.

[57] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A Case For Grid Computing On Virtual Machines. In *Proc. 23*$^{\mathrm{rd}}$ *International Conference on Distributed Computing Systems (ICDCS)*, pages 550–559, Providence, RI, 2003.

[58] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, and X. Zhang. Virtual Clusters for Grid Communities. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Singapore, May 2006.

[59] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, February 1997.

[60] T. Funkhouser and C. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In *Proc. 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 247–254, Anaheim, CA, August 1993.

[61] S. W. Galley. PDP-10 Virtual Machines. In *Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, MA, 1969.

[62] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Proc. ACM/IEEE Conference on Supercomputing*, pages 606–607, Seattle, WA, November 2005.

[63] Google Gears – Improving your Web Browser. `http://gears.google.com/`.

[64] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Brisbane, Australia, May 2001.

[65] Ashvin Goel. Utterance, 2006.

[66] GoGrid Cloud Hosting. http://www.gogrid.com/.

[67] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6), June 1974.

[68] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Proc. 5th International Conference Vector and Parallel Processing (VECPAR)*, Porto, Portugal, 2002.

[69] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing, Boston, MA, 1996.

[70] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. In *Proc. 17th Symposium on Operating Systems Principles (SOSP)*, pages 154–169, Kiawah Island, SC, December 1999.

[71] GPGPU – General Purpose Programming on GPUs. What programming API's exist for GPGPU. http://www.gpgpu.org/w/index.php/FAQ#What_programming_APIs_exist_for_GPGPU.3F.

[72] Tungsten Graphics. Gallium3d. http://www.tungstengraphics.com/wiki/index.php/Gallium3D.

[73] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22(4):421–486, November 2004.

[74] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March 2006.

[75] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machine. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 309–322, San Diego, CA, December 2008.

[76] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM:GPU-accelerated Virtual Machines. In *Proc. Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, Nüremberg, Germany, April 2009.

[77] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *Proc. 10th Workshop on Hot Topic in Operating Systems (HotOS)*, Santa Fe, NM, June 2005.

[78] J. Gorm Hansen. Blink: Advanced Display Multiplexing for Virtualized Applications. In *Proc. 17th International workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2007.

[79] G. Heiser, V. Uhlig, and J. LeVasseur. Are Virtual Machine Monitors Microkernels Done Right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, January 2006.

[80] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *Proc. USENIX Annual Technical Conference*, pages 113–128, Boston, MA, June 2008.

[81] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, Scalable Disk Imaging with Frisbee. In *Proc. of the USENIX Annual Technical Conference*, pages 283–296, San Antonio, TX, June 2003.

[82] D. Higgins, J. Thompson, and T. Gibson. CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Research*, 22(22):4673–4680, November 1994.

[83] M. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. Conference on Virtual Execution Environments (VEE)*, pages 51–60, Washington, DC, March 2009.

[84] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*, Austin, TX, September 2007.

[85] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a Stream-processing Framework for Interactive Rendering on Clusters. In *Proc. 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 693–702, San Antonio, TX, 2002.

[86] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the* 3rd *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 187–200, New Orleans, LA, February 1999.

[87] I. Foster, C. Kesselman, J. M. Nick and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002. `http://www.globus.org/alliance/publications/papers/ogsa.pdf`.

[88] IBM Corporation. *IBM Virtual Machine /370 Planning Guide*, 1972.

[89] ID Software. Quake III Arena. `http://www.idsoftware.com/games/quake/quake3-arena/`.

[90] Intel Corporation. Intel Virtualization Technology (VT). `http://www.intel.com/technology/virtualization/index.htm`.

[91] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3):337–352, December 1997.

[92] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[93] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proc. Conference on Virtual Execution Environments (VEE)*, pages 1–11, San Diego, CA, June 2007.

[94] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron. Virtual Workspaces in the Grid. *Lecture Notes in Computer Science*, 3648:421–431, August 2005.

[95] Kernel-based Virtual Machine. `http://kvm.qumranet.com`.

[96] Khronos Group. OpenGL – The Industry Standard for High Performance Graphics. `http://www.opengl.org`.

[97] A. King. *Inside Windows 95.* Microsoft Press, 1995.

[98] Kmenc15. `http://kmenc15.sourceforge.net/`.

[99] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris. Global-scale Service Deployment in the XenoServer Platform. In *Proc. 1th Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2004.

[100] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proc. 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Callicoon, NY, June 2002.

[101] L4Ka Virtual Machine Technology. `http://l4ka.org/projects/virtualization/`.

