NEW ABSTRACTIONS AND MECHANISMS FOR VIRTUALIZING FUTURE MANY-CORE SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Sanjay Kumar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2008

NEW ABSTRACTIONS AND MECHANISMS FOR VIRTUALIZING FUTURE MANY-CORE SYSTEMS

Approved by:

Prof. Karsten Schwan, Committee Chair
College of Computing
Georgia Institute of Technology

Prof. Karsten Schwan, Adviser
College of Computing
Georgia Institute of Technology

Prof. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Prof. Calton Pu
College of Computing
Georgia Institute of Technology

Prof. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Parthasarathy Ranganathan
HP Labs

Date Approved: June 19, 2008

To my parents, Gulabi Devi and Ram Anugrah.

ACKNOWLEDGEMENTS

I have finally completed this dissertation but there are multiple people, without whose support I would have never reached this stage. These are people, who have always believed in me, and supported me through the ups and downs of this eventful journey. If I may say so, I have survived the PhD program because of them and I would like to take this opportunity to thank them all.

First of all, I would like to thank my advisor Prof. Karsten Schwan. Prof. Schwan not only patiently guided my research and kept me on track, he also provided valuable advice when I almost decided to quit PhD. He has been the best advisor I could have hoped for and it has been a real pleasure working under his guidance. I would like to thank Dr. Ada Gavrilovska who has been both a friend and mentor to me throughout this journey. I would also like to thank my mentors Dr. Vanish Talwar, Dr. Partha Ranganathan, Dr. Kiran Panesar, and Dr. Volkmar Uhlig whose guidance during various internships has significantly helped my research.

I would like to express my special thanks to Dr. Mustaque Ahamad, Dr. Calton Pu, and Dr. Sudhakar Yalamanchilli for giving me valuable inputs, thoughts and comments about my research. I am really grateful to my friends for being part of my support system and helping me endure and enjoy the ups and downs of Ph.D. life and Vibhore, Radhika, Vikas, Sandip, Himanshu, Ripal, Bala, Dulloor, Vishakha, Priyanka, Shalmali, and Dilip deserve a special mention.

Finally, I would like to thank my family members especially my nieces Saumya, Shubhi and Swati and my nephews Shobhit and Saurabh. Talking to them has always cheered me up and made me forget the disappointments and failures during the course of my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

To abstract physical into virtual computing infrastructures is a longstanding goal. Efforts in the computing industry started with early work on virtual machines in IBM's VM370 operating system and architecture, continued with extensive developments in distributed systems in the context of grid computing, and now involve investments by key hardware and software vendors to efficiently virtualize common hardware platforms. Recent efforts in virtualization technology are driven by two facts: (i) technology push – new hardware support for virtualization in multi- and many-core hardware platforms and in the interconnects and networks used to connect them, and (ii) technology pull – the need to efficiently manage large-scale data-centers used for utility computing and extending from there, to also manage more loosely coupled virtual execution environments like those used in cloud computing. Concerning (i), platform virtualization is proving to be an effective way to partition and then efficiently use the ever-increasing number of cores in many-core chips. Further, I/O Virtualization enables I/O device sharing with increased device throughput, providing required I/O functionality to the many virtual machines (VMs) sharing a single platform. Concerning (ii), through server consolidation and VM migration, for instance, virtualization increases the flexibility of modern enterprise systems and creates opportunities for improvements in operational efficiency, power consumption, and the ability to meet time-varying application needs.

This thesis contributes (i) new technologies that further increase system flexibility, by addressing some key problems of existing virtualization infrastructures, and (ii) it then directly addresses the issue of how to exploit the resulting increased levels of flexibility to improve data-center operations, e.g., power management, by providing lightweight, efficient management technologies and techniques that operate across the range of individual many-core platforms to data-center systems. Concerning (i), the thesis contributes, for large many-core systems, insights into how to better structure virtual machine monitors

(VMMs) to provide more efficient utilization of cores, by implementing and evaluating the novel *Sidecore* approach that permits VMMs to exploit the computational power of parallel cores to improve overall VMM and I/O performance. Further, I/O virtualization still lacks the ability to provide complete transparency between virtual and physical devices, thereby limiting VM mobility and flexibility in accessing devices. In response, this thesis defines and implements the novel *Netchannel* abstraction that provides complete location transparency between virtual and physical I/O devices, thereby decoupling device access from device location and enabling live VM migration and device hot-swapping. Concerning (ii), the *vManage* set of abstractions, mechanisms, and methods developed in this work are shown to substantially improve system manageability, by providing a lightweight, system-level architecture for implementing and running the management applications required in data-center and cloud computing environments. vManage simplifies management by making it possible and easier to coordinate the management actions taken by the many management applications and subsystems present in data-center and cloud computing systems. Experimental evaluations of the Sidecore approach to VMM structure, Netchannel, and of vManage are conducted on representative platforms and server systems, with consequent improvements in flexibility, in I/O performance, and in management efficiency, including power management.

CHAPTER I

INTRODUCTION

1.1 Background

The computing industry is currently undertaking a major push towards virtualized infrastructures, to realize their proven advantages in manageability, security and isolation. Virtualization is a key enabler of server consolidation and for sharing available platform resources among different applications. By running such applications in different, isolated *Virtual Machines* (VMs) or domains, isolation guarantees are provided by underlying virtual machine monitors (VMMs) or hypervisors (HVs). VMMs provide such guarantees by exporting virtual rather than physical resources and then controlling all virtual to physical resource mappings. VMMs can also help with the management of larger platforms like massively parallel processor (MPP) systems [102], by dividing them into multiple, smaller virtual platforms, thereby avoiding certain scalability issues experienced by the guest operating systems running on these platforms.

While system virtualization has traditionally been used in high end mainframes [102, 63], it is now becoming common in server class computing, personal computing, and to some extent, in the high performance domain (HPC) [25]. In part, this is because of the opportunities provided by virtualization technologies to better utilize and manage the increased processing power and other resources offered by these systems. The recent technology 'push' towards new hardware support for virtualization in multi- and many-core hardware platforms and in their interconnects is evidenced by Intel's VT and AMD's Pacifica extensions to their processor architectures, by Infiniband's hardware support for virtual lanes offering different levels of quality of service (QoS) and similar ongoing efforts for Ethernet-based technologies [54].

Concurrent with the ongoing technology 'push', a technology 'pull' for virtualization technologies is created by the need to efficiently manage the large-scale data-centers used

for utility computing and more loosely coupled virtual execution environments like those used in cloud computing [3]. Here, companies like HP, IBM, and VMWare are developing a rich set of virtualization technologies, to attain goals like blade server disaggregation, efficient system deployment and provisioning, ease of update and patching, and emergency response [83, 34]. At the platform level, for instance, platform virtualization provides an effective way to partition and use the ever-increasing number of cores in many-core chips. Similarly, I/O virtualization enables decoupling between I/O device access and its location, and the sharing of I/O devices across many VMs, which increases device throughput. With respect to management, virtualization enables, for instance, server consolidation and VM migration, which improve management of large data-centers, including improvements in operational efficiency, power consumption, and the ability to provision VMs to meet various requirements of the applications and the underlying platform [83, 86].

Ongoing development efforts face a number of challenges that can seriously affect the performance and functionality of virtual machines. For instance, (i) current VMM implementations face scalability issues on large scale many-core systems, thereby creating a need to redesign VMMs [71] concerning the ways in which their services are provided to guest VMs; (ii) VMMs lack proper support for *location transparency* of I/O devices, which results in inefficiencies concerning device handling during live VM migration and in dynamic changes to virtual to physical device mappings; and (iii) management methods targeting individual platforms to large scale virtualized data-centers remain lacking in terms of their scalability, efficiency, and effectiveness, because of inefficient underlying low-level mechanisms and lack of coordination between different management subsystems and applications. The following section describes these problems in more detail.

1.2 Key Research Problems

This dissertation addresses some of the issues and opportunities arising from the confluence of virtualization technology, many-core architectures and the management challenges presented by these systems, the latter including the lack of a coordination architecture to

facilitate coordination between VMs, VMMs, host hardware, and data-center level management applications. The following specific issues are addressed:

- *VMM efficiency:* Large scale many-core systems have fundamental differences both from SMP machines and from small scale multicore systems. For example, the compute-to-cache ratio for a large scale many-core platform is orders of magnitude larger than for traditional SMPs or for small scale multicore systems. The latency to access data from a different H/W thread is an order of magnitude lower than for a traditional SMP. Further, the specialized cores in heterogeneous many-core systems (accelerators), optimized for certain tasks, such as GPUs, network processors (NPs), etc. pose additional challenges to the VMMs, including efficient access and sharing of these heterogeneous cores among multiple VMs. As a result, current VMMs, originally designed for small scale (containing at most a few tens of cores) SMP systems, will not be sufficiently scalable nor efficient for large and possibly heterogeneous many-core systems containing hundreds of cores. It has been shown critical to address these differences in order to implement efficient VMMs and system software stacks [71]. *A redesign of systems software (and in virtualized environments, of the VMM) is required to create scalable and efficient future many-core systems.*

- *Location transparency* is a highly desirable properties for I/O virtualization, referring to the fact that a VM should be able to access its I/O devices irrespective of the VM's or device's locations. Transparency is important in multiple settings, including blade-servers, data centers, enterprise settings, and even in home-based, personal computing environments. It is important in data center environments, for instance, for live VM migration, for dynamic device consolidation, and for flexibility in server hardware configuration (e.g., for hardware disaggregation). It is important in home and office environments to attain higher levels of flexibility in accessing the variety of devices present in these infrastructures. *Current VMMs do not have adequate and efficient support for transparent access to I/O devices*, instead depending on external network transparent devices (e.g., iSCSI, SAN, NFS etc.) for such access. This results in

inefficient handling of I/O devices during live VM migration, and during dynamic changes of virtual to physical device mappings. VMMs also provide VMs with pass-through access to I/O devices to improve their I/O performance. However, location transparency is even more difficult to achieve for VMM pass-through devices, because VMMs are not involved during device access by VMs. As a result, live migration of VMs accessing pass-through devices is not supported by current VMMs. Hence, *an efficient and flexible VMM-level solution is needed for providing location transparency which works for both virtualized and pass-through devices.*

- *Effective management:* Managing a large number of VMs on a single large-scale many-core platform with devices, accelerators, CPUs, memory etc. already presents significant challenges. The manageability problem becomes even more complex for large scale data centers. An important issue in this context is that previous work on management has focused on solving specific problems, like power management [53, 65], VM provisioning based on resource requirements [83], VM migration [96] etc. This has resulted in solutions that operate in 'silos', use proprietary interfaces, and/or do not coordinate with each other. The resulting lack of coordination can cause inefficient or inappropriate management actions, and proprietary interfaces make the development of management solutions more costly. For example, power management can often run in conflict with performance management if both do not coordinate. On the other hand, proper coordination between reliability management and VM migration solutions can provide the novel functionality of migrating VMs to more reliable machine before the current platform fails, thereby significantly improving VM availability. Hence, *a 'coordination architecture' is required to enable coordination among the multiple management applications present in large-scale data centers.*

1.3 Thesis Statement

New VMM level abstractions and mechanisms are required to meet the needs of future many-core systems. Lower level VMM mechanisms must enable high performance via the flexible use of the many cores on these platforms, as well as the efficient use of their I/O

devices. Higher level abstractions must leverage the improved flexibility realized by lower level mechanisms to ensure, via runtime management, high performance despite varying workloads and dynamic changes in resource availability.

1.4 Contributions

This dissertation makes the following contributions:

- To improve VMM scalability and efficiency in large many-core systems, the dissertation introduces and experimentally evaluates the novel concept of "Sidecore", which is a system level abstraction enabling the presence and use of service cores specialized to certain tasks. A Sidecore is a 'dedicated' core providing specialized services to 'normal' cores. A Sidecore could be realized as a special purpose code running on a regular core and/or running on a specialized core (e.g., on a hardware accelerator like a communication processor or a GPU). In either case, the efficient use of Sidecores requires the re-design of certain VMM functionality. The Sidecore concept, its associated VMM re-design and its realization on both general purpose and specialized cores constitute one set of contributions of this dissertation.

- To realize location transparency for I/O devices, the dissertation defines, implements and evaluates the novel *Netchannel* mechanism. Netchannel extends the existing VMM level methods for I/O virtualization to provide for continuous access to I/O devices, during live VM migration, termed *virtual device migration*, and during dynamic changes to virtual to physical device mappings, termed *device hot-swapping*, for both virtualized and VMM pass-through device access. It also exploits the underlying I/O virtualization mechanism of the VMM to provide VMs access to remote devices without the need to modify them, essentially making remote devices appear as if they were local.

- Using the mechanisms described above and to exploit the improved degrees of runtime flexibility they enable, the dissertation also introduces the novel concept and realization of a 'coordination architecture', termed *vManage*, to improve the manageability of

virtualized systems and applications. vManage enables seamless coordination across the different management applications and policies present in large-scale data centers. The utility of vManage is demonstrated via a case study example of VM placement and dynamic VM provisioning, where coordination is shown useful across power management, SLA management, and reliability management.

1.5 Organization

The remainder of this dissertation is organized as follows.

Chapter 2 describes the design and implementation of the Sidecore approach, along with an implementation and its performance analysis for both homogeneous and heterogeneous many-core systems.

Chapter 3 describes the Netchannel mechanism, along with a specific realization and performance analysis for both virtualized and pass-through access.

Chapter 4 describes the design and implementation of the vManage coordination architecture, the dynamic VM provisioning example, and evaluations of its performance and utility.

Chapter 5 discusses related work, including brief architectural specifics of future multi-core systems and their impact on VMM/OS design.

Chapter 6 concludes the dissertation and presents ideas for future research.

CHAPTER II

RE-ARCHITECTING VMMS FOR MULTICORE SYSTEMS: THE SIDECORE APPROACH

Virtualizing many-core systems constitutes both an opportunity, i.e., to gain more flexibility in exploiting their cores, and a challenge, i.e., to do so in a fashion that can scale to the hundreds of cores foreseen in future systems. This chapter describes a technique to efficiently virtualize cores in a many-core system that attains improved scalability and gains the desired flexibility, by designating some cores as 'service' cores providing functionality that supports 'normal' general purpose cores. The result can be improved efficiency and better resource management, for both homogeneous and heterogeneous many-core systems.

2.1 Sidecores: Structuring VMMs for Many-Core Platforms

Virtualization technologies are becoming increasingly important for fully utilizing future many-core systems. Evidence of this fact are Virtual Machine Monitors (VMMs) like Xen [21] and VMWare [85], which support the creation and execution of multiple virtual machines (VMs) on a single platform in secure and isolated environments and manage physical resources of the host machine [26]. Further evidence are recent architecture advances, such as hardware support for virtualization (*e.g.* Intel's VT [87] and AMD's Pacifica [4] technologies) and I/O virtualization support from upcoming PCI devices [56].

There are multiple fundamental differences between large scale many-core systems and SMP machines (including small scale multicore systems). For example, the compute-to-cache ratio for a large scale many-core platform is orders of magnitude larger than for traditional SMPs or for small scale multicore systems, and the latency to access data from a different H/W thread is an order of magnitude lower than for a traditional SMP [71]. Further, the specialized cores in heterogeneous many-core systems (accelerators) [82], optimized for certain tasks, such as GPUs, NPs, etc., and less power consumption pose additional challenges to the VMMs, including efficient access and sharing of these heterogeneous

cores among multiple VMs. As a result, current VMMs, originally designed for small scale (containing at most a few tens of cores) SMP systems, will not be sufficiently scalable nor efficient for large and possibly heterogeneous many-core systems containing hundreds of cores.

Current VMM designs are monolithic, that is, all cores on a virtualized multi-core platform execute the same set of VMM functionality. This chapter describes an alternative design choice, which is to structure a VMM as multiple components, with each component responsible for certain VMM functionality and internally structured to best meet its obligations [42]. As a result, in multi- and many-core systems, these components can even execute on cores other than those on which their functions are called. Furthermore, it becomes possible to 'specialize' cores, permitting them to efficiently execute certain subsets of rather than complete sets of VMM functionality. While prior attempts at using specialized/dedicated cores to improve scalability and/or performance have focused on SMP operating systems in non-virtualized environments [14, 80, 101], this chapter describes an approach to enable VMMs to efficiently utilize specialized cores for improved performance and scalability.

There are multiple reasons why functionally specialized, componentized OS/VMMs are superior to the current monolithic implementations of OS/VMMs, particularly for future many-core platforms. Some reasons are applicable to both OSes and VMMs while others are specific to VMM based systems.

- Since only specific OS/VMM code pieces run on particular cores, improved performance is derived from reductions in cache misses, including the trace-cache, D-cache, and TLB. Further, VMM code and data are less likely to pollute a guest VM's cache state, thereby improving performance isolation for VMs and improving overall VM performance [71].

- By using a single core or a small set of cores for certain VMM functionality (e.g., page table management), locking requirements may be reduced for shared data structures,

such as guest VM page tables [8]. This can positively impact the scalability of multi-processor guest VMs.

- In heterogeneous many-core systems, some cores may be specialized for certain tasks (e.g., accelerators, such as GPUs, network processors (NPs), etc.) and hence, can offer improved performance for performing these tasks compared to other non-specialized cores [43, 19]. Using a componentized VMM to run the code for which these cores are optimized can significantly improve the performance and flexibility of many-core systems.

- Unique to the VMM-level targeted by this work is the fact that when a core executes a VMM function, it is already in the appropriate VMM-level processor state for running another such function, thus reducing or removing the need for expensive processor state changes (e.g., the VMexit trap in Intel's VT architecture). Some of the performance measurements presented in this chapter leverage this fact (see Section 2.2).

- To take full advantage of many computational cores, future architectures will likely offer fast core-to-core communication infrastructures [71], rather than relying on relatively slow memory-based communications. The communication between VMM components running on different cores can leverage these technology developments. Initial evidence are high performance inter-core interconnects, such as AMD's HyperTransport [33] and Intel's planned CSI.

In this chapter, we propose an abstraction of specialized cores, termed *Sidecore*, as a means for structuring future VMMs in many-core systems. *Sidecores* act as 'service' cores that provide services to 'normal' cores running VM and VMM code. A Sidecore can be similar to normal cores (e.g., same instruction set architecture) or it can be a specialized core (e.g., hardware accelerator like network processor) in heterogeneous many-core systems. However, it differs from *normal* cores in that it only executes one or a small set of VMM or VM functionality, whereas normal cores execute generic guest VM and VMM code. A service request to any such Sidecore is termed a *sidecall*, and such calls can be made from

a guest VM or from a platform component, such as an I/O device. The result is a VMM that attains improved performance by internally using the client-server paradigm, where the VMM (server) executing on a different core performs a service requested by VMs or peripherals (clients).

We demonstrate the viability and advantages of the Sidecore approach using two example implementations: (i) we use a homogeneous core as a Sidecore performing efficient routing of service requests from the guest VM to a VMM, to avoid costly *VMexits* in VT-enabled processors, and (ii) we use a network processor (NP) as an accelerator Sidecore in heterogeneous many-core system and enable its efficient use among VMs to improve application performance. Section 2.2 and Section 2.3 describe these two implementations and their evaluations in more detail.

2.2 Efficient Guest VM-VMM Communication in VT-enabled Processors

Earlier implementations of the x86 architecture were not conducive to *classical* trap-and-emulate virtualization [1] due to the behavior of certain instructions. System virtualization techniques for x86 architecture included either *non-intrusive* but costly binary rewriting [85] or efficient but highly intrusive paravirtualization [21]. These issues are addressed by architecture enhancements added by Intel [87] and AMD [4]. In Intel's case, the basic mechanisms in VT-enabled processors for virtualization are *VMentry* and *VMexit*. When the guest VM performs a *privileged* operation it is not permitted to perform, or when the guest explicitly requests a service from the VMM, it generates a VMexit, whereupon control is transferred to the VMM. The VMM performs the requested operation on the guest's behalf and returns to the guest VM using VMentry. Hence, the costs of VMentry and VMexit are important factors in the performance of implementation methods for system virtualization.