[102] O. Laadan, D. Phung, and J. Nieh. Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*, Boston, MA, September 2005.

[103] H. A. Lagar-Cavilla, M. Satyanarayanan N. Tolia, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proc. Conference on Virtual Execution Environments (VEE)*, pages 33–43, San Diego, CA, June 2007.

[104] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, S. A. Rumble, M. Brudno, E. de Lara, and M. Satyanarayanan. Impromptu Clusters for Near-Interactive Cloud-Based Services. Technical Report CSRG-578, University of Toronto, July 2008.

[105] 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 *Proc. of Eurosys 2009*, pages 1–13, Nüremberg, Germany, April 2009.

[106] H. Andrés Lagar-Cavilla. VMGL Site. `http://www.cs.toronto.edu/~andreslc/vmgl`.

[107] H. Andrés Lagar-Cavilla, Niraj Tolia, Eyal de Lara, M. Satyanarayanan, and David O'Hallaron. Interactive Resource-Intensive Applications Made Easy. In *Proc. of the ACM/IFIP/USENIX 8th International Middleware Conference*, pages 143–163, Newport Beach, CA, November 2007.

[108] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Francisco, CA, December 2004.

[109] J. Liedtke. On Microkernel Construction. In *Proc. 15*th *Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain, CO, December 1995.

[110] B. Lin and P. Dinda. VSched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proc. ACM/IEEE Conference on Supercomputing*, Seattle, WA, November 2005.

[111] Linux VServer Project. Architecture Paper. `http://linux-vserver.org/Paper`.

[112] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17*th *USENIX Security Symposium*, pages 243–258, San Jose, CA, July 2008.

[113] L. Litty, H. Andrés Lagar-Cavilla, and David Lie. Computer Meteorology: Monitoring Compute Clouds. In *Proc. 12*th *Workshop on Hot Topic in Operating Systems (HotOS)*, Monte Verità, Switzerland, May 2009.

[114] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8*th *International Conference of Distributed Computing Systems (ICDCS)*, San Jose, CA, June 1988.

[115] J. López and D. O'Hallaron. Evaluation of a Resource Selection Mechanism for Complex Network Services. In *Proc. IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 171–180, San Francisco, CA, August 2001.

[116] Mark Williamson. XenFS. `http://wiki.xensource.com/xenwiki/XenFS`.

[117] M. McNett, D. Gupta, A. Vahdat, and G. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proc. 21*st *Large Installation System Adminstration Conference (LISA)*, pages 167–181, Dallas, TX, November 2007.

[118] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. USENIX Annual Technical Conference*, pages 15–28, Boston, MA, May 2006.

[119] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. Conference on Virtual Execution Environments (VEE)*, pages 13–23, Chicago, IL, June 2005.

[120] The Mesa 3D Graphics Library. `http://www.mesa3d.org/`.

[121] D. Meyer, G Aggarwal, B. Cully, G. Lefebvre, N. Hutchinson, M. Feeley, and A. Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proc. Eurosys 2008*, pages 41–54, Glasgow, Scotland, April 2008.

[122] Microsoft Inc. Azure Services Platform. `http://www.microsoft.com/azure/default.mspx`.

[123] Microsoft Inc. DirectX Home Page. `http://www.microsoft.com/windows/directx/default.mspx`.

[124] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[125] Mosso cloud service. `http://www.mosso.com/`.

[126] Motherboards.org. How to benchmark a videocard. `http://www.motherboards.org/articles/guides/1278_7.html/`.

[127] Mplayer Headquarters. `http://www.mplayerhq.hu/`.

[128] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, Boston, MA, December 2002.

[129] R. Nathuji and K. Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proc. 21st Symposium on Operating Systems Principles (SOSP)*, pages 265–278, Skamania Lodge, WA, October 2007.

[130] National Laboratory for Applied Network Research (NLANR). *RTT And Loss Measurements*, April 2006. `http://watt.nlanr.net/active/maps/ampmap_active.php`.

[131] E. B. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the Sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, Seattle, WA, November 2006.

[132] H. Nishimura, N. Maruyama, and S. Matsuoka. Virtual Clusters on the Fly – Fast, Scalable and Flexible Installation. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Rio de Janeiro, Brazil, May 2007.

[133] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 361–376, Boston, MA, November 2002.

[134] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[135] Parallels. 3D Graphics – Parallels Desktop for Mac. `http://www.parallels.com/products/desktop/features/3d/`.

[136] S. Parker and C. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proc. ACM/IEEE Conference on Supercomputing*, San Diego, CA, December 1995.