Microbenchmark results presented in Figure 1 compare the costs of VMentry and VMexit with the inter-core communication latency experienced by the Sidecore approach. These results are gathered on a Dell 1950 with 2.67 GHz dual-core X86-64 bit Intel Xeon, VT-enabled system, running a uni-processor VT-enabled guest VM (hereafter referred to as hvm domain). The hvm domain runs an unmodified Linux 2.6.16.13 kernel and is allocated

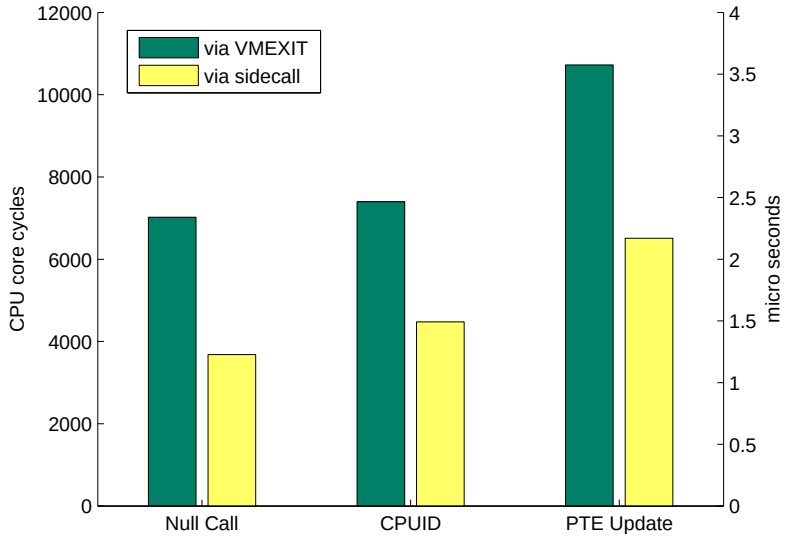

Figure 1: Latency comparison of VMexit and Sidecall approach

256MB RAM. The latest unstable version of Xen 3.0 is used as the VMM. The figure shows the VMexit latency for three cases when the hvm domain needs to communicate with the VMM: (1) for making a 'Null' call, where the VMCALL instruction is used to cause VMexit but VMM immediately returns; (2) for obtaining the result of a CPUID instruction which causes a VMexit and then, the VMM executes the real CPUID instruction on the hvm domain's behalf and returns the result; and (3) for performing page table updates, which may result in a VMexit and corresponding shadow page table management by the VMM.

The figure also presents comparative results when VM-VMM communication is implemented as a sidecall using shared memory (shm), as depicted in Figure 2. In particular, one core is assigned as the Sidecore, and the other core runs the hvm domain, with a slightly modified Linux kernel. When the hvm domain boots, it establishes a shared page with the Sidecore to be used as a communication channel. The operations mentioned above are implemented as synchronous shared memory requests to avoid VMexits. In the first case ('Null' call), the Sidecore immediately returns a success code via the shared memory. In the second case, it executes the CPUID instruction on the hvm domain's behalf and returns the result. The third case of page table updates is discussed in detail in Section 2.2.1.

Results demonstrate considerably higher performance for the sidecall compared to the

VMexit path. Hence, by replacing VMexits with cheaper Sidecore calls, the performance of the hvm domain and of system virtualization overall can be improved significantly. The higher performance is because of the fact that shared memory based sidecall latency is less compared to VMexit latency. It is notable that the experiments were conducted in April 2007, and with continuous optimizations in Intel's VT performance, the VMexit latency has been reducing over time (similar to system call latency optimizations using specialized instructions like sysenter, sysexit, etc.). Hence, we predict that at some point VMexit latency will cease to be more than the current shared memory based sidecall latency. However, we also predict that, with improvements in many-core architectures and development of inter-core interconnects like HyperTransport [33] and CSI, the inter-core communication latency will also be significantly reduced. Hence, we predict that sidecall latency will still be better than VMexit latency in future many-core systems. The Sidecore approach also benefits by avoiding system state pollution, e.g., cache, TLB, etc. Further, Sidecore requires minimal modifications to guest VMs and VMMs. To implement the sidecalls described above, only 7 lines of code were modified in the guest kernel, and only 120 lines of code in the form of a new kernel module were added.

2.2.1 Page Table Update

This section describes how the Sidecore approach can be used to reduce the number of VMexits during page table updates in a hvm domain.

Xen runs hvm domains by maintaining an extra page table, called the *shadow page table*, for every guest page table. The hardware actually uses the shadow page tables for address translation. The changes to the guest page tables are propagated to the shadow page tables by Xen. Page faults inside hvm domains cause VMexits and the control goes to Xen. If the fault didn't happen because the guest and shadow tables were not in sync, the fault is reflected back inside hvm domain and its page fault handler is invoked. It brings the faulting page into memory and updates the guest page table. Updating the guest table again causes VMexit because it was marked read-only by Xen. This time, Xen does the necessary propagation of changes from the guest page table to the shadow page table and

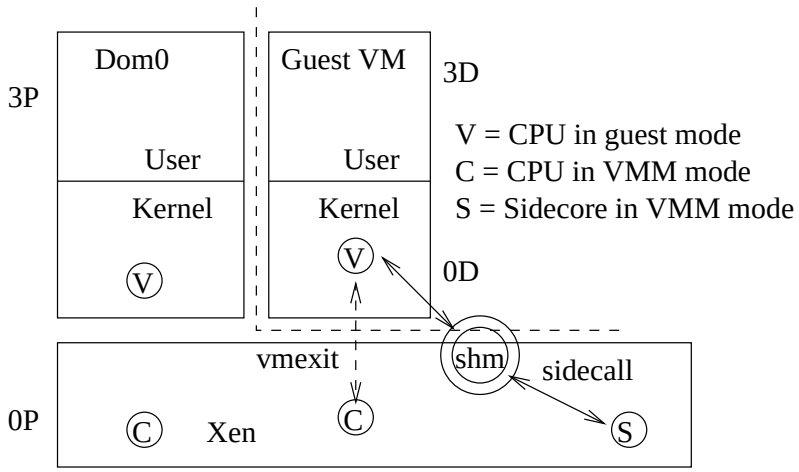

Figure 2: Implementing VMM services using Sidecore on an X86 64bit VT machine

resumes the hvm domain using VMentry. Hence, a typical page fault corresponding to creating a new page table entry causes two VMexits and two VMentries.

In this case, the Sidecore approach reduces the number of VMEntries and VMexits to one. The Sidecore spin waits for hvm domain requests in a tight loop. The domain's page fault handler code is modified so that instead of updating the guest page table itself (which would cause a VMexit), it makes a request to the Sidecore, providing the faulting address and the page table entry values. Since the Sidecore already runs in VMM mode, this process avoids the VMexit. The Sidecore updates the guest page table, propagates the values to the shadow page table and returns control to the hvm domain.

In another case of page table update when a guest page table entry is removed or modified (e.g., when a page is swapped out to disk), we also remove the corresponding shadow page table entry and in addition, we must flush the corresponding TLB entry. With the Sidecore approach, this requires sending an IPI for remote TLB flush. This is because the page fault handler is running on a different core and current processors do not provide an efficient mechanism for remote TLB flushes. This fact ends up eliminating the benefits of the Sidecore approach, since an IPI implicitly causes a VMexit. A hardware recommendation from our work, therefore, is that future many-core systems can benefit from efficient implementations of certain cross-core functions, in this case, an efficient hardware mechanism for remote TLB flushes.

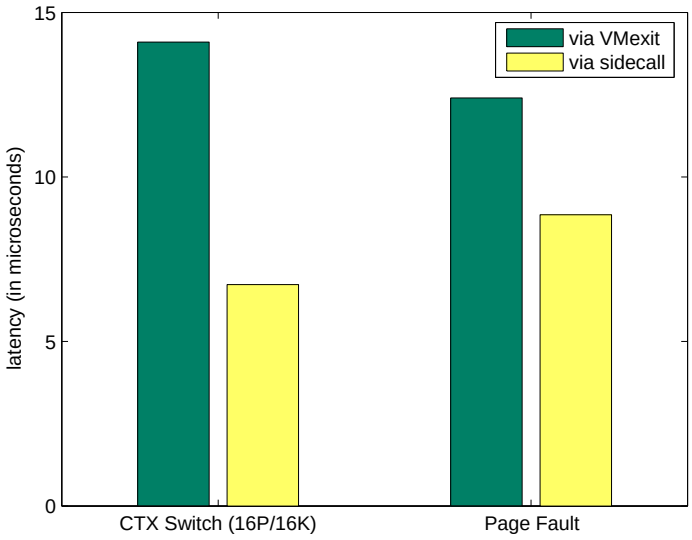

Figure 3: LMbench performance comparison for VMexit and Sidecore Call

Figure 1 shows the latency benefits of using the Sidecore approach for guest page table entry (PTE) updates. The results clearly show the approach's benefits, on average providing a 41% improvement in latency. Hence, using Sidecores for page table management can significantly improve the performance of virtual memory-intensive guest applications.

We also ran LMbench [51] performance benchmarks and a Linux kernel compilation to evaluate Sidecore page table management. Figure 3 shows LMbench's context switching and page fault performance comparisons. The context switch benchmark is shown for 16 processes, with process sizes of 16KB each. The latency improvement in page fault handling is due to low latency PTE updates performed by the Sidecore, as shown in Figure 1. The context switching performance for Sidecore is improved as a result of page fault performance improvements. The Linux kernel (version 2.6.16.13) compilation takes 676 seconds with VMexits and 668 seconds with Sidecore. This relatively low performance benefit is due to the fact that the majority of page faults being experienced are for memory mapped I/O (file I/O), which are not yet handled by our Sidecore implementation and therefore, follow the VMexit path.

2.3 C-core: Communication Accelerator using Sidecore

Sidecore approach can not only benefit VMMs by implementing their functions more efficiently, it can also benefit VMs, e.g., by improving I/O performance, by accelerating specialized VM processing, etc. Specifically, Sidecore approach can be used in heterogeneous many-core systems to efficiently use specialized cores (accelerators) among VMs. The use of heterogeneous cores in many-core systems has been increasing because of their higher performance and less power consumption [17]. Complimentary to previous efforts at improving application performance [80, 19] in non-virtualized environments, Sidecore approach focuses on efficiently sharing the accelerators among VMs. This section describes the use of the Sidecore approach to efficiently utilize network processors (NPs) as communication cores (c-core) in a heterogeneous many-core system to improve application performance by deploying onto those cores application specific processing. The Sidecore approach is used to create a virtualized network accelerator similar to self-virtualized network device [66] which can be efficiently shared by multiple VMs.

A network accelerator capable of application specific processing of VMs' network applications is useful under many circumstances. Large-scale distributed applications increasingly use middle-ware that dynamically deploys processing along the data paths of application-level overlays [67]. Processing ranges from simple data routing and forwarding, to the boolean functions carried out in distributed publish/subscribe [100] infrastructures, to application-specific actions that manipulate, transform, aggregate, and/or distribute information on sets of source-to-sink paths [45]. For instance, in commercial applications like the operational information systems used by airlines [55], middle-ware services execute simple business rules to transform and route business events between the company's central processing site and remote sites like airport terminals or baggage agents. In distributed scientific collaborations, including those supported by the SmartPointer framework [94], scientists rely on middle-ware services to monitor and steer remote experiments, accessing the subsets of experiments' outputs relevant to their current interests, at levels of detail appropriate for their local platforms (e.g., PCs vs. high end workstations) or their communication resources (e.g., available network bandwidth). An important attribute of all

such applications is that they require substantial amounts of processing on network data. Hence, a network accelerator can significantly improve overall application performance by accelerating the network processing.

We leverage current hardware trends [13, 82], which indicate that future processors will be multi-core systems comprised of many processing cores (processors) on the same chip. Moreover, there will be both homogeneous (like SMPs but more tightly coupled) and heterogeneous multi-core systems, the latter consisting of different cores optimized for different purposes. One example is IBM's cell processor [13], which has cores specialized for gaming applications. The example considered in this section is a multi-core platform with general computational and specialized communication cores, emulated by using a general purpose CPU with an attached network processor (NP). In particular, we are using an IXP2400 NP attached to a general Intel Xeon Linux host through a dedicated PCI bus. The VMM enables sharing of the NP by virtualizing and using it as a specialized core.

This approach to improving the performance of middleware-based applications running inside VMs is to enable them to 'best' use the heterogeneous, computational vs. communication-centric processors present in underlying hardware platforms. Toward this end, the Sidecore approach allows the VMM to use the communication core as a specialized core and through its virtualization, enable its sharing among multiple VMs. This allows the VMs to deploy their network specific processing on the communication core similar to one used for GPUs in non-virtualized environments [19].

Communication Core: A communication core (c-core) is similar to those of current NPs, with many internal processing units, each independently programmable and with sufficient resources to support a single communication stream at link speed. Intel's IXP NPs, for instance, has hardware processing units termed microengines that can operate in parallel, have their own registers and small amounts of program memory, and have shared access to hierarchically arranged memories of different speeds. These engines are programmed such that one or more of them can be allocated for control plane operations, and others can be allocated for data-plane operations. In addition, engine actions can be chained to form processing pipelines that implement more complex messaging operations. Engines

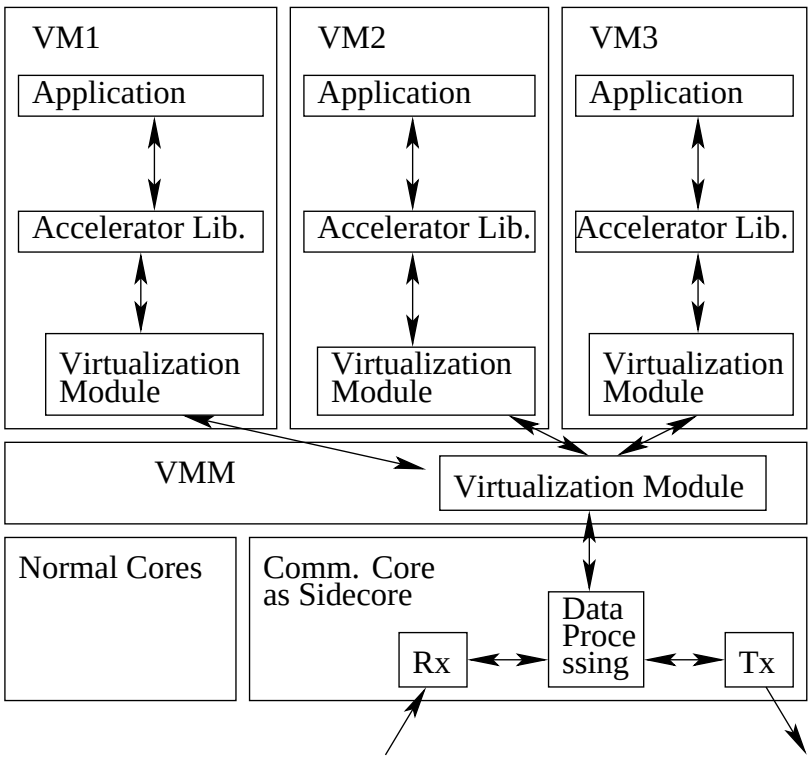

Figure 4: Using Sidecore for communication-core aware processing

have direct access to high speed network interfaces, with additional hardware present (in some IXP processors) for specialized processing tasks like encryption. Using the parallelism present in them, c-cores can significantly improve performance of network data processing.

Figure 4 shows the use of the Sidecore approach for utilizing communication cores in a virtualized system. The normal cores run general purpose code while the Sidecore runs VMs' data processing codes for which it has been optimized. An accelerator virtualization module runs in the VMM which virtualizes the accelerator and enables its sharing among multiple VMs with the help of a virtualization module inside every VM. The applications access the accelerator using an accelerator library which implements APIs for sending, and receiving data and sending application specific code to the accelerator. The library accesses the accelerator through the virtualization module present in each VM. Applications deploy their custom processing code on the accelerator which operates on the data belonging to the VM. The Rx unit receives network packets which gets operated upon by the data-processing unit before sending it to the VM. Similarly, the Tx unit sends data to the

network after it has been processed by the data-processing unit. To improve performance, the virtualization module inside the VMM can be moved to the accelerator itself, thus creating a 'self-virtualizing' accelerator and bypassing the VMM.

While the c-core can be used to improve any streaming data application, in this section, it is used with applications using the publish/subscribe messaging model [59, 100]. Supporting this model is particularly challenging because of the potential presence of a large number of subscribers to the same information, each of which may require data to be customized before receiving it. Customization is expressed with data filters provided by subscribers that must be executed on messages. A specific example is a set of filters that extract different information from airline flight records such as those that extract passenger vs. baggage vs. meal preference information. With its multiple processing units, the c-core can use either parallel processing or pipelined processing to execute the customization handlers/filters more efficiently.

2.3.1 Using C-cores in a Middleware System

The capabilities of traditional publish/subscribe middleware can be enhanced by making it 'c-core aware', i.e., by enabling it to dynamically create, deploy, and configure data processing. The execution model uses a VMM to share c-cores among VMs where VMs run pub/sub middleware which deploy application specific processing on the specialized cores. Since specialized cores typically have a different instruction set architecture (ISA) compared to the general purpose cores, the programming model uses a special compiler and a general purpose programming language to program them. The application code consists of two components; one that runs on normal cores and the other that runs on the c-core [19], termed 'kernel'[1]. These two components permit applications, via the middleware, to establish parallel or pipeline-structured sets of communication actions on the data-streams. The application calls APIs from the accelerator library to access the c-core hardware, and deploys the kernels on the c-core. The library allows multiple kernels to run simultaneously on the c-core thus enabling its sharing my multiple VMs. The library also manages all the resources of

[1]Not to be confused with an OS kernel

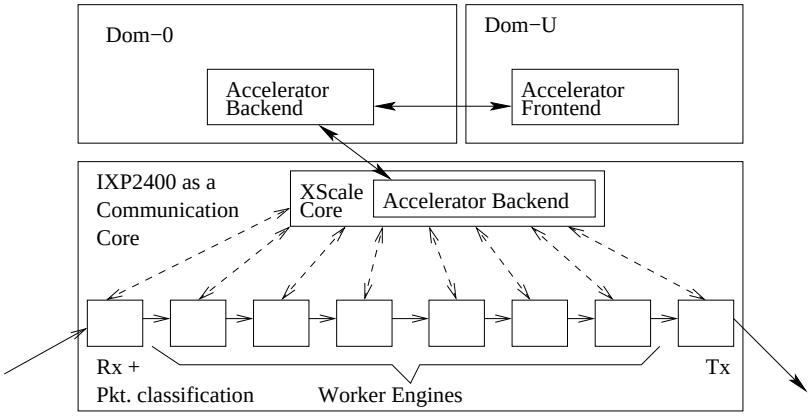

Figure 5: Using IXP2400 as a communication core in Xen

the c-core and performs admission control. These kernels get deployed in the data processing block in Figure 4 and operate on the incoming and outgoing data streams of the VMs. The goal is to permit c-cores to perform meaningful application-specific actions, thereby permitting applications to directly leverage their abilities to run at physical link speeds, *close* to the physical network, tightly linked with standard communication processing actions, and utilize hardware optimized for communication processing [57, 99] (e.g., multiple hardware queues, direct low latency access to physical network links, etc.).

2.3.2 Xen Implementation

We have implemented a prototype of applying the Sidecore approach to communication processing in Xen using IXP2400 NP [35] as a c-core, interconnected via a dedicated PCI interface to the host system. The goal is to demonstrate the benefits of using the approach for heterogeneous many-core systems. Application-level benefits are shown by implementing efficient 'continual' query processing on its data streams, by deploying the query processing kernels on the NP. The 8 microengines available on the IXP2400 provide the processing contexts and run either dedicated components of the execution environment like Rx, Tx, message fragmentation and reassembly, or they provide processing contexts for executing query processing kernels. The XScale core on the IXP runs virtualization and initialization operations, facilitates data transfers across the PCI interface, and performs c-core reconfiguration (e.g., deployment of new kernels on the c-core). As shown in Figure 5, all 8

microengines are assigned to different tasks, able to work in parallel on different message streams and/or in a pipelined fashion for a single stream. In our current implementation, six microengines remain available to execute kernels or chains of kernels (worker engines in Figure 5) on application-level messages. We expect future heterogeneous cores to have significantly more engines available for application-level processing tasks, e.g., nVidia GPUs.

Virtualization of network accelerator (c-core). We utilize our prior work on self-virtualized network devices [66] to implement a virtualized network accelerator in Xen. Figure 5 shows the current implementation, where an accelerator frontend OS kernel module runs inside guest VMs (Dom-Us, and an accelerator backend module runs in Dom-0 and in the IXP's Xscale core. The accelerator backend in Dom-0 enables sharing of the accelerator between multiple Dom-Us, while the backend on Xscale enables the dynamic deployment of kernels on IXP microengines. Two microengines handle receive (Rx), packet classification, and transmit (Tx) operations and 6 microengines are available for running application specific kernels. The accelerator backend enables Sidecore functionality by enabling guest VMs to use the IXP NP as a service core to deploy kernels. As shown in Figure 4, the applications access the NP using the accelerator library which in turn uses the accelerator frontend (equivalent to the virtualization module). The accelerator frontend forwards the requests to the Dom-0 backend which performs multiplexing and de-multiplexing of various requests and accesses the NP by communicating with the Xscale backend. For example, to deploy a kernel, the application calls the accelerator library API which forwards the request to Dom-0 backend via Dom-U frontend. Dom-0 backend requests the Xscale backend to deploy the kernel binary on the required microengines. The Xscale backend initializes the microengines, copies the kernel code into microengine instruction store, sets up required queues and data structures and finally starts the microengines. Although not implemented yet, the Dom-U frontend can directly communicate with the Xscale backend for fast data-path to improve performance, thus creating a self-virtualized accelerator using the Sidecore approach.

Kernels and their execution environment. Kernels are specified in microcode object format (.UOF), which has the capability to represent multiple processing contexts and

also has information about which processing context should be run on which microengine. The Xscale backend reads this information from the kernel header and copies the specific contexts to their respective microengines' instruction memories. Kernels can be created to operate in parallel or in a pipelined fashion(see Figure 5) using the UOF format.

Pipelined implementation of kernel chains. Representing more complex application-level processing actions as sets of chained kernels enables us to split the execution of time-consuming processing across multiple microengines, to gain pipeline parallelism, and to prevent any one execution context from becoming a bottleneck. The c-core pipeline implementation uses non-copying message queuing between different pipeline stages and implements an efficient hand-off protocol.

Dynamic reconfiguration through hot-swapping. In order to best utilize resources and match current application needs and platform resources, kernels need to be deployed and configured dynamically (hot-swapping of kernels). This can be initiated in response to changes in system resources or in end user interests. Simple reconfigurations involve changes to stream kernel parameters, such as changing the selection criteria in a select query kernel. More complex configurations involve hot-swapping stream kernels, including to address limitations in the amount of instruction memory in the microengines. Hot-swapping of kernel code is also required for kernel specialization where a kernel is redesigned to specialize it for its current execution environment and redeployed in place of its older version. Automatic kernel specialization, based on runtime monitoring as in other specialization systems has not yet been implemented.

The current implementation of hot-swapping keeps one of the microengines in idle state, while others are used to run kernels. During hot-swapping, the new kernel is loaded onto the idle microengine and then the control is switched from the old microengine to the new one. The idle microengine is started while the old microengine is stopped and becomes the idle microengine to be used for next hot-swap. This implies that the actual downtime for kernel processing is equivalent to the costs of stopping one microengine and starting the other one. Measurements show that this can be done in about 30 microseconds as compared to around 400 microseconds when the same microengine is stopped, new kernel code is loaded

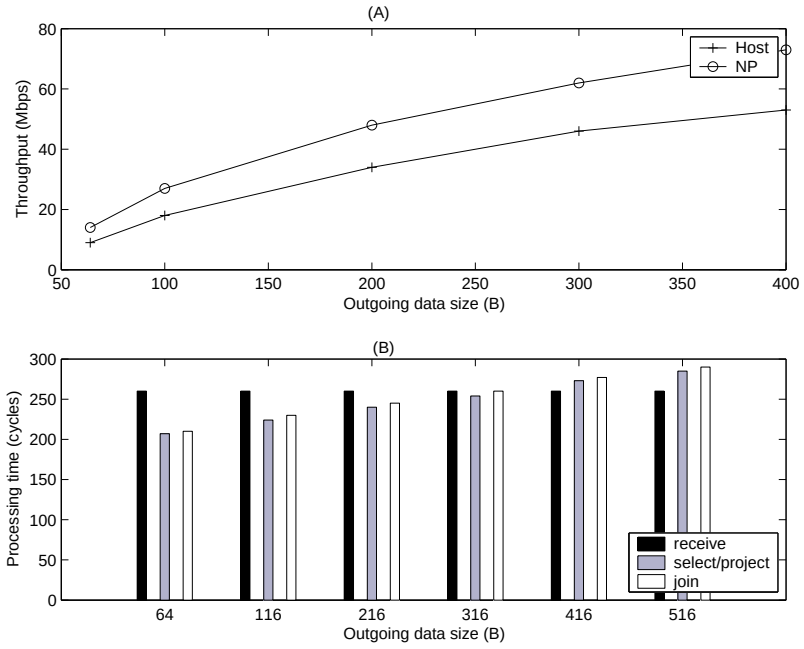

Figure 6: A. Query throughput with and without c-core, and B. Processing time of various pipeline stages

and then restarted. The drawback of this method is that one of the IXP2400 microengines must be kept idle for hot-swapping.

2.3.3 Efficient Query Processing on Data-Streams

To evaluate the performance and benefits of c-core, an example implementation of continual query processing of data-streams is evaluated on Xen hosts and their attached IXP2400s. The experimental testbed is comprised of 8 Dell Poweredge 2650 Machines (4 Xeon 2.8 GHz each), each having one IXP2400 (Radisys' ENP2611 board) attached to it via a PCI interface. IXPs are interconnected via a Gigabit LAN. Results demonstrate the benefits and feasibility of enabling the execution of application-level data processing on specialized heterogeneous cores.

Benefits of using c-core to run application kernels. The first set of experiments evaluates the c-core's ability to execute middleware-provided kernels. This evaluation uses an implementation of 'continual database queries' [45], comparing its performance with a corresponding general purpose core (henceforth referred to as g-core) based implementation. Queries implemented by operators are applied to streaming data in order to create

customized/personalized representations of that stream for different clients. The operators evaluated are the database operators (i.e., select, project, join) used in publish/subscribe infrastructures like IBM's Gryphon [100] product. The following specific test case is used.

Two publishers generate data streams and send them to a single *broker* VM. In addition, two subscribers submit queries to the *broker*, one query doing select/project on individual streams and the other doing join operation on the two streams. A total of three sub-streams is generated, two corresponding to individual streams and the third is join sub-stream. In the c-core based implementation, the *broker* receives the query kernel from a subscriber and deploys it on the NP. The query operators are applied to data streams that carry data from the operational information system of an airline (see [55] for more detail on that application). These data streams are directly sent to the c-core via its Gigabit links. Similarly, the sub-streams produced by the c-core are directly sent, via its output links, to the subscribers that desire them. On the c-core, operators are executed in pipelined fashion. In comparison, the implementation without c-core deploys query operators on the normal core, using a multithreaded approach, where the same message streams are processed with the same select, project, and join operators as those used in the c-core scenario.

A representative query for stream A is as follows: *select passengerList, mealPreference from A where A.source="Atlanta" AND A.destination="Paris" AND A.departTime="20:40 pm"*. Note that query operators can substantially differ in complexity, where complexity not only derives from the number of conditions tested and evaluated, but also from the number of different message fields accessed, the sizes of such fields, and the sizes of the messages created for output sub-streams.

The result shown in Figure 6.A compares the execution of a set of these queries on the NP vs. the g-core. The results shown are the attained throughput for c-core and g-core for different query complexities. We observe that for all data sizes, the c-core is capable of delivering improved throughput levels compared to the host. Note that the reasons for these improvements are complex, involving both the innate differences in hardware structures and capabilities on NP vs. g-cores and the general vs. specialized nature of g-core vs. NP execution environments, in the critical data path involving an entire Linux VM on the

Table 1: Query throughput and overhead calculation for various input stream speeds on the IXP2400.

I/P Thrput	O/P Thrput	Overhead(usec)	pkts sent	pkts processed	pkts dropped
200 Mbps	200 Mbps	31	200000	255000	0
350 Mbps	350 Mbps	32	200000	230000	20000
495 Mbps	450 Mbps	35	200000	160000	80000
695 Mbps	450 Mbps	50	200000	160000	105000

g-core vs. minimal runtime support on the NPs.

Ability to deploy kernel chains. Figure 6.B presents time in terms of the processing cycles required by different components of the application, and by different query kernels (for different message sizes). We observe that query operations can be executed in approximately the same amount of time as the receive operation for output data streams of the same size. This demonstrates that the NP is capable of supporting chained queries. Moreover, we can hypothesize from these results that when operators reduce the amounts of data produced vs. received (e.g., an outgoing sub-stream contains a subset of the information contained in the arrival stream), similar performance results will be attained. It is also notable that, by executing stream processing on the c-core, the g-core is freed to carry out other application-level tasks.

We next determine the limits to which the IXP2400 can sustain query processing by increasing stream throughput. The case measured constitutes a worst case in that it does not reduce the size of output compared to input data (using 600 byte messages). Table 1 shows that the IXP2400 can sustain close to 350 Mbps of throughput. Beyond that, packet drops become significant. We expect the NP to be able to sustain higher levels of throughput for queries that lead to reduced size output streams (approximating link rates). Measurements also show the overheads incurred in query execution. In fact, overheads increase as the input speed increases because of memory and other resource contention issues. Please note that the output throughput is more than the input throughput in some cases. This is because we send two input streams (of 100000 packets each) but output three sub-streams,

two corresponding to the individual streams and the third sub-stream as a result of a join query operation on the two streams.

2.4 The Sidecore Approach: Discussion

The performance benefits of the Sidecore approach for VM-VMM communication might become less pronounced as VMexit/VMentry operations are further optimized [1]. However, we believe that the Sidecore approach can still provide significant advantages. First, since low latency inter-core interconnects are important for attaining high performance for parallel programs on future many-core platforms, they are likely to be an important element of future hardware developments. The latency of inter-core communication can be further decreased by cache sharing among cores or by direct addressed caches from I/O devices [93]. Second, re-architecting the virtualized system as a client-server design via Sidecores provides a clear separation of functionality between VM and VMMs. This can result in better performance for VMs due to reduced VMM *noise*, caused by the pollution of architectural state [7]. Moreover, it has been shown that using functional partitioning is one of the important techniques for improving scalability of system software in large scale many-core and in multiprocessor systems [71].

One disadvantage of the Sidecore approach is that its current implementation requires minor modifications to the guest OS kernel. However, these changes are significantly smaller than a typical paravirtualization effort – e.g., for page tables updates using Sidecore, 120 lines for setting up the shared communication ring and 7 lines for sending a sidecall request. Hence, the approach has a desirable property of *minimal* paravirtualization. Besides, the approach can be dynamically turned on/off with a simple flag, allowing the same guest kernel binary to execute on a VMM with/without the Sidecore design. Another trade-off is that the Sidecore approach causes wasted cycles and energy due to the CPU spinning used to look for requests from guest VMs. This can be alleviated via energy-efficient polling methods, such as the monitor/mwait instructions available in recent processors. Furthermore, the use of Sidecores to run specialized functions might make them unavailable for normal processing. A Sidecore implementation that dynamically finds available cores would

alleviate this problem, but we have not yet implemented that generalization and therefore, cannot assess the performance impacts of runtime core selection. For a static approach, we hypothesize that in future large-scale many-core systems like those in Intel's tera-scale computing initiative [81], it will be reasonable to use a few additional cores on a chip for purposes like these, without unduly affecting the platform's normal processing capabilities.

Using the Sidecore approach to efficiently share heterogeneous cores among VMs requires that these cores should be able to run multiple kernels simultaneously. While the IXP NP allows this, many other specialized cores, e.g., nVidia's GPU, do not open up their implementations to enable such sharing. The Sidecore approach would not be as useful for these cores, unless dynamic compiler techniques to merge multiple kernels into a single kernel can be developed [19].

2.5 Related Work

Substantial prior research has addressed the benefits of utilizing dedicated cores, both in heterogeneous [28] and homogeneous [69] multicore systems. Self-virtualized devices [66] provide I/O virtualization to guest VMs by utilizing the processing power of cores on the I/O device itself. In a similar manner, driver domains for device virtualization [62] utilize cores associated with them to provide I/O virtualization to guest VMs. The Sidecore approach presented in this paper utilizes dedicated host core(s) for system virtualization tasks. Particularly, we advocate the partitioning of the VMM's functionality and utilizing dedicated core(s) to implement a subset of them. Similar approaches are used in operating systems, where processor partitioning is used for network processing [68, 10].

Computation Spreading [14] attempts to run similar code fragments of different threads on the same core and dissimilar code fragments of the same thread on different cores. Another approach is to run hardware exceptions on a different hardware thread (or core) instead of running it on the same thread (core) [101]. While these solutions are targeted for better utilization of the micro-architecture resources such as caches, branch predictors, instruction pipeline etc., our solution is targeted at improving VMM performance and scalability for large scale many-core systems.

Intel's McRT (many-core run time) [71] in sequestered mode uses dedicated cores to run application services in non-virtualized systems. This approach requires major modifications in the application to utilize the parallel cores. This is in contrast to the Sidecore approach, which requires only minor modifications to the guest VM's kernel and is aimed at improving overall system performance.

There have been previous attempts at using host attached network processors in non-virtualized environments to execute application specific services closer to the network [24]. Further, examples of executing various protocol- vs. application-level actions in different hardware context include splitting TCP/IP stack across general purpose cores and dedicated network devices such as NPs, FPGA-based line cards, or dedicated cores in SMP systems [9, 69], or splitting the application stack as with content-based load balancing for an http server or for efficient implementations of media services. Besides NPs, there are multiple ongoing efforts on building heterogeneous many-core systems, e.g., Intel QuickAssist [82], and then exploiting the specialized and massively parallel nature of heterogeneous cores to improve application performance, e.g., Cell SDK for Cell processor [13], CUDA [19] and Accelerator [80] for GPUs, etc. Other ongoing efforts on specialized execution environments using heterogeneous cores include Cellule [27] using Libra [5]. While these efforts utilize accelerators in non-virtualized environments, the Sidecore approach attempts to efficiently utilize and share these accelerators among multiple VMs in virtualized environments.

2.6 Summary

This chapter presents the Sidecore approach to enhance system-level virtualization in future multi- and many-core systems. The approach factors out some parts of the VMM or VM functionality in order to execute it on a specific host core, termed *Sidecore*. We demonstrate the benefits of this approach by using it to: (i) avoid costly VMexits on VT-enabled processors, and (ii) deploy application specific continual query processing on communication cores (NPs). Performance results demonstrate that the Sidecore approach improves the overall performance of the virtualized system. Apart from such performance improvements, the approach also improves the way in which the resources present in many-core systems

are used. For example, through dynamic hot-swapping of kernels, Sidecore can efficiently manage resources used by different VMs. The accelerator backend can also dynamically map a VM's request for an accelerator to various accelerators, when there are multiple such accelerators present in the system.

The Sidecore approach, however, alone can not improve the overall performance and management of virtualized systems. There are many other components in the system (e.g., I/O devices) which are also very important to virtualize efficiently apart from the processing cores. I/O virtualization is the next most important aspect of these systems because: (i) I/O virtualization incurs the most amount of overhead during virtualization, and (ii) the vast amount and variety of I/O devices present in current data centers and other computing systems makes it very important to efficiently share and manage them. In response, the next chapter in this dissertation attempts to solve the problem of *location transparency* which provides flexibility in I/O virtualization and further improves manageability of virtualized systems.

CHAPTER III

NETCHANNEL: A VMM-LEVEL MECHANISM FOR LOCATION TRANSPARENCY OF I/O DEVICES

In the previous chapter, we discussed the *Sidecore* approach to efficiently utilizing and managing cores in (possibly heterogeneous) many-core systems. A second challenge for VMMs is their efficient interaction with I/O devices especially since VM migration is one of the most powerful mechanisms for improving the manageability of virtualized systems. Current I/O virtualization, however, presents *location transparency* problems for 'live' VM migration under certain circumstances. This chapter addresses this problem, by describing a novel mechanism, termed *Netchannel*, to provide *location transparency* to I/O virtualization. The performance of Netchannel can be improved by using dedicated cores provided by Sidecore approach.

3.1 Background

System-level virtualization [21, 85] is becoming increasingly important because of benefits in providing isolation, consolidation, containment, and manageability. An important attribute of virtualization is the ability to completely and seamlessly migrate a virtual machine (VM) from one physical machine to another [73, 84, 16]. This helps VMMs (e.g., Xen [21] and VMWare [85]) to provide promised consolidation and manageability benefits. A highly desirable feature of virtualization is *location transparency*, which refers to the complete decoupling between a VM's location and the location of its I/O devices. Location transparency of I/O devices is useful for 'live' VM migration, i.e., the efficient and dynamic migration of VMs with un-noticeable service disruption [16]. A key requirement for live VM migration without downtime for the services being run is continuous and transparent access to its virtualized I/O devices, termed *virtual device migration*. Unfortunately, current methods for I/O virtualization do not provide such continuity and transparency for VMs' devices. Instead, they must rely on other technologies to provide them, such as

hardware-based methods like storage area networks (SANs) for disks.

Location transparency is also useful in dynamically changing virtual to physical I/O device mapping. Also known as *device hot-swapping*, this refers to changing a VM's physical I/O device with another 'similar' I/O device on the fly without any observable service disruption. Device hot-swapping may be required for multiple reasons, e.g., to improve performance, to improve reliability or to do hardware maintenance. Device hot-swapping is even more critical when a VM with pass-through access to an I/O device is migrated to another machine because this must also perform a device hot-swapping to maintain the VM's pass-through access to the I/O device. Current VMMs do not provide support for device hot-swapping and often they depend on costly external solutions like SAN or smart hardware based disk replication controllers.

This chapter presents VMM-level support for location transparency in accessing I/O devices, termed *Netchannel*. Netchannel provides *virtual device migration* (VDM) and *device hot-swapping (DHS)* for both virtualized and pass-through access to the devices. Finally, for locally attached devices, Netchannel also provides *transparent device remoting (TDR)*, where a locally attached device can be transparently accessed remotely by a VM. The combination of seamless VDM, DHS, and TDR enables and benefits the dynamic management and fault tolerance methods envisioned for next generation virtualized systems. More generally, it provides servers and end user applications running in VMs with richer choices in device usage, without requiring costly hardware solutions. Seamless VDM addresses a key issue with live VM migration, for both fully and para-virtualized VMs, which is their ability to continue to access the devices they are using. For instance, if the VM used devices present on the machine from which it is being migrated, migration makes these remote devices inaccessible to the VM. Further, even when the devices are present on the network (e.g., an NFS-based disk), migration requires that the virtual device state (including pending I/O transactions) be seamlessly migrated along with the VM. Unfortunately, VMMs do not currently handle pending I/O operations during VM migration, instead relying on external techniques like the Fiber Channel-based hardware solutions used for disks or like the software-based remote device access solutions integrated into guest operating systems

(e.g., iSCSI disks inside VMs). In contrast to such point solutions, our approach is to design a new, general VMM-level mechanism, termed *Netchannel*, enabling VDM, DHS, and TDR.