[137] P. Patchin, H. A. Lagar-Cavilla, E. de Lara, and M. Brudno. Adding the Easy Button to the Cloud with SnowFlock and MPI. In *Proc. Workshop on System-level*

Virtualization for High Performance Computing (HPCVirt), Nüremberg, Germany, April 2009.

[138] Personal communications with J. Gorm-Hansen and Y. Liu.

[139] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 351–366, Seattle, WA, November 2006.

[140] E. Pitt and K. McNiff. *java.rmi: The Remote Method Invocation Guide.* Addison-Wesley Professional, 2001.

[141] Pixar. RenderMan. `https://renderman.pixar.com/`.

[142] Platform Computing, Inc. Platform Enterprise Grid Orchestrator (Platform EGO), 2006. `http://www.platform.com/Products/platform-enterprise-grid-orchestrator`.

[143] pNFS.com. `http://www.pnfs.com/`.

[144] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–422, July 1974.

[145] S. Potter and J. Nieh. Reducing Downtime Due to System Maintenance and Upgrades. In *Proc. 19th USENIX Large Installation System Administration Conference (LISA)*, pages 47–62, San Diego, CA, December 2005.

[146] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proc. 9th Symposium on Operating Systems Principles (SOSP)*, pages 110–109, Bretton Woods, NH, October 1983.

[147] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proc. 18th USENIX Large Installation System Administration (LISA) Conference*, pages 241–254, Atlanta, GA, November 2004.

[148] QuantLib: a Free/Open-source Library for Quantitative Finance. `http://quantlib.org/index.shtml`.

[149] Chris Rapier and Michael Stevens. High Performance SSH/SCP - HPN-SSH, `http://www.psc.edu/networking/projects/hpn-ssh/`.

[150] E. Rather, D. Colburn, and C. Moore. The Evolution of Forth. In *History of Programming Languages-II*, pages 177–199, Cambridge, MA, April 1993.

[151] Red Hat. LVM2 Resource Page. `http://sources.redhat.com/lvm2/`.

[152] P. Reisner. DRBD - Distributed Replicated Block Device. In *Proc. 9*th *International Linux System Technology Conference*, Cologne, Germany, September 2002.

[153] T. M. Rhyne. Computer games' influence on scientific and information visualization. *IEEE Computer*, 33(12):154–159, December 2000.

[154] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, Jan/Feb 1998.

[155] D. Ritchie and K. Thompson. The UNIX Time-sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[156] J. Scott Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proc. 9*th *USENIX Security Symposium*, Denver, CO, August 2000.

[157] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *Proc. 11*th *Workshop on Hot Topic in Operating Systems (HotOS)*, San Diego, CA, May 2007.

[158] G. Van Rossum and F. L. Drake. *Python Language Reference Manual.* Network Theory Limited, 2003. `http://www.network-theory.co.uk/python/language/`.

[159] RPS-BLAST. `http://www.ncbi.nlm.nih.gov/Structure/cdd/cdd_help.shtml`.

[160] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: Accurate Mapping of Short Color-space Reads. *PLoS Computational Biology*, 5(5), May 2009.

[161] P. Ruth, P. McGachey, J. Jiang, and D. Xu. VioCluster: Virtualization for Dynamic Computational Domains. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*, Boston, MA, September 2005.

[162] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proc. of the Summer USENIX Conference*, Portland, OR, June 1985.

[163] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 377–390, Boston, MA, December 2002.

[164] M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.

[165] M. Satyanarayanan, M. A. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing*, 1(2):157–189, July 2005.

[166] D. Sayre. On Virtual Systems. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1969.

[167] C. Smowton. Secure 3D Graphics for Virtual Machines. In *European Workshop on System Security (EuroSec)*, Nüremberg, Germany, April 2009.

[168] S. Soltesz, H. Pötzl, M. Fiuczynski, A. C. Bavier, and L. L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proc. Eurosys 2007*, pages 275–287, Lisbon, Portugal, March 2007.

[169] Splash Damage. Enemy Territory Press Release. `http://www.splashdamage.com/?page_id=7`.

[170] S. Stegmaier, M. Magallón, and T. Ertl. A Generic Solution for Hardware-accelerated Remote Visualization. In *VISSYM '02: Proc. Symposium on Data Visualisation 2002*, pages 87–94, Barcelona, Spain, 2002.

[171] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *Proc. 10th Integrated Network Management (IM) Conference*, Munich, Germany, 2007.

[172] Sun Microsystems. Solaris ZFS - Redefining File Systems as Virtualised Storage. `http://nz.sun.com/learnabout/solaris/10/ds/zfs.html`.