The Netchannel VMM-level abstraction presented in this chapter has multiple interesting properties. First, it is OS agnostic, since it does not rely on guest operating systems to provide OS support for remote device access and for device hot-swapping. Second, it is general, in that it works for both fully and para-virtualized guests. Third, it is cost-effective, as it eliminates the need for hardware solutions where such costs are not warranted. To demonstrate these properties, *Netchannel* has been implemented for both block and USB devices inside the Xen VMM. The implementation offers the following degrees of flexibility to virtual machines and their applications:

- *Virtual Device Migration (VDM)* – for both locally attached and networked devices, Netchannel enables the seamless migration of virtual device state (including pending I/O transactions) without any noticeable downtime, thereby permitting continuous device operation transparent to the VM, during and after live VM migration;

- *Device Hot-Swapping (DHS)* – virtual device hot-swapping, again without any noticeable downtime, can be done for both locally attached and networked devices; this makes it easy to develop rich fault tolerance and load balancing solutions for virtualized systems [79, 74].

- *Transparent Device Remoting (TDR)* – besides providing virtual device migration for locally attached devices, Netchannel's TDR functionality provides a common infrastructure for remotely accessing non-networked devices, by using existing device virtualization solutions.

- *High Performance* – for strongly networked systems, like those found in datacenters or even in offices and homes, Netchannel offers levels of performance for remote device access similar to those seen for local devices, in terms of realized device bandwidths.

Experimental evaluations on a cluster of machines presented in Section 3.5 of this chapter validate these claims.

Netchannel affords administrators with substantial flexibility in how to structure or configure their systems. In blade servers, for example, some cabinets may be configured to be *device-less*, coupled with device-heavy cabinets elsewhere. This can reduce per blade costs without limiting configuration flexibility in terms of where certain virtual machines may be run. An advantage in home or office settings is the ability to access arbitrary devices, which may be attached to other machines or to the network, without having to use costly hardware solutions. Beyond these basic capabilities, extensions of the Netchannel concept can be used to realize new levels of device sharing and new device capabilities, by presenting to different VMs different views of the devices present in the underlying platforms, which we term 'logical' or 'soft devices' [89]. An example is the association of distributed file system properties like fine-grained sharing, dynamic allocation etc. with virtual disks, essentially creating a virtualization aware file system [58].

We summarize by briefly outlining this chapter's contributions. First, the Netchannel architecture and its Xen implementation provide a flexible and efficient approach to providing the rich device-level functionality needed for effective virtualization, i.e., VDM, DHS, and TDR. Second, experimental evaluations with both remote disk and remote USB devices demonstrate that the Netchannel-based realization of TDR offers levels of performance comparable to those of existing non-transparent kernel-level solutions. Third, evaluations with a representative multi-tier application demonstrate seamless live VM migration when using Netchannel-based VDM and DHS for the disk devices used by this application.

The remainder of the chapter is organized as follows. Section 3.2 presents Netchannel's software architecture. Section 3.3 presents Netchannel's implementation in Xen. Section 3.5 evaluates various Netchannel functionalities. Section 3.6 discusses some of the related work, and Section 3.7 provides summary and conclusions.

3.2 Netchannel Software Architecture

The Netchannel VMM extension for efficient device virtualization is depicted in Figure 7. To explain it, it is necessary to describe the device virtualization mechanisms used in current VMMs. Device virtualization typically involves running two stacks of device drivers: one in

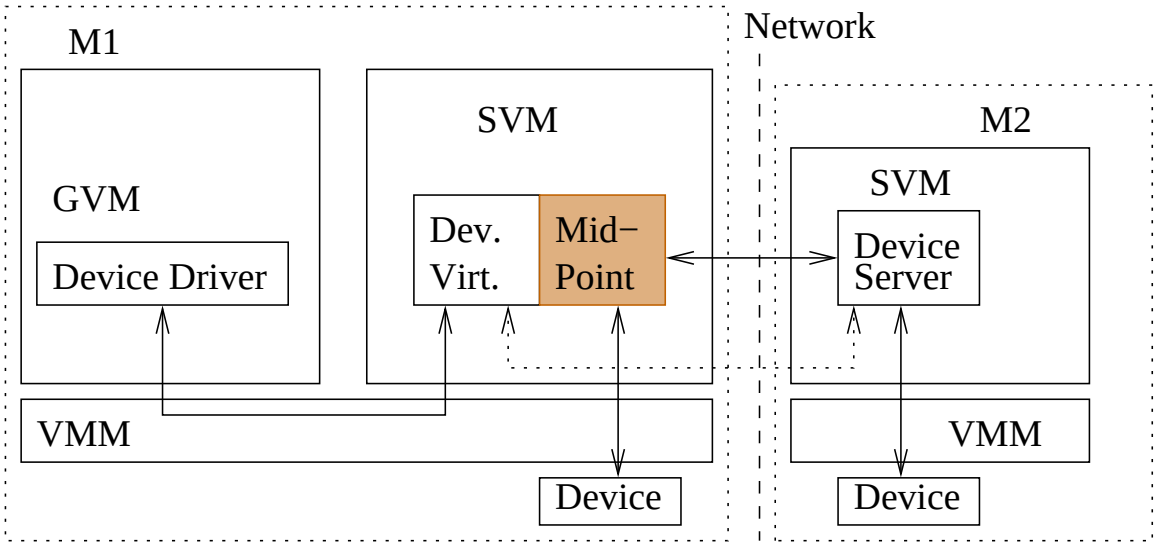

Figure 7: Netchannel Software Architecture

the guest VM (GVM), which is the GVM's device driver (DD) and another, termed device virtualization (DV) driver, running in the VMM or more likely, in a special privileged VM, called a Service VM (SVM). The DV exports a virtual device inside GVM and DD attempts to access it. Every attempt by the DD to access the device causes control to transfer to the DV, which properly virtualizes these accesses. The DV can use either a locally present physical device for the GVM's virtual device, or it can use a remote device by accessing a Device Server (represented by a dotted line) using some remote access protocol (e.g., NFS). In fully virtualized systems, the GVM runs the entire unmodified device driver stack and DV emulates a virtual device, while in the para-virtualized case, the driver stack is split: the GVM runs only the upper part of the split device driver stack, termed frontend (FE), and the SVM runs the lower part of the split stack, termed backend (BE). Examples of both implementations are VMWare workstation [78] and Xen [21], respectively. Netchannel's current implementation uses Xen's FE/BE mode of device virtualization but without further modifying the VM. Live migration support for fully virtualized VMs has been recently added in Xen and we are currently working on implementing Netchannel for fully virtualized VMs.

Multiple technical issues must be addressed when implementing virtual device migration, device hot-swapping, and transparent device remoting. Most important of these is to deal with I/O and device states. Specifically, the DV driver maintains the state of the VM's

virtual devices which during live VM migration, must be transferred from the old DV to the new DV on the new machine. This state also includes the currently pending I/O transactions (i.e., the I/O operations issued by the DD but not yet completed) issued before the completion of VM migration. These transactions are issued by the DV to the device and results must be returned to the DD to ensure proper device operation. After VM migration, however, the VM has moved to a new machine, and these completed transactions cannot be returned to the DD in the normal manner. Thus, old and new DV must cooperate to ensure these pending I/O operations are transparently handled, along with the rest of the device state. In doing so, the sequence of I/O operations (issued before and after live migration completes) must be maintained to provide complete transparency. If the devices are locally attached to the machine before VM migration, live migration further requires that the new DV must cooperate with the old DV to access the devices remotely. Similarly, device hot-swapping requires that the DV must ensure that the new device is exactly in the same state as the original device to provide complete transparency. The next subsections describe how the Netchannel architecture addresses these issues.

3.2.1 Virtual Device Migration

As shown in Figure 7, to provide seamless VDM, the Netchannel architecture extends the device virtualization (DV) module with a *Mid-Point (MP)* module that monitors the virtual device state and the pending I/O operations. During live migration, the Mid-Point (MP) establishes a communication channel with the other MP (on the new machines) to co-ordinate and allow for the seamless transfer of the device state, including pending I/O operations. The extended DV can implement multiple approaches to handle these pending I/O operations.

3.2.1.1 Basic Mechanism

VM migration involves multiple steps, including freezing a VM, destroying its interfaces with the current VMM, creating a new VM on the destination machine, filling this VM's memory with the memory pages of the frozen VM, and finally, unfreezing the new VM. Live VM migration is a special case in which the freeze duration of the VM is kept un-noticeably

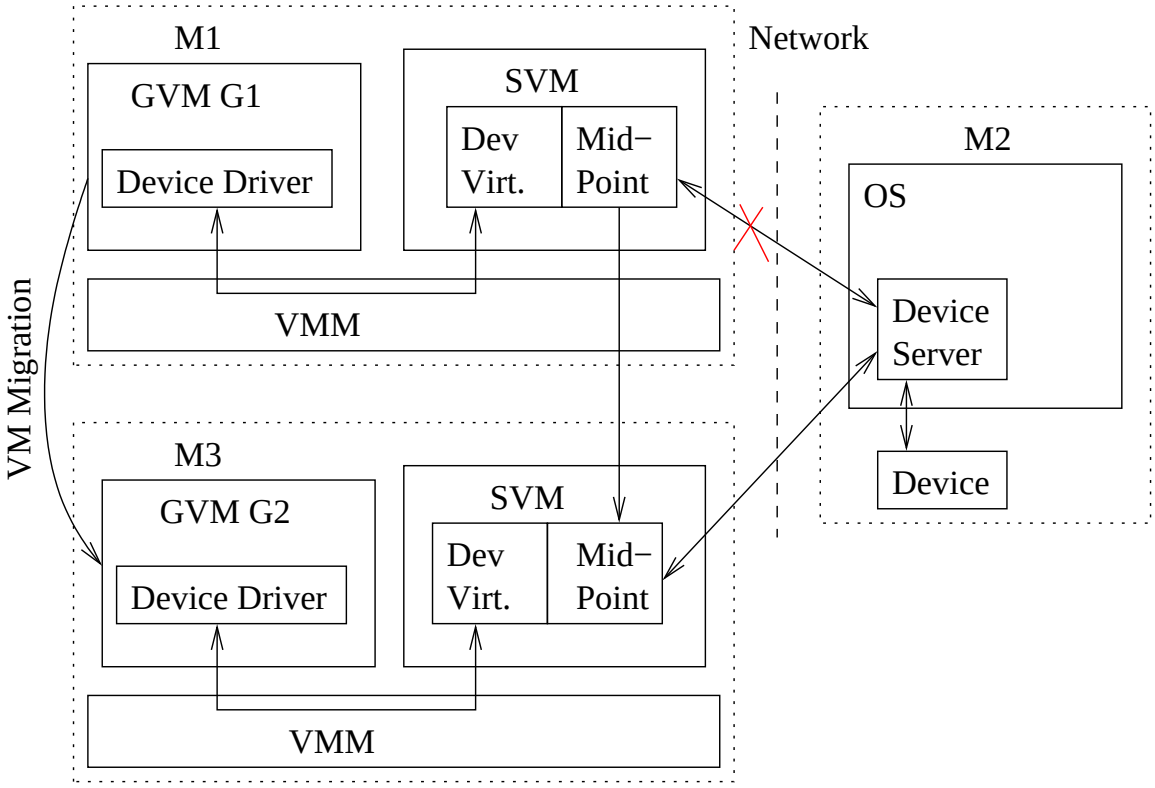

Figure 8: Virtual Device Migration

small. The method for live migration used in Xen3.0 is described in [16].

Virtual Device Migration is depicted in Figure 8. Assume that a guest VM named G1 is migrating from host machine M1 to host machine M3, while G1 is accessing a device on M2 (the device could be locally attached to M1 as well). During live VM migration, G1 is frozen and the DD-DV communication is suspended. During this suspension, DV breaks its connection with the DD, but it does not break its connection with the device. On the destination host M3, a new VM G2 is created, and its OS pages are filled from the suspended VM G1. G2 uses G1's configuration so that it exactly looks like G1. The old DV sends the device state to the new DV which creates a new virtual device with the same state. The new DV on M3 connects to the Device Server on M2, while the old DV breaks its connection with the device server. Next, G2 is un-paused and its DD starts communicating with the DV. At this point, migration completes and communication between DD and DV on M3 resumes. While all subsequent accesses to the device use the new path, the entire process of VM migration is transparent to the guest VM.

3.2.1.2 Handling Pending I/O Transactions

As mentioned earlier, a potential issue for virtual device migration is that there may be pending I/O transactions issued by G1's DD at the time G1 is suspended. By the time these I/O transactions complete, the VM has already migrated, and the old DV cannot return the I/O results to the DD in the normal fashion. Transparency demands that we deal with these pending transactions. The Netchannel architecture, comprised of Mid-Points and the communication channel connecting them, makes it possible to implement a variety of approaches for dealing with pending I/O transactions:

- **Discard pending I/O transactions** – the DV on M1 DV_{M1} can discard these I/O transactions. This approach relies on the error recovery mechanism in the DD and in its upper level drivers. After migration, eventually, DD will realize the failure of the pending I/O transactions and as part of error recovery, its retry of those I/O operations will eventually succeed. While suitable for devices with highly resilient protocol stacks, e.g., the NIC device, the drawback of this approach is its reliance on robust error recovery in guest device drivers.

- **Re-issue pending I/O transactions** – the Mid-Point on M1 MP_{M1} can store the list of pending transactions and send it to the new Mid-Point (MP_{M3}) during live migration which re-issues these transactions on the new machine M3. However, issuing these pending transactions twice (once on M1 and then on M3) can cause inconsistent end results (e.g., if these transactions include a read followed by a write to the same block on a disk, re-issuing them will result in second read returning different value than first read). It also causes significantly more latency during live migration, because all pending operations are re-issued from scratch.

- **Bringing the device into a quiescent state before migration** – before freezing the VM, its virtual device is brought into a state with no pending I/O operations, termed the *quiescent* state. To do this, however, the virtual device should be put into a state where it can't accept any I/O requests, causing DD to stop issuing I/O operations to the device. The VM is migrated when all the pending I/O transactions complete.

After migration the virtual device is again brought into the normal operating mode causing DD to resume issuing I/O operations. This approach requires that the virtual device and DD must support such a 'no more I/O operations' state. It also increases VM freeze time because it waits for all the pending I/O transactions to complete.

- **Returning the completed I/O transactions** – alternatively, MP_{M1} waits for the pending transactions to complete while the VM is being migrated. It receives all the completed transactions and sends them to the new Mid-Point (MP_{M3}), which in turn returns them to the DD. This approach causes least increase in VM freeze time because VM migration and completion of pending I/O transactions continue in parallel. We use this approach for block devices in this chapter.

3.2.2 Device Hot-Swapping

Device hot-swapping permits a VM to dynamically re-wire its virtual to physical device connections while the device is in operation. This is useful in multiple circumstances, e.g., fault maintenance of the device/machine, run-time performance management (e.g., the new device performs better than the old one), or a hot-swap required because of a higher level policy change. This chapter uses device hot-swapping to improve the performance of a running application, where a remote disk being accessed by a VM is hot-swapped with a local disk to significantly improve its throughput and remove network dependence. Such transparent hot-swapping requires that a 'similar' device be used (e.g., a disk with the same content as the original disk), implying the need to efficiently transfer the device contents and state. When the hot-swapping is performed, there could be pending I/O transactions to the old device and it must be ensured that any state changes made by these pending transactions to the old device are reflected onto the new device before it is used. The Netchannel architecture supports multiple implementations of this functionality:

- **Weak consistency** – the MP can access both the old and the new devices during hot-swapping (i.e., while the state changes made by pending transactions are being transferred from the old to the new device). When the complete state has been transferred, the Mid-point stops using the old device and accesses only the new device.

This approach, while requiring smaller downtime in device access, can be hard to implement because the MP may have to implement complex semantics, such as deciding which disk contains the latest data block for a particular read operation.

- **Strong consistency** – this approach requires that the new device be in complete sync before hot-swapping can take place. MP does this by first bringing the physical device into a *quiescent* state. At the start of hot-swapping, MP queues all transactions from the DD instead of issuing them to the device and then waits for all pending transactions to complete. After these are complete, the device is in a quiescent state, and this state is transferred to the new device. Once the new device has been updated, the hot-swap takes place and MP starts using the new device by issuing all of the queued transactions to it instead of to the old device. This is the approach used for disk hot swapping in this chapter.

For devices like NICs, frequent hot-swapping is reasonable due to their small internal states. For disks and similarly state-rich devices, hot-swapping will likely remain infrequent.

3.2.3 Transparent Device Remoting

Although Figure 7 shows the device on a remote machine M2 accessed via a Device Server, device virtualization is not complete without also addressing locally attached devices. In such situations, virtual device migration presents additional challenges because of the lack of a device server since the DV accesses the local device directly via local device driver. Hence, after live migration, the new MP cannot access the now remote device, because there is no Device Server to provide such access. To solve this problem, we have extended the MP module to also provide minimal Device Server functionality. The pending I/O transactions during live migration are handled in the same way as described in Section 3.2.1. In this configuration, after live migration, the new MP forwards I/O transactions to the old MP which passes them to the DV which in-turn issues them to the device (see Figure 9 for a Xen specific implementation). The completed transactions are returned by the old MP to the new one. This essentially provides transparent device remoting (TDR), where a VM accessing a locally attached device can continue to access it after live migration despite the

fact that the device has now become remote.

Apart from supporting live migration for locally attached devices, TDR also provides a common infrastructure for transparently accessing remote devices present on other machines running the same VMM. Since the MP also works as the Device Server, it can export locally attached devices to the DVs present on remote machines which in turn export them to their guest VMs. Here, Netchannel provides the common infrastructure of Mid-points and the communication channels connecting them.

3.2.4 Architectural Considerations

Experiments described in Section 3.5 show the simple implementation of Netchannel to be useful for both disk and USB devices. Several interesting implementation issues arising in those contexts are discussed next.

Communication Protocol. The Netchannel architecture is independent of the communication protocol between the Mid-point and the Device Server. We could use device-specific remote access protocols (e.g., NFS, iSCSI etc. for disks) or more general remote access protocols (e.g., 9P in Plan 9 [61]) with some device-specific extensions. Similarly, for TDR operation, the communication channel between the two Mid-points (acting as client and Device Server) can use one of the protocols described above. In this chapter, we have used a socket-based client-server communication protocol to implement the Netchannel mechanisms, and we are currently working on implementing them using the more general Plan 9 approach.

Latency and Bandwidth. Since network communications introduce additional latency in accessing remote devices, a guest application may timeout on pending device requests during TDR operation. However, we have not observed it for the devices we have evaluated. These include bulk devices like SCSI and USB disks and isochronous devices like USB cameras. For the devices evaluated in our research, mechanisms like buffer caching [20] tend to reduce network bandwidth needs. We also note that future work on virtualized networks will likely address issues like congestion and isolation in the network, thereby providing further support for the TDR methods advocated in our work.

Device Naming and Discovery. Device naming and discovery are essential functions of TDR. Netchannel is designed to use the naming and discovery protocols provided by underlying remote device access protocols. In the current implementation, it uses a combination of IP and device addresses to create unique remote device identifiers. As we move towards using 9P for TDR, we will use its implementations of these functionality.

Device Sharing. The TDR feature of Netchannel can allow the remote device to be shared between multiple VMs. The Mid-point (acting as Device Server) can export the same device to multiple DVs to enable sharing. It must also handle the device-specific sharing semantics (e.g., read-only vs. read-write sharing of disks). The Mid-Point's current implementation can be extended to support device sharing.