[173] V. S. Sundaram. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[174] A. I. Sundararaj and P. A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *Proc. 3rd Virtual Machine Research and Technology Symposium*, pages 177–190, San Jose, CA, May 2004.

[175] A. Surie, H. A. Lagar-Cavilla, E. de Lara, and M. Satyanarayanan. Low-Bandwidth VM Migration via Opportunistic Replay. In *Proc. Ninth Workshop on Mobile Computing Systems and Applications (HotMobile)*, Napa Valley, CA, February 2008.

[176] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Francisco, CA, December 2004.

[177] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. 19th Symposium on Operating systems principles (SOSP)*, pages 207–222, Bolton Landing, NY, October 2003.

[178] SWSoft. Virtuozzo Server Virtualization. `http://www.swsoft.com/en/products/virtuozzo`.

[179] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292, Seattle, WA, November 2006.

[180] G. te Velde, F. M. Bickelhaupt, E. J. Baerends, C. Fonseca Guerra, S. J. A. van Gisbergen, J. G. Snijders, and T. Ziegler. Chemistry with ADF. *Journal of Computational Chemistry*, 22(9):931–967, September 2001.

[181] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17:323–356, February-April 2005.

[182] The Hadoop Distributed File System: Architecture and Design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[183] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proc. 10th Symposium on Operating System Principles (SOSP)*, pages 2–12, Orcas Island, WA, December 1985.

[184] S. Thibault and T. Deegan. Improving Performance by Embedding HPC Applications in Lightweight Xen Domains. In *Proc. Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, Glasgow, Scotland, 2008.

[185] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An Architecture for Internet Data Transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[186] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[187] Tom's Hardware. 3D Benchmarking – Understanding Frame Rate Scores. `http://www.tomshardware.com/2000/07/04/3d_benchmarking_/index.html`.

[188] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[189] University of Toronto. SnowFlock: Swift VM Cloning for Cloud Computing. `http://sysweb.cs.toronto.edu/snowflock`.

[190] Unreal Torunament 2004 Demo: Assault. `http://www.3dcenter.org/downloads/ut2004demo-assault.php`.

[191] UT2K4 Engine Technology. Unreal engine 2. `http://www.unrealtechnology.com/html/technology/ue2.shtml/`.

[192] VirtualBox. VirtualBox Changelog – Support for OpenGL Acceleration. `http://www.virtualbox.org/wiki/Changelog`.

[193] VirtualGL. `http://virtualgl.sourceforge.net/`.

[194] VMware. Virtual appliances and application virtualization. `http://www.vmware.com/appliances/`.

[195] VMware. VMotion: Migrate Virtual Machines with Zero Downtime. `http://www.vmware.com/products/vi/vc/vmotion.html`.

[196] VMware. VMware Workstation. `http://www.vmware.com/products/ws/`.

[197] VMware. Workstation 6 User's Manual – Experimental Support for Direct3D. `http://www.vmware.com/pdf/ws6_manual.pdf`.

[198] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proc.* 20th *Symposium on Operating Systems Principles (SOSP)*, pages 148–162, Brighton, UK, October 2005.

[199] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proc. 5*th *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Boston, MA, 2002.

[200] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the Development of Soft Devices – Short Paper. In *Proc. USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[201] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5*th *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–209, Boston, MA, December 2002.

[202] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5*th *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, Boston, MA, November 2002.

[203] Wikipedia. Quake3 engine technology. `http://en.wikipedia.org/wiki/Quake_III_Arena#Technology/`.

[204] Enemy Territory Demo: Radar. `http://www.3dcenter.org/downloads/enemy-territory-radar.php`.

[205] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

[206] Yahoo. Yahoo! Launches New Program to Advance Open-Source Software for Internet Computing. `http://research.yahoo.com/node/1879`.

[207] E. Zayas. Attacking the Process Migration Bottleneck. In *Proc. 11th ACM Symposium on Operating System Principles (SOSP)*, pages 13–24, Austin, TX, November 1987.

[208] N. Zeldovich and R. Chandra. Interactive Performance Measurement with VNC-Play. In *Proc. USENIX Annual Technical Conference, FREENIX Track* , pages 189–198, Anaheim, CA, April 2005.

[209] X. Zhang, K. Keahey, I. Foster, and T. Freeman. Virtual Cluster Workspaces for Grid Applications. Technical Report TR-ANL/MCS-P1246-0405, University of Chicago, April 2005.

[210] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practice and Experience*, 23(12):1305–1336, December 1993.