Residual Dependencies and Security. During the migration of locally attached devices permitting VMs to access their now remote devices leads to residual dependencies. Hot-swapping can be used to remove such dependencies, which is one reason why Netchannel supports it.

To provide security through authentication and authorization for accessing remote devices, Netchannel depends on the underlying communication protocol between the Mid-point and Device Server. Many existing remote access protocols (e.g., iSCSI, 9P etc.) already provide security measures. Even for the TDR feature, the Mid-Points can use these to provide desired security guarantees.

3.3 Netchannel Implementation for Virtualized I/O Devices in Xen

To demonstrate the ideas presented above, we have implemented Netchannel and its support of TDR in the Xen environment. The implementation not only gives VMs transparent access to remote devices, but also provides for virtual device migration and device hot-swapping for locally attached devices.

Netchannel-Xen is based on the frontend/backend mode of device virtualization [38], as shown in Figure 9. Xen virtualizes I/O devices by splitting the device stack at the 'class-driver' level. Special 'class-drivers' are used in both Dom0 (SVM) and DomU (GVM). The 'class-driver' in DomU and Dom0 are the frontend (FE) and backend (BE), which act as

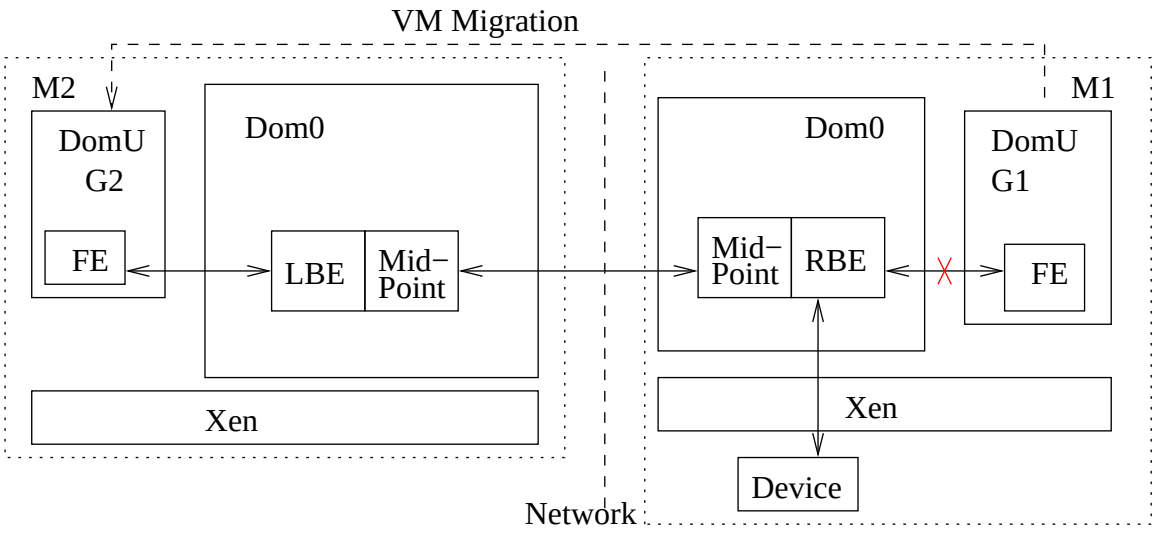

Figure 9: Netchannel Implementation in Xen

DD and DV, respectively. Concrete examples of class-level drivers are block, network, USB, etc. Figure 9 depicts a client-side BE as a local BE (LBE) and a server-side BE as a remote BE (RBE).

3.3.1 Transparent Device Remoting

The MP acting as the Device Server MP_{M1} exports the locally attached device over the network. The connection information for the remote device provided by MP_{M1} is specified in the VM G2's configuration and passed down to MP_{M2} using the *xenstore* utility. MP_{M2} also maintains generic remote device state and any device-specific states LBE may require. Every Netchannel communication over the network, then, is preceded by a Netchannel header. Each such header has a common and a device-specific part. The common part contains information common to all devices, e.g., request-response id, device id, length of the data following the header, etc. Examples of the device-specific parts appear later in this section.

Device-specific actions are linked to Netchannel communications by having device-specific callback functions implemented for each class of device. The following function call is used to register the device specific callbacks by MPs (for both client and server):

```
void register_netchannel_device(struct xenbus_device *xenbus_dev, u64_t
```

```
devid, void *dev_context, netchannel_connect_callback *connect,
netchannel_disconnect_callback *disconnect, netchannel_receive_callback
*receive, netchannel_xmit_callback *xmit, netchannel_migration_callback
*migration, netchannel_hotswap_callback *hotswap);
```

The first four callbacks (connect, disconnect, receive, xmit) provide TDR functionality, and the last two callbacks (migration, hotswap) provide VDM and DHS functionality, respectively.

In Figure 9, FE in DomU G2 on machine M2 accesses the device on M1 as if it were a locally attached device. FE makes requests to LBE via a shared memory communication channel which are forwarded by the MP_{M2} (client) to MP_{M1} (server). The I/O buffers corresponding to these requests are also shared between FE and LBE. MP_{M2} also enqueues this transaction in its list of pending transactions. MP_{M1} causes RBE to issue the transaction to the device. Upon completion of the transaction, it follows the reverse path. Upon receiving the transaction, MP_{M2} removes it from the pending list and LBE returns it to FE.

For efficiency, in DomO, Netchannel communications and the execution of MP client and server callbacks are carried out by kernel-level, per-device receive and transmit threads. The receive thread, for instance, will first perform some checks on Netchannel header, e.g., matching a request response id pair, and then forward the actual data contained in each request to the device-specific callback function for further processing. We next describe the specific Netchannel devices implemented in our work.

Remote Virtual Block Device Access

Xen virtualizes block devices using block BE and FE drivers. Block device specific callbacks are implemented for the MP client and servers. For read requests, the LBE transmit the requests to the RBE via the MP communication channel. For write operations, the LBE additionally sends the blocks to be written. When the request completes, RBE sends the result to LBE (along with any blocks read for the read requests).

Remote USB Device Access

USB devices are virtualized at port granularity, using USB BE and FE drivers. The

USB ports are assigned to guest VMs, and any device attached to a port belongs to the respective VM. When a device is attached to a USB port, the root hub driver notifies the corresponding BE (RBE), which notifies the guest FE about the attach event through LBE. The guest USB driver initializes the remote virtual device via MP channel. The USB drivers in the guest issue requests using USB Request Blocks (URBs), which the FE forwards to the RBE via the LBE. For write operations, it also sends the transfer buffer. For isochronous devices, it additionally sends an isochronous schedule to the RBE. Upon URB completion, the RBE sends the results back to FE via the LBE, along with any transfer buffer.

3.3.2 Virtual Device Migration

Virtual device migration operates in conjunction with live VM migration in Xen. In Figure 9, DomU G1 is migrated from M1 to M2 while it is still accessing the block device. During live VM migration, the *VM save* process on M1 sends G1's configuration and memory pages to M2, and the *VM restore* process on M2 creates a new VM G2 using the configuration and memory pages. When the VM is finally suspended during the last phase of live migration and the channel between FE and RBE is broken, MP_{M1} transfers the list of pending block requests to MP_{M2}. It also starts transferring all completed block requests to MP_{M2} as they complete instead of dropping them. This however, requires that (1) the I/O buffers shared between FE and RBE on M1 corresponding to the pending transactions and (2) the communication channel must be shared between FE and LBE on M2, as well. To implement this sharing, MP_{M1} sends the guest page frame numbers (GPFN) of the shared buffers to MP_{M2} which maps those pages into its own address space, with the help of Xen. Similarly, the virtual interrupt shared between FE and RBE on M1 is also shared between FE and LBE on M2. In addition, MP_{M1} begins to act like a Device Server and services the requests from MP_{M2}. Summarizing, the Netchannel implementation establishes the necessary Mid-Points and communication channels between them, as described in Section 3.2.1.1, using Mid-Point extensions provided by the developer.

After resuming as VM G2, FE makes further block requests to LBE, which are issued to the device via RBE as described in Section 3.3.1. However, since the completed pending

transactions were returned by MP_{M1} to MP_{M2} in parallel to live migration, most of the pending transactions were returned to MP_{M2} by the time G2 resumed. This causes only a small increase in latency for completing pending transactions.

3.3.3 Device Hot-swapping

This section describes an implementation of disk hot-swapping, the purpose of which is to improve VM and system performance. In this scenario, after VM migration, a new disk partition of the same size as the VM's current disk size is created on the VM's new local machine (M2), and the disk is replicated onto this partition over the network. Block level replication is used because disk virtualization operates at that level. The guest VM continues to operate throughout the replication period, of course, since it can continue to access the remote disk. This masks the potentially large disk replication time, which depends on the size of the disk and can be done in phases, similar to the pre-copy phase of live VM migration [16]. Intelligent disk replication techniques [70] can be used to further reduce overheads, but that is not the focus of this chapter.

At the start of hot-swapping, the MP_{M2} places the disk into a quiescent state by queuing the FE requests instead of forwarding them to the RBE. It also waits for all pending I/O operations to complete. The last phase of block replication is completed to address any changes made by the I/O operations that were pending at the start of the quiescent state. At this point, the hot-swap happens, and the virtual device is released from its quiescent state. Henceforth, the LBE makes I/O requests to its local disk instead of sending them to the RBE. The disk stays unavailable to the VM during its quiescent period.

3.4 Netchannel Implementation of Pass-Through I/O Devices in Xen

This section focuses on high performance I/O device virtualization because I/O device performance suffers the most from virtualization overheads [78]. While fully-virtualized VMs use device emulation to access virtual devices, paravirtualized VMs use a split driver stack to access them [21, 78]. Both the techniques, however, require switching of control between the guest VM and the service VM, providing the major source of overhead. For example, the network interface card (NIC) throughput decreases by almost 70% because of

NIC virtualization [85]. Such performance drop may become a bottleneck in the overall VM performance and may not be acceptable to some performance critical VMs. For example, a web server may not be able to perform optimally because the virtual NIC is the throughput bottleneck. Similarly a DB server may have a virtual disk as bottleneck.

The problem of device virtualization scalability is even worse with multicore systems and the consequent increase in the number of guest VMs. Multicore systems, however, are also device rich i.e. they have multiple devices in the platform. For example, modern chipsets even have builtin NICs inside them and there are future plans for integrating GPUs as well. Given this scenario, to improve I/O device performance, a performance critical VM can be given direct access to an I/O device. We term this as *pass-through* access since VMs bypass the VMM to access the device. The VM can access and control the device directly which would provide close to non-virtualized device performance.

There are multiple issues with this approach. First, giving pass-through access of a device to a VM renders the device unsharable for most of the devices, e.g., a NIC device cannot be shared between multiple VMs if they are all accessing it directly. This would break one of the fundamental advantage of virtualization that resources can be shared between multiple VMs. Second, it creates a security and isolation problem in computer systems where an IOMMU [21] in not present. Because of absence of the IOMMU, devices can access any part of system memory. Since the VM has direct access to the I/O device, it can program the device to do a DMA to regions of memory that it does not own, causing a major threat for overall system security. Third, unmodified VMs having direct access to physical I/O devices are 'tied' to the physical machine. As a result, they can not be migrated to other machines because unmodified VMs are not aware of migration and the physical device can not be migrated to the other machine with the VM. For example, when the VM's device driver fails when it attempts to access its device on the destination machine.

The first two issues can easily and much more efficiently be solved by hardware enhancements. For example, regarding the first issue, I/O devices can be enhanced to support virtualization of I/O devices in the device itself. This will enable sharing of I/O devices while still having pass-through access to them. In fact, we are already seeing emergence

of such 'smart' devices in Infiniband controllers and self-virtualized NICs [66]. The second issue can be solved by the IOMMU that is being integrated into the upcoming chipsets by all major manufactures (e.g., Intel, AMD, etc.). Once an IOMMU is in place, devices will not be able to access any arbitrary part of memory, but instead, will have to go through the IOMMU which can be programmed to enforce isolation among different VMs. The third issue of VM migration can be solved by a variant of the Netchannel architecture described next.

Live VM migration with pass-through access requires that during migration, a similar physical device (e.g., a NIC of the same model) be present on the destination host, and that device hot-swapping be performed between the devices on source and destination hosts. This will switch the VM's I/O device from source to destination machine and allow the VM continuous access to the I/O device after migration. The problem with this approach is that because of pass-through access, the VMM is not involved during I/O device access causing the VMM to be completely unaware of the device state. Therefore, during live VM migration, device hot-swapping cannot be performed because the VMM can not transfer the device state to the destination device.

The Netchannel architecture provides a solution to this problem by enabling device hot-swapping of pass-through devices during live VM migration. As described in Section 3.2, the Mid-Point (MP) module inserted between the VM's device driver and the physical device, keeps track of all the device accesses and maintains the device state. With pass-through access, however, the MP cannot reside in the SVM, because the SVM is not involved during pass-through access. Further, the MP cannot reside inside the guest VM because this would require modifying the VM. Netchannel solves these restrictions by running the MP inside the virtual BIOS (VBIOS) area of the guest VM using the *Integrated Device Model (IDM)* described next.

Integrated Device Model The Integrated Device Model (IDM) is similar to pre-virtualization [49] which allows device models (DV module) to run inside the guest VM's address space without modifying it. The device models are run inside 'special' memory areas of the VM, e.g.,

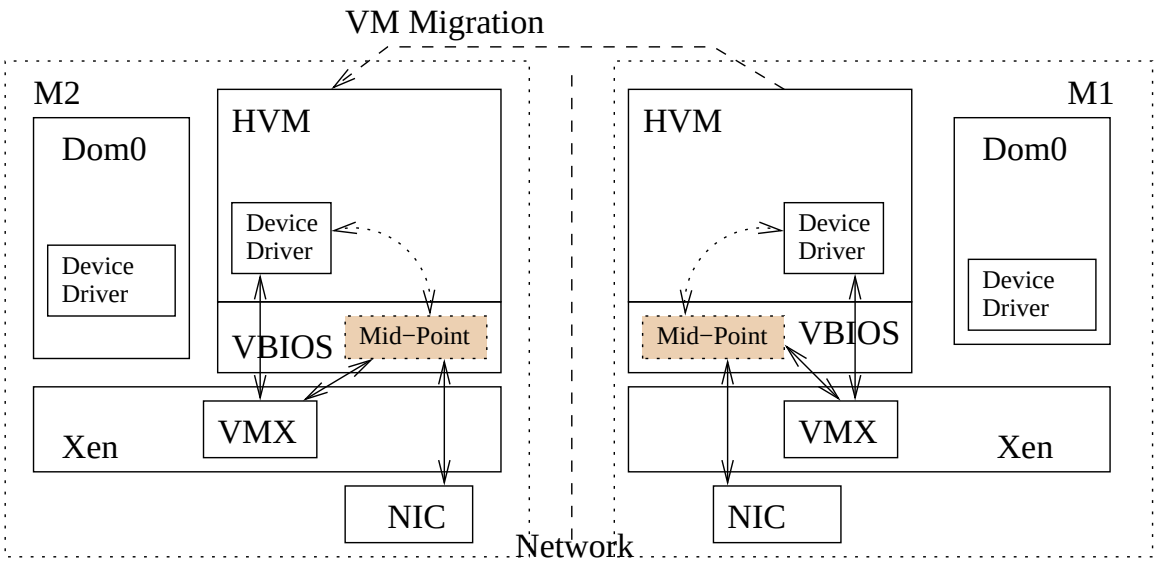

Figure 10: Netchannel Implementation for Pass-Through Devices in Xen

virtual BIOS (VBOIS) area, I/O memory area etc. The advantage is that the I/O virtualization overhead can be significantly reduced because now every device access by the VM can be handled by the VM itself (hence eliminating the overhead of domain switching to the SVM), and in addition, running the device model in the special memory regions keeps the VM unmodified. The device model integrated inside the VM can handle non-I/O requests such as modifying certain device registers, setting up DMA descriptors, etc. For actual I/O requests, the device model can call the DV module inside the SVM which can perform the actual I/O. As a result, the Integrated Device Model enables the I/O virtualization performance of fully-virtualized VMs to be similar to para-virtualized VMs.

3.4.1 Pass-through Access and Device Hot-Swapping of NIC Using IDM

This section describes the implementation of hot-swapping of pass-through devices in Xen using IDM and using a NIC device as an example. Figure 10 shows the implementation of Mid-Point (MP) using IDM and device host-swapping during live VM migration. On host M1, an unmodified VM (hardware virtual machine or HVM) is provided pass-through access to a NIC. However, to enable device hot-swapping, a MP module has been inserted into the VBIOS area of the HVM using IDM. MP runs a *partial device model* which maintains the complete state of the device in software. The state of a device is device specific and

typically consists of values of all the registers of the device, the pending I/O descriptors and any state machine (including device content) the device may have. We only partially emulate it, to keep track of the device's exact state at any point in time. The partial device model intercepts every access to the device by the HVM's device driver and updates its software state before passing on the access to the device. To ensure this, the MP is given direct access to the NIC but the HVM driver is given only virtualized access to it. This is achieved by creating twice the size of NIC's register space. The first half is unmapped and accessible to the legacy driver of HVM while the second half is mapped to the physical NIC and accessible to the MP device model.

When the HVM device driver accesses the NIC through memory-mapped I/O, it generates a VMexit (because the address is unmapped, and control is transferred to Xen. Since the address belongs to the pass-through NIC, Xen returns control to the MP module in the VBIOS area instead of sending the control to the DV module in Dom0. The MP device model updates the software state of the NIC (register values, DMA structures, etc.) according to the access made by the HVM driver and makes the request to the NIC. When an interrupt is generated by the NIC, Xen calls an interrupt handler in the MP device model instead of the HVM driver's interrupt handler. The MP interrupt handler again updates its software state according to the interrupt before transferring control to the HVM driver's interrupt handler. This ensures that the MP device model always has a exact snapshot of the NIC's state.

It is worth noting that inserting a MP module between the HVM driver and the NIC causes performance to drop compared to true pass-through access without MP. However, the experimental evaluations discussed in Section 3.5.4 show that the performance overheads are minimal and NIC throughput and latency are close to true pass-through NICs.

Next we describe the device hot-swapping mechanism during live VM migration. As shown in Figure 10, when the live migration of HVM is initiated from host M1 to M2, the MP module is informed about the event. During the last phase of live migration, the HVM is suspended and the remaining pages of the VM is transferred to M2. A similar NIC (same make and model) is present on host M2. Before the HVM is resumed on M2,

the MP module is notified of the resume event. MP module accesses the new NIC and by configuring it, brings it into the same state as the one NIC on M1. This involves configuring the various registers with the same values as the old NIC and configuring all the pending I/O operations (DMA structures) so that they can be accessed by the new NIC. When the MP module returns from the resume operation, the HVM is resumed. The HVM driver again starts accessing the NIC through the MP and since the new NIC is in the same state as the old NIC, the HVM driver remains completely oblivious to the device hot-swapping during the live migration.

3.5 Experimental Evaluation

This section quantifies the overheads and performance of the TDR functionality of Netchannel implementation in Xen, demonstrates the benefits of VDM during live VM migration, and evaluates overheads and benefits of DHS. Experiments are conducted using two hosts, which are Dell PowerEdge 2650 machines connected with a 1 Gbps gigabit Ethernet switch. Both hosts are dual 2-way HT Intel Xeon (a total of 4 logical processors) 2.80GHz servers with 2GB RAM running Xen3.0. Dom0 and DomU both run a para-virtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution. Dom0 runs a SMP kernel while DomU a uni-processor kernel. Adaptec AIC7902 SCSI (36GB) disks are used for block device experiments. Both hosts have EHCI USB host controllers, and we use a USB flash disk and a USB camera to evaluate remote USB device access.

Experiments are divided into 3 sections. The first section measures the overhead and performance (both latency and throughput) of TDR for both block and USB devices using micro-benchmarks. Throughput is measured with the Iozone [1] file I/O benchmarks for ext3 file systems. The next section evaluates VDM by live migrating a database server in the 3-tier RUBiS application. Finally, the third section shows the effects of disk hot-swapping in the database VM of the RUBiS application from a remote disk to a local disk.

[1] http://www.iozone.org/

3.5.1 Transparent Device Remoting

3.5.1.1 Netchannel Overheads

The inherent overheads incurred by the Netchannel implementation are derived from the need to encapsulate I/O blocks with additional control information. This includes the network header (TCP/IP and Ethernet, 40 + 16) bytes and the Netchannel header bytes. The latter differ depending on the device being accessed remotely, since they also contain some device-specific information. For Block devices, the Netchannel header is comprised of:

28 (request) + 4 (response) = 32 bytes.

On average, one request transfers 4 blocks of data over the network, where block size is 4096 bytes. Hence the overhead for block devices is:

((no of packets * network overhead) + Netchannel overhead)/(total useful data transferred) = (5 * 56 + 32) / (4 * 4096) = 1.904%

For USB devices, the Netchannel header consists of:

16 (request) + 16 (response) bytes.

USB devices transfer data in URB's transfer buffers. For isochronous transfers, there is an additional transfer of the isochronous schedule. Experiments with the USB bulk (disk) and isochronous (camera) has shown that on average, they transfer 3050 and 4400 bytes of data per URB request, respectively. Hence the overhead for USB bulk devices is:

((no of packets * network overhead) + Netchannel overhead)/(total useful data transferred) = (2 * 56 + 32) / (1 * 3050) = 4.721%

and the overhead for USB isochronous devices is:

((no of packets * network overhead) + Netchannel overhead)/(total useful data transferred) = (3 * 56 + 32) / (1 * 4400) = 4.545%

The straightforward computations in this section demonstrate that the additional bandwidth overheads incurred by Netchannel encapsulation are low, never exceeding more than a few percent.

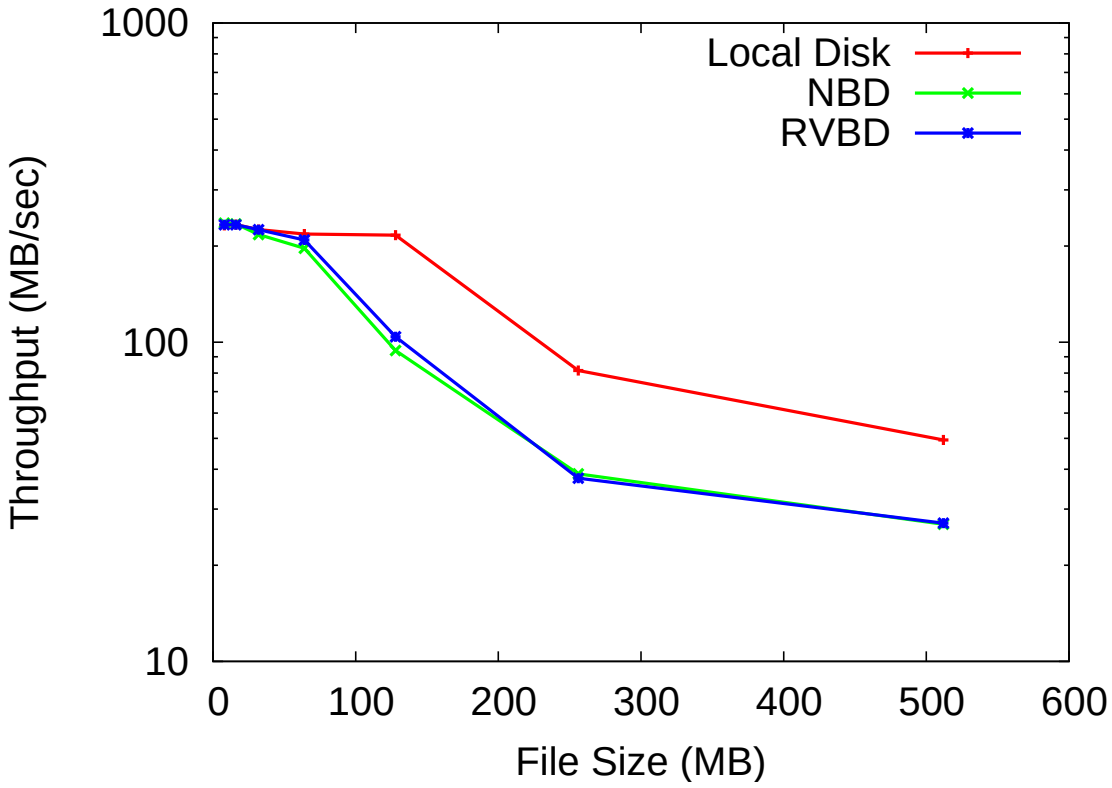

Figure 11: Write throughput of Block devices (record size=8MB)

3.5.1.2 Block Device Performance

To measure the throughput of remote virtual block devices (RVBDs) provided by Netchannel, we boot a guest VM from a remote disk and run Iozone benchmarks. Dom0 is configured with 512 MB of memory while the guest VM uses 256 MB. Read, write, buffered read, and buffered write tests are performed. Iozone performs these tests for different file sizes and different record sizes. We compare the RVBD throughput with locally attached disk (local VBD) and an NBD (network block device) disk which is directly accessed from guest NBD client.

The results for the write tests are shown in Figure 11 for different file sizes with a record size of 8 MB. Throughput is shown on a logarithmic scale. The results for other tests and record sizes show similar patterns and for brevity, are not included here.

Throughput for file sizes less than 64MB is quite high, and it is similar for local VBD, RVBD, and NBD. This is because of buffer caching in the Linux, which caches the files in

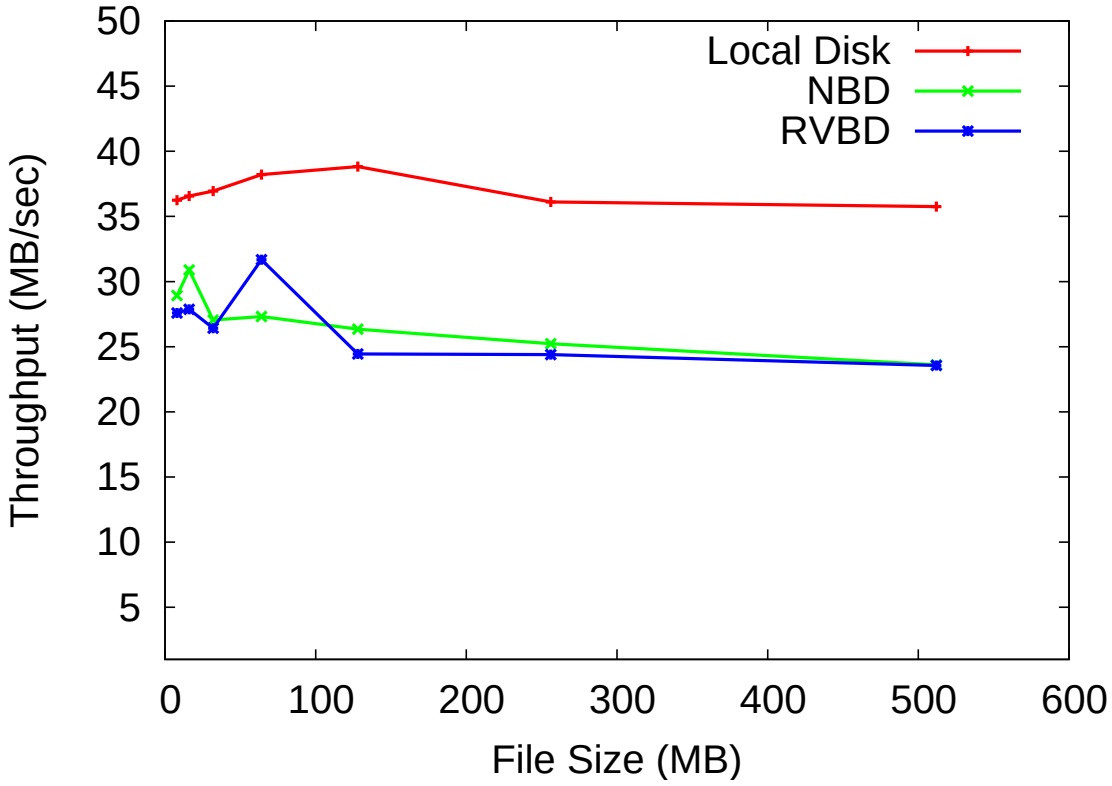

Figure 12: Write throughput of block devices without buffer caching (record size=8MB)

memory and so avoids disk access for subsequent file accesses. However, throughput sharply decreases for file sizes greater than 64 MB, because for bigger file sizes, the buffer cache cannot contain the file, which means that file contents must be pushed to the disk. Hence the actual disk I/O throughput starts to dominate. For the file size of 512 MB, RVBD and NBD have similar throughput, offering about 60% of the local VBD case.

Actual device I/O throughput without buffer caching is shown in Figure 12, on a normal scale. Throughput without caching is drastically lower than that with caching. The performance of NBD and RVBD remain similar for all file sizes, and show the same relative performance compared to local VBD for other record sizes. For the record size of 8 MB, throughput is about 65% of local VBD throughput.

We conclude that RVDB and NBD offer performance comparable to that of locally attached disks, in part due to optimization techniques like buffer caching. This clearly demonstrates the utility of the proposed Netchannel implementation for transparent remote

disk access.

3.5.1.3 USB Performance

Remote USB (RUSB) bulk device experiment uses a USB flash disk with the same experiment setup as in the previous section. The USB disk (a 1GB Sandisk cruzer micro) is attached to the USB port of a machine, and it appears as a local USB disk to the guest VM on the other machine.

Figure 13 depicts the throughput of USB vs. RUSB disks, using a logarithmic scale. The results are from the Iozone benchmark's write test with a record size of 4 MB. Results are very similar to those seen for block devices. For small file sizes, throughput is similar for USB and RUSB disks because of buffer caching. For larger file sizes, throughput decreases sharply because of the actual device I/O. For the RUSB case, the decrease in throughput is higher. For the file size of 256 MB, RUSB throughput is approximately only 48% of USB throughput. Iozone throughput without buffer caching shows patterns similar to those of block devices (see Figure 12) and for brevity, they are not included here.

The Linux USB storage driver used in this experiment does not support multiple outstanding URB requests. Instead, it issues an URB to the USB FE and waits for it to complete before issuing another one. To optimize sequential I/O throughput for RUSB bulk devices, we can use a USB storage driver which supports multiple outstanding requests.

An alternate set of measurements demonstrates the generality of the Netchannel solution, by measuring the TDR throughput of a USB camera as an example of an isochronous device operating over Netchannel. We use a Logitech QuickCam web-camera and measure its throughput for both local and remote access. The results are shown in Figure 14. Since this camera only supports two small image sizes, its bandwidth requirements are modest. The throughput for Netchannel is about 83% of local throughput for image size 320x240 and about 90% for image size 640x480.

The differences in throughput reductions experienced by disks, USB bulk devices, and USB isochronous devices for TDR access demonstrate the fact that actual performance

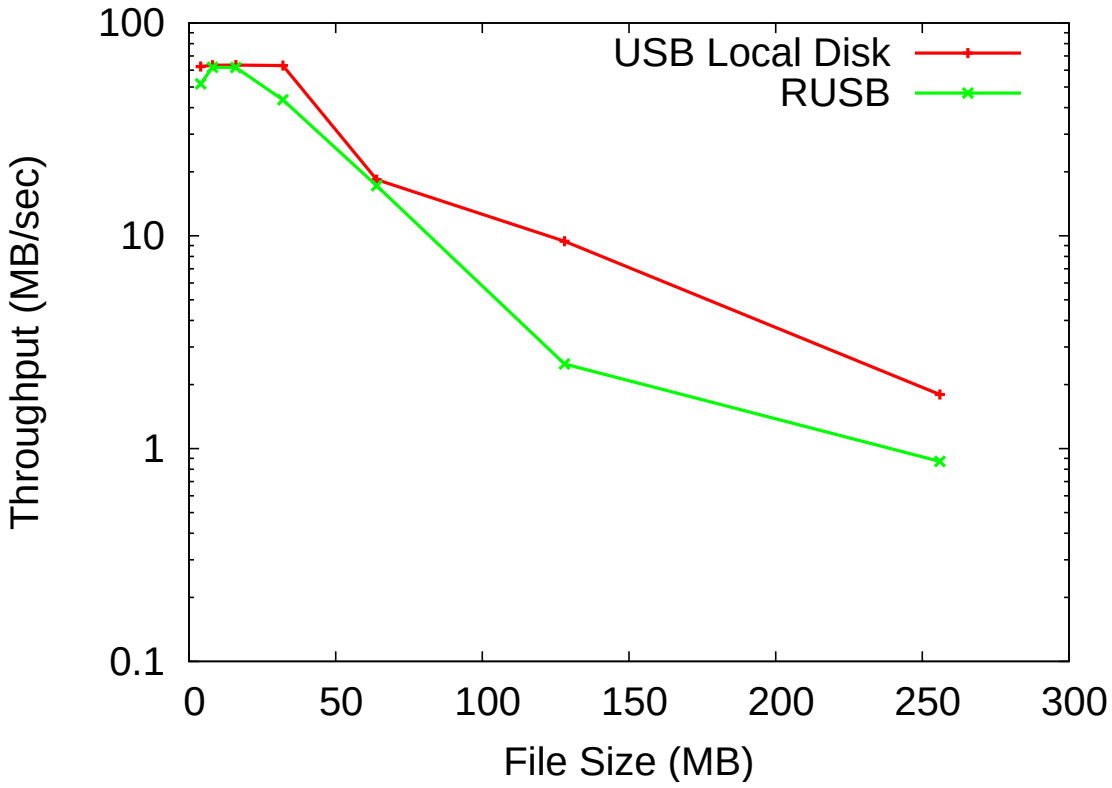

Figure 13: Write throughput of Netchannel USB devices (record size=4MB)

depends on the combination of three principal factors: (1) network bandwidth, (2) the applications' and device drivers' ability to efficiently utilize network bandwidth (by supporting multiple pending I/O requests), and (3) actual device throughput.

3.5.1.4 RVBD and RUSB Latencies

This section contains detailed latency measurements when accessing remote vs. local devices. To minimize TCP/IP buffering latency, we use the TCP NODELAY socket option.

End to end latency encompasses the entire time between the FE sending a request to the device and the FE receiving the device's response. We divide this latency into stages and calculate the time spent in every stage. In Xen, the stages are Frontend (time spent by the FE driver in GVM), Network (time spent by the TDR processing and network propagation delay), and Backend (time spent by BE processing and actual I/O operation). Every stage is measured individually, but we report the combined latency in both the forward (towards the device) and the backward (towards the FE) directions for each stage. Latency is measured

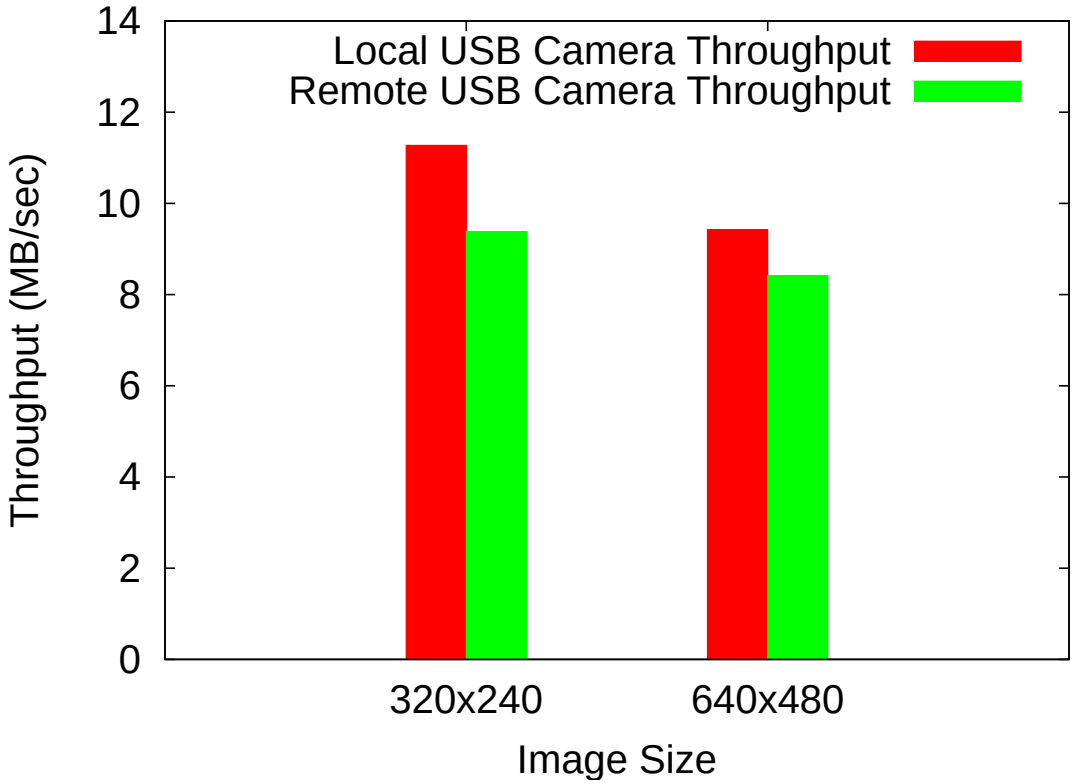

Figure 14: Throughput of remote USB camera using Netchannel

using the sched_clock() function which gives synchronized time in the guest VM as well as in Dom0. To remove the effects of large latencies due to infrequent system activity, we sample the latency for 100 transactions to the device and compute the average.

Figure 15 shows the normalized latencies of accessing 4KB disk block incurred in various stages of local VBD, RVBD, and NBD devices. The total access latency is slightly lower for NBD compared to RVBD since NBD server runs at the application layer and can utilize file system provided caching. It is apparent that the total increase in RVBD and NBD latency are mainly due to additional time spent in the Network stage. The total RVBD access latency is only 25% more than the local VBD.

Figure 16 shows the normalized latency for accessing 4KB data incurred in various stages of accessing local USB and RUSB devices. The normalized latency of USB devices is significantly more than block devices because blocks device drivers transfer significantly more data with every request. The increase in I/O latency for the RUSB case is again due

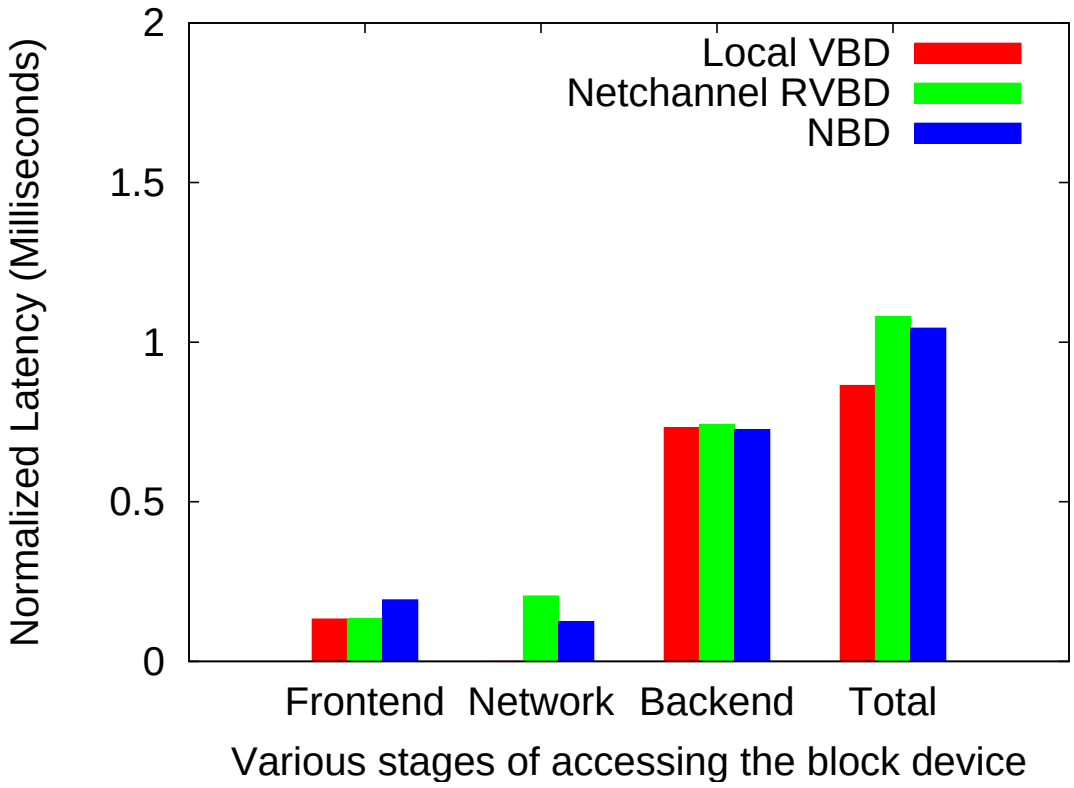

Figure 15: Latency incurred by various components in accessing block devices (milliseconds)

to Netchannel processing and the extra copying of data over the network. The total USB access latency is 66% the RUSB access latency.

We see that for both block and USB devices, the majority of I/O access latency is incurred in servicing an actual I/O request by the device and in case of remote devices, by the network (protocol stack processing and propagation delay) delay. The time taken by the device is characteristic of the device and cannot be improved by software solutions. The latency induced by the network, however, can be reduced by using low latency communication technologies like RDMA, Infiniband, etc.

3.5.2 Virtual Device Migration

A strong advantage of Netchannel is the ability to migrate virtual devices while they are being used. We evaluate the effects of such migration with a multi-tier web application called RUBiS that heavily accesses a disk, and we show how the VM running the application can

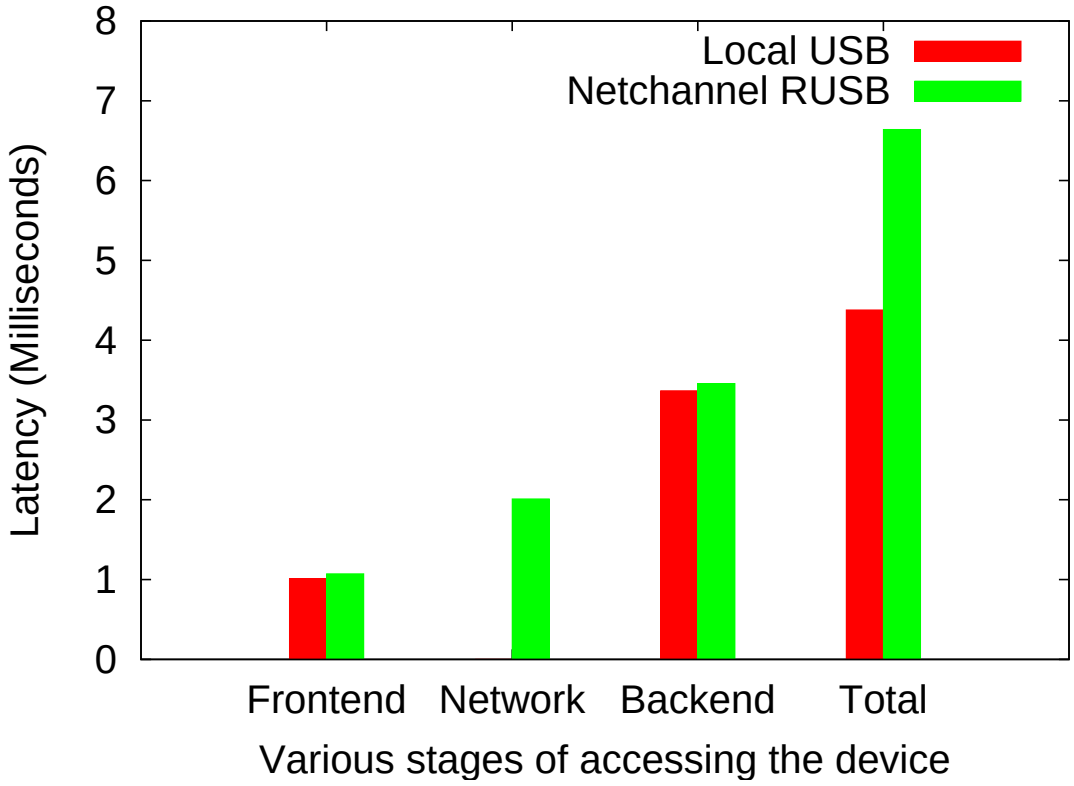

Figure 16: Latency incurred by various components in accessing USB devices (milliseconds)

be migrated seamlessly without any disconnection from the device and hence, without any significant degradation in its performance.

Application Description. Many enterprise applications are constructed as multi-tier architectures, with each tier providing its own set of services. A typical e-commerce site, for instance, consists of a web server at the front-end, a number of application servers in the middle tier, and database servers at the backend [11]. In this environment, it may be desirable to migrate one or more components in a tier to another physical machine for performance (load balancing, etc.) or for maintenance (hardware upgrades, applying software patches, etc.). Experiments with live migration in multi-tier applications use the RUBiS open source online auction benchmark [12]. It implements core functionality of an auction site like selling, browsing and bidding.

Figure 17 shows the basic setup and the live migration of a MySQL server from node A to node B. The workload is generated using *httperf* running on two separate client machines.

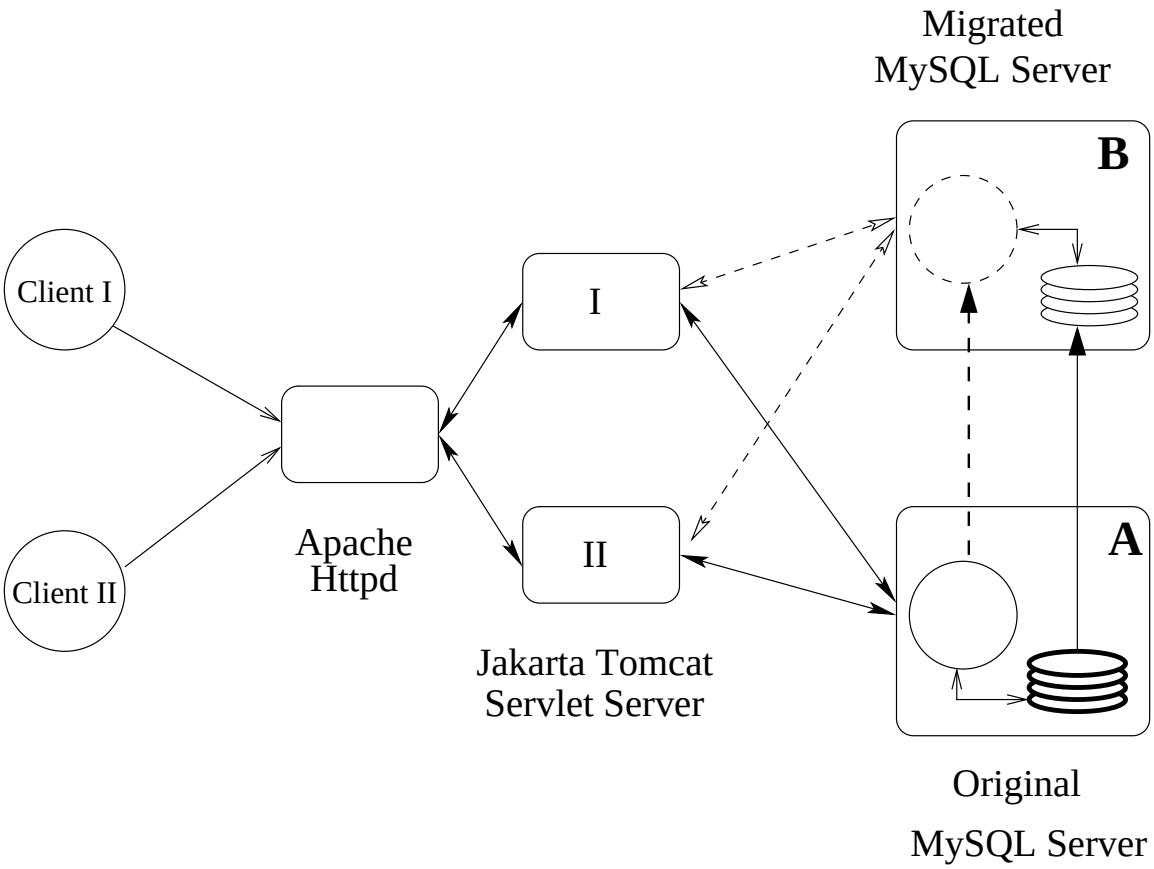

Figure 17: MySQL Server migration in RUBiS

Each of the httperf instances create 30 parallel sessions that issue user registration requests to the RUBiS web-server. The web-server forwards the requests to the two application servers, which in turn communicate with the MySQL server backend. The MySQL server runs in a guest VM and to satisfy client requests, heavily accesses a database of size more than 6 GB which is stored on a local disk. In order to demonstrate the seamless migration of block devices, at some point during the experiment, the MySQL server VM is migrated from host A to host B. For this migration, we evaluate the change in throughput seen by the clients and time taken in handling pending I/O requests. After migration, at some point we perform a disk hot-swapping. The results are shown in Figure 18.

Figure 18 shows the drop in total throughput (measured in response/sec) due to this migration. Note the change in performance in the *pre-copy phase*, when the memory pages of the MySQL server VM are copied to node B. This phase lasts for about 6 seconds and

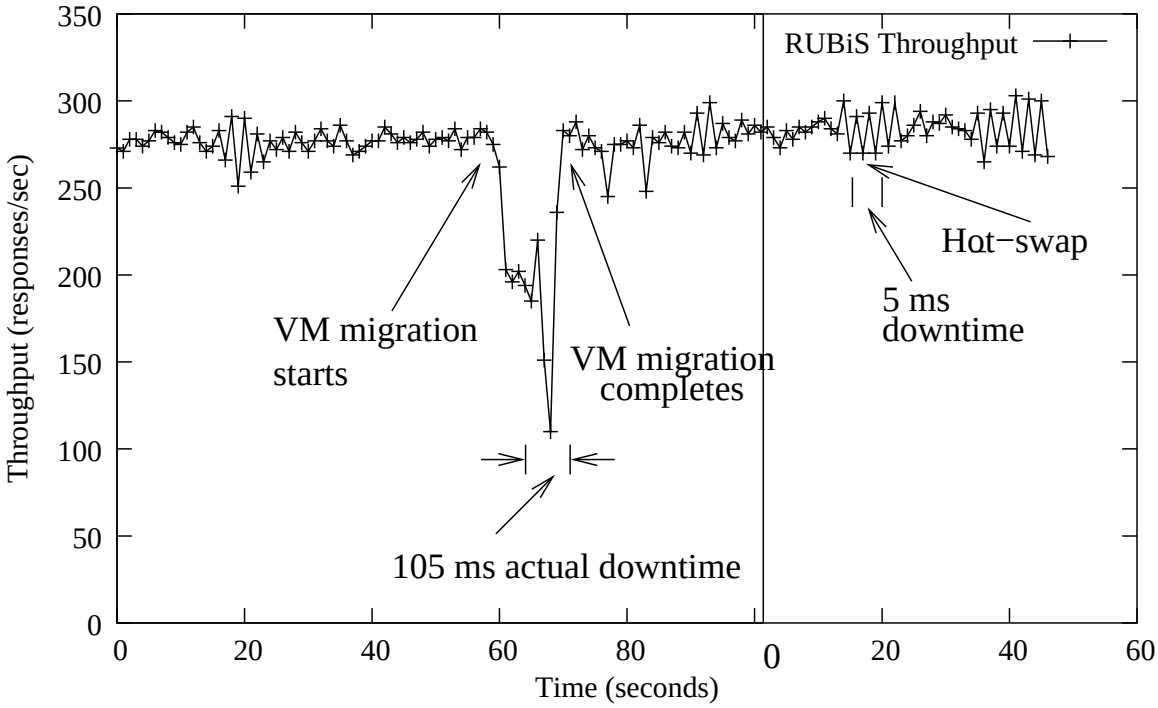

Figure 18: Effect on RUBiS throughput due to MySQL Server migration and hot-swapping

during this phase; the throughput drops by about 28%. In the *stop-and-copy phase*, further throughput reductions are experienced. This phase lasts for approximately 105 milliseconds. It includes the wait time for the Mid-points to handle the pending I/O requests which is approximately 13 ms. This wait time is actually dependent on the number of pending block requests which was on average 6 in this test. These requests are 'returned' to the VM at its new location. We observe that because of virtual device migration, the database server seamlessly accesses the disk remotely.

However, we do not observe any significant reduction in disk throughput. This is because of the buffer caching effect (discussed in Section 3.5.1.2), which causes the throughput for remote disk access (the throughput after "Migration completes" in Figure 18) to be same as the throughput for local disk access (before VM migration).

We further compare the time taken to complete pending I/O transactions by our 'return' approach with the approach where all the pending I/O transactions are reissued again after live migration. Current block FE driver buffers all the pending I/O transactions and when resuming after live migration, it reissues them to the device. We ran Iozone benchmark

Table 2: Time taken in handling pending disk requests during Iozone VM migration

Pending reqs.	Incomplete reqs.	Return time	Replay time
5	0	1.94 ms	23.43 ms
12	0	2.74 ms	40.29 ms
16	1	12.93 ms	45.01 ms
23	5	23.41 ms	81.23 ms
26	2	16.69 ms	85.46 ms
32	7	29.24 ms	96.90 ms

during live VM migration and measured the time taken inside the guest VM. Table 2 shows the comparison of time taken with varying number of pending I/O requests. We observe our 'returning' approach takes significantly less time to complete the I/O operations compared to 'reissuing' approach. This is because in 'return' approach, most of the pending I/O operations are already complete when the VM resumes on the new machine while in 'reissue' approach we have to reissue all the pending operations from scratch and wait for their completion. The column 'Incomplete reqs.' lists the incomplete requests at the time of VM resumption for 'return' approach. We see that the number of incomplete requests is significantly smaller than the total pending requests which make the 'return' approach less time-consuming. By reducing the time to complete pending requests, 'return' approach significantly improve the device availability for the VM during live migration.

3.5.3 Device Hot-swapping

Figure 18 also shows the results of disk hot-swapping of database VM. In this experiment, the MySQL server VM's remote disk is hot-swapped with a local disk while the server is running. However, to simplify disk replication required for hot-swapping, we only make read requests to the database server from the httperf clients. We do block-level replication of the disk used by the MySQL server from host A to a disk on host B using *dd* utility in Linux before starting the experiment.

Interestingly, the throughput does not drop during hot-swapping despite the device going into quiescent state. This is because the device's quiescent period lasts only about 5 ms, which is not noticed by the clients due to buffer caching effects. Even after hot-swapping, there is no change in the client observed throughput. This is again because of the buffer

caching effect (discussed in Section 3.5.1.2), which causes the throughput for remote disk access to be same as the throughput for local disk access.

To measure the throughput effects of device hot-swapping on disk access (without any caching), we use Iozone to measure disk throughput before VM migration, after VM migration and after disk hot-swapping. Iozone is run inside guest VM to measure the disk throughput as seen by the guest VM. The results are shown in Figure 19.

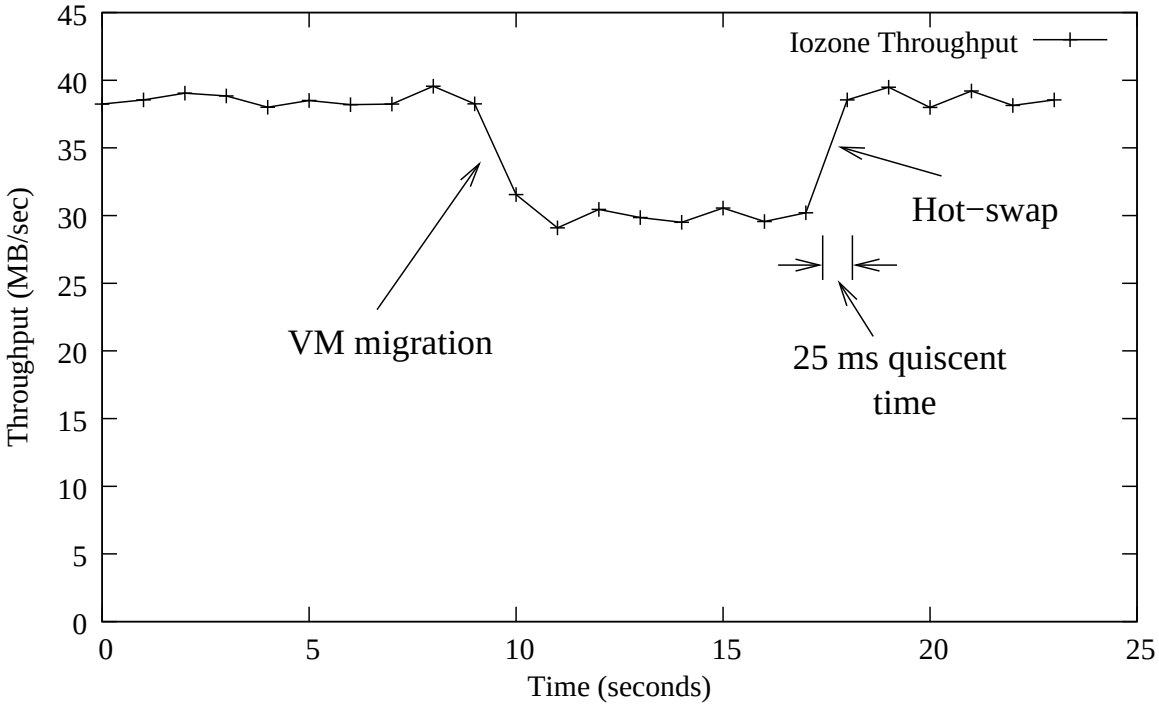

Figure 19: Effects on Iozone throughput due to VM migration and disk hot-swapping (file size = 32MB, test=read, record size = 8MB)

Figure 19 clearly shows the throughput benefits derived from device hot-swapping. The throughput level of Iozone drops after VM migration and is restored after device hot-swapping. The number of pending I/O requests at the start of the quiescent state is approximately 30 in this test, and the measured downtime for the disk because of this state is 25 ms.

3.5.4 Pass-Through Access to NIC

To evaluate the performance of Netchannel pass-through access, we used an IBM ThinkCenter workstation with an Intel Pentium Dual-Core processor with 1 GB RAM and Intel

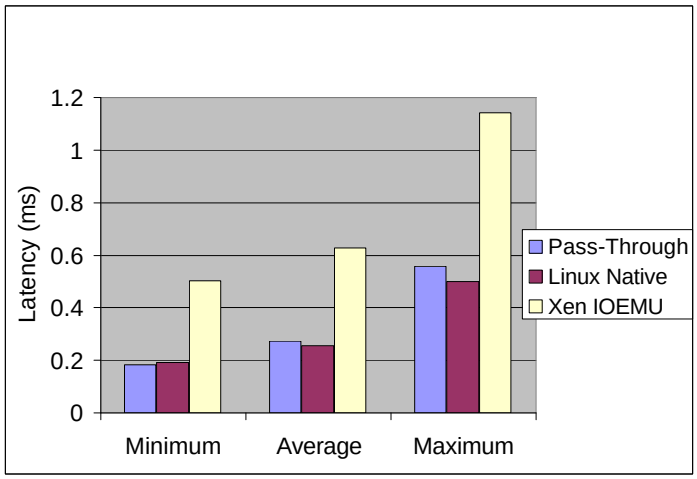

Figure 20: Latency overhead of Netchannel pass-through access and comparison with IOEMU based access

Pro/1000 NIC with i82573 controller. We used the latest version for Xen as of May 2006 and 2.6.16.13 kernel version for Dom0 and HVM. The HVM domain was booted with 256 MB of memory and was given pass-through access to the NIC. Three cases are evaluated and compared; (i) Netchannel's pass-through access to NIC as described in this section, (ii) Linux native access to NIC without any virtualization layer, and (iii) Xen's IOEMU based emulation of NIC. To measure the network performance, we used *iperf* traffic generator with TCP/IP traffic from a non-virtualized machine to the HVM over a gigabit network. To avoid any timing related virtualization, all the timing related measurements were done on the non-virtualized machine.

Figure 20 and Figure 21 show Netchannel's network latency and throughput overheads respectively, compared to Linux native and Xen's IOEMU access. Figure 20 compares the minimum, average and maximum latencies for the three access methods. We see that Netchannel's pass-through access latency is very close to Linux native compared to Xen IOEMU. The average latency overhead for pass-through access is only about 4% compared to Linux native demonstrating the efficiency of Netchannel. Figure 21 compares the throughput overhead of Netchannel's pass-through access to Linux native and Xen IOEMU

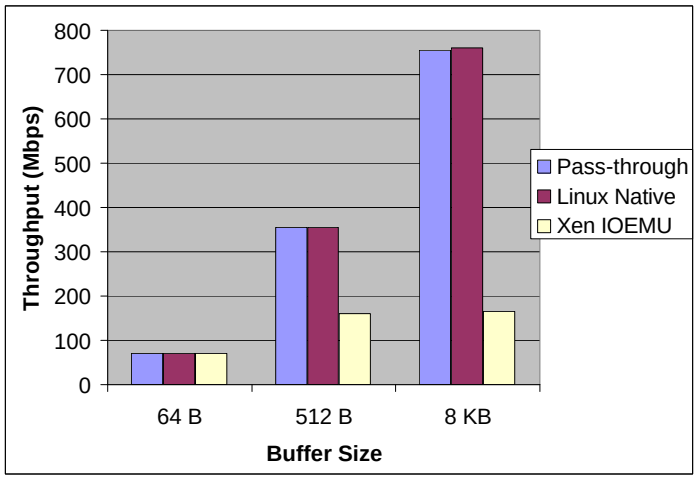

Figure 21: Throughput overhead of Netchannel pass-through access and comparison with IOEMU based access

for different packet sizes. We see that for 64 byte size packets the throughput of all 3 methods is the same because the access overhead is negligible compared to processing time for packets which is the same for all the 3 cases. For 512B and 8KB size packets, however, we see that Netchannel's pass-through throughput is almost similar to Linux native while Xen's IOEMU throughput is significantly less than the other two cases. These results demonstrate that although the MP module in Netchannel creates processing overheads for maintaining the device state of the NIC, these overheads are very minimal compared to true pass-through access.

Next we evaluate the CPU overhead of Netchannel's pass-through access of NIC devices. The experiment is the same as the throughput experiment. Figure 22 shows the normalized CPU overhead (CPU cycles per Mbps of throughput) of pass-through access and compares it to Linux native and Xen IOEMU. We see that for all packet sizes, pass-through access has significant CPU overhead compared to Linux native (39% more CPU for 512 byte packets) but still consumes significantly less CPU than Xen IOEMU (62% less CPU for 512 byte packets).

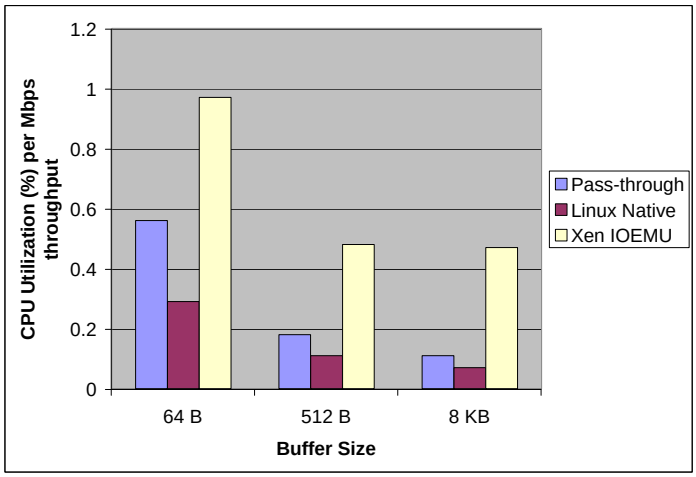

Figure 22: Normalized CPU overhead of Netchannel pass-through access and comparison with IOEMU based access

3.5.5 Evaluation Summary

These results show that Netchannel provides an efficient and low overhead mechanism for providing transparent device remoting of locally attached devices for virtualized access. Further, They show that for complex applications like RUBiS, virtual device migration via Netchannel is not only seamless and transparent to guest VMs, but is also performance transparent. Further they show that device hot-swapping requires un-noticeably small downtime but improves true device throughput (without caching) significantly. They also show that Netchannel provides a low overhead mechanism for pass-through access to I/O devices while still be able to migrate VMs by using device hot-swapping. These features are attained without the need for any special hardware.

3.6 Related Work

There has been significant work on live VM migration support in various VMMs, including VMotion in VMWare's ESX server [84] and live migration in Xen [16]. According to our knowledge, however, these solutions still do not handle pending I/O transactions during live migration and hot-swapping, instead relying on the error recovery mechanisms of

higher layer driver stacks. Further, these solutions do not handle locally attached devices, instead relying on costly 'networked' solutions (e.g., SAN or NAS for storage devices). The Netchannel architecture improves these VMMs with VMM-level handling of I/O devices during live migration and with additional migration support for locally attached devices.

Nomad [32] enables live migration of VMM pass-through network devices, depending heavily on the intelligence in the Infiniband controller. Netchannel's more general approach depends only on the underlying device virtualization layer, thereby addressing arbitrary devices. Previous work on hot-swapping [6, 64] has targeted the run-time exchange of software modules or objects, rather than the physical devices addressed by our work.

Multiple protocols exist for remote device access. There are device specific solutions like iSCSI, NBD, and NFS [72] for storage,more general remote access protocols like 9P in Plan 9 [61], and device access via web services. The Netchannel architecture can utilize any of these solutions and maintain their device states. Netchannel differs in that it does not depend on the network stacks in the corresponding guest OS for remote device access, and therefore, does not have the guest OS dependencies present elsewhere. Instead, Netchannel relies only (1) on the device virtualization (DV) module and (2) on network stack being present in the VMM.

Finally, our TDR functionality is similar to methods like USB/IP [29], which enable remote access to USB devices. The comparative advantage of the Netchannel architecture is that it utilizes the pre-existing device virtualization (DV) module to provide Device Server functionality instead of demanding the creation of an entirely new solution. One view of the TDR functionality, therefore, is that it generalizes USB/IP to work with arbitrary devices and in addition, supports live VM migration.

3.7 Summary

This chapter presents a general mechanism, termed Netchannel, to provide *location transparency* for I/O devices in virtualized environments. Netchannel enables transparent (i.e., not visible to guest operating systems) and continuous access to I/O devices during live VM migration for both locally attached devices and networked devices. Netchannel also

supports seamless hot-swapping of devices and provides a common infrastructure for transparent device remoting of locally attached devices. Using the Integrated Device Model, Netchannel enables live migration of unmodified VMs having pass-through access to I/O devices by doing device hot-swapping along with VM migration. Particular advantages of Netchannel are its guest OS-agnostic approach and its independence of potentially costly hardware support for remote device access.

To demonstrate Netchannel functionality for virtualized devices, it is implemented in Xen for block and USB devices. Virtual device migration, when applied to complex multi-tier web services like RUBiS, cause little or no effects on application behavior and performance. In addition, for I/O-intensive applications, virtual device migration is supplemented with device hot-swapping, to remove negative performance effects due to remote device access. Finally, for remote access to block and USB devices in Xen, the Netchannel performance seen is similar to that of local devices accesses, because of caching effects. To demonstrate Netchannel functionality for pass-through devices, it is implemented in Xen for NIC. Results show that Netchannel overheads are minimal making the NIC perform very close to unvirtualized access.

Netchannel significantly improves the manageability of virtualized systems by enabling live migration of VMs using virtualized as well as pass-through devices. However, VM migration itself is insufficient to improve overall management of virtualized data-centers. This is because, there are multiple management solutions in modern data-centers operating in 'silos' which do not coordinate with each other. This lack of coordination results in inefficient management actions, e.g., inefficient VM migration decisions. A generic coordination architecture is required to help these management application coordinate with each other to take more efficient management actions, resulting in overall improvement in manageability and reduction in development and administration cost of management applications. The next chapter describes such a coordination architecture, termed *vManage*, and using this architecture, provides a solution to one of the most important management problems in virtualized data-centers.

CHAPTER IV

vMANAGE: COORDINATED CROSS-LAYER MANAGEMENT IN VIRTUALIZED SYSTEMS

The previous two chapters describe Sidecore and Netchannel, two VMM-level techniques to enhance the virtualization of computational cores and of I/O devices in future many-core systems. These techniques provide efficient methods for utilizing cores and I/O location transparency, both of which enhance the flexibility and manageability of these resources. This chapter describes the logical next step in this research, since exploiting such improved flexibility and manageability is difficult without also making it easier to actually manage VMs and the virtualized platforms on which they run. Further, to demonstrate scalability, we go beyond single platforms to entire blade servers and ultimately, to virtualized data-centers.

The specific technical contribution provided in this work follows a vision in which larger systems are managed by coordinating a federated set of management entities, across the different levels of abstraction of hardware, to VMM, to VM, to applications, and across different subsystems, such as platform- vs. server-level controllers. The vision is both practical, in that current management systems are naturally structured in this fashion, and it is scalable, in that centralized control cannot scale to continuously manage the hundreds of thousands of manageable entities (e.g., VMs running on cores) present in large-scale data-centers. Instead, we design and implement a software architecture for *coordinated management*, termed *vManage*. By using vManage's approach and mechanisms, multiple management solutions in a data-center can coordinate with each other to take more efficient management actions, thereby reducing the overall cost of management. Specific use cases explored and evaluated with vManage leverage the improvements in flexibility and manageability attained with our earlier work, by providing a solution for coordinated VM and host provisioning, including performing VM migration so as to take into account both

VM-level and host-level attributes and requirements.

4.1 Background

The effective use of IT infrastructures strongly depends on easily and efficiently managing server hardware, system resources, and applications. However, rising complexity and scale in today's enterprise data centers has led to increased costs for management, and in some cases, it consumes the largest fraction (60-70%) of IT budgets [52]. Additional challenges are introduced by the adoption of virtualization in enterprise systems, which enables new levels of flexibility like dynamic VM migration [16] and other runtime changes in mappings between virtual and physical resources [44]. 'Cloud computing' and its abstractions further extend the domain [3].

Reacting to these trends, a spectrum of management solutions has been developed and deployed, e.g., the range of operations required to maintain the whole system through its lifecycle phases [39] – bring up, operation, failures/changes, and retirement. These solutions can be broadly classified as providing platforms management, virtual machine (VM) management, or application management. Examples include power and thermal management at the platform level [65, 53], VM provisioning and migration at the VM management level [83], SLA (service level agreement) management at the application level [46, 15], and resource provisioning for multi-tier applications [98, 88, 37]. Table 3 summarizes the scope and examples of these different classes, where it is critical to realize that each of its entries represents substantial industry investments in improving the ways in which platforms, systems, and applications are managed.

Two basic issues with current management approaches and solutions are (1) the presence of *solution silos* and (2) the resulting autonomous operation of methods used within such silos. Concerning (1), while each of the individual solutions shown in Table 3 provide improved and required functionality, the autonomous actions taken by each can lead to ineffective management in today's increasingly complex data centers. Concerning (2), it is neither desirable nor viable to integrate the rich and useful management solutions already present in each such silo, for practical reasons like the need for concurrent development

Table 3: Management Classification

Mgmt. Class	Function	Examples
Platforms Mgmt.	Manage hardware resources	Server config, HW monitor, Power,thermal mgmt.
Virtualization Mgmt.	Manage VM resources	VM provisioning, runtime monitoring, retirement,
App/Service Mgmt.	Manage application resources	SLA management, patch updates, backup

and progress and for theoretical reasons like the difficulty of finding effective and general methods for controlling multi-layer systems and applications [23].

vManage is a VMM-level architecture and approach to managing complex systems and applications. It directly addresses the two issues raised above, by offering support at both the levels of mechanism and policy:

1. *Coordination Architecture and Mechanisms:* vManage provides a simple set of mechanisms for coordinating, rather than integrating, the actions taken by different management layers and methods. Coordination policies are embedded in *mediation brokers* and *management VMs*, and they are enabled by *coordination channels*.

2. *Dynamic Coordination Assessment:* vManage also offers a rigorous approach to assessing coordination actions before carrying them out, the objective being to carry out such actions only if they are likely to have their intended effects. For example, while doing VM migration, perhaps to reduce power consumption via consolidation, the dynamic assessment of future resources available on a target platform is used to prevent undesirable action effects like VM 'ping-ponging'.

vManage's VMM-level support for lifecycle management enables practical approaches to coordinating across different management subsystems – via its management architecture. At the same time, vManage does not impose the disruptive requirement to re-architect existing solutions. Its open nature is in contrast to proprietary approaches to interfacing across different management domains [40], and/or the use of system-specific implementations which often limit desired functionality extensions or generalizations. The HP PowerRegulator for example, implements power management in the firmware of the processor and has no

feedback about how its power management decisions are affecting the SLAs of applications running on top of it. vManage, in comparison, encourages solutions in which coordinated cross-layer management can take into account VM, application, and host metrics.

vManage leverages rich prior work in the domains of adaptive and autonomic systems [75, 39], but implementation uses a light-weight approach suited to the VMM and system levels it targets, perhaps even enabling future hardware support for select vManage functions. vManage makes the following technical contributions:

- vManage extends the design of virtualized systems to achieve cross-layer, coordinated management. Extensions include (i) the introduction of management coordination channels to define seamless communication between management layers abstracted from underlying implementation and platform details, (ii) mediation brokers that define policy managers to perform coordination across layers, and (iii) a per platform management VM (MVM) responsible for overall management and coordination of the system.

- A case study application of vManage provides effective methods for VM placement and dynamic provisioning in data centers, considering both VMs' SLAs and platform metrics (power, reliability). For such a vManage application, policies using vManage's methods for dynamic action assessment are shown to satisfy the key constraint of stability desired for online management methods.

- A prototype open source implementation of vManage realized in the Xen environment offers channel libraries and policy modules embedded in a separate management VM, *Dom-M*, which is also enabled as a virtual appliance. Evaluation of sample vManage policies for power and fault management show improved management & data center efficiency compared to the state of the art. For example, it shows that novel SLA based power management policies enabled by vManage provide better power savings compared to traditional utilization based policies. Microbenchmark studies and qualitative evaluations show that vManage's proposed design extensions to virtualization infrastructures show improved benefits at acceptable overheads.

Finally, vManage addresses known challenges associated with system management including (i) dealing with heterogeneous management entities across hardware/firmware, VMM, and guest VMs, (ii) dealing with insufficient privileges and conflicting management actions of VMs, and (iii) lack of visibility into the SLA and performance requirements of the VMs' applications at the management hardware layer. Going beyond such mechanism-level contributions, vManage's dynamic action assessment constitutes a first step toward providing ways in which semantic issues may be addressed when faced with coordinating different management policies.

This chapter demonstrates concrete benefits derived from using vManage's coordination approach to system management. Coordination across solution silos can avoid redundancy and conflicting actions. Power management provides a concrete illustration of this problem, where management agents in guest VMs (e.g., the Linux ondemand governor [53]) react to local resource usage and make changes to actuators (e.g., CPU power states) associated with virtualized hardware. The virtualization layer may in turn, perform its own power management across a collection of VMs, for example, by migrating virtual machines to reduce power. At the physical platform layer, hardware controllers may manage power based on aggregate information visible at that layer. As shown in this chapter, with vManage, algorithms can be realized that properly coordinate such actions, for cluster-size and in the future, data center-size systems. Similar scenarios can be identified for other management applications.

Specific performance results attained with vManage in the context of VM provisioning show benefits that include the following: (1) SLA-based power regulation enabled by its cross-layer coordination obtains improved (about 8% more in small-scale experiments) power savings compared to traditional utilization-based policies, (2) novel VM migration methods implemented with vManage can resolve conflicts between solutions for SLA management and power capping, where (3) its dynamic assessment algorithm is shown to make stable VM migration decisions which in turn, reduces the number of VM migrations, and (4), novel functionality of VM provisioning based on reliability metrics improves overall VM availability through VM migration.

Table 4: Cross layer management approaches

Approach	Characteristics	Limitations
None	No cross-layer coord.	Ineffective mgmt., redundancies
Manual	Multiple console views for mgmt info, Admin executes policies	High latency, high cost
Ad-hoc	Point solution, Implementation and platform dependent	Non-scalable/portable, complex, costly
Structured	Built-in infrastructure hooks, automated	Initial dev. cost community adoption

The remainder of the chapter is organized as follows. Section 4.2 describes the approach and detailed design of vManage. Section 4.3 presents a case study application. Section 4.4 describes the implementation in Xen. Section 4.5 provides evaluation of the vManage mechanisms and its application to the case study. Section 4.6 discusses the related work, and Section 4.7 summarizes the chapter.

4.2 vManage: Architecture Design

4.2.1 Basic Design Elements

Existing deployments providing cross-layer coordination, if at all, are either manual-based or use ad-hoc approaches. Table 4 summarizes the characteristics of these approaches. As seen, these approaches when applied to the development of coordination engines lead to inefficiencies and high costs. For example, a coordination policy engine developed using an ad-hoc approach would hard-code the discovery of individual management entities, require intricate knowledge of the specific data collectors and actuators in the system along with their access method, require awareness of platforms implementation details, and would lead to tightly integrated coordination policies. With the diversity in data collectors, information delivery mechanisms, policy implementations, platforms, virtualization & applications implementations, such existing approaches would not scale, are not portable, and introduce complexity.

More importantly, given the vast benefits possible from cross-layer coordination, such approaches present a hindrance towards wider adoption and development of coordination

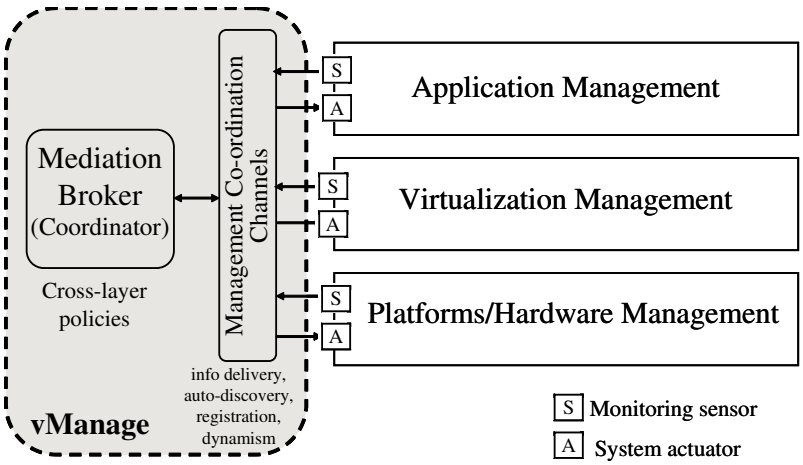

Figure 23: Conceptual view of cross-layer coordinated management. Coordination channels and mediation brokers represent the vManage approach.

engines in the systems and resource management ecosystem.

To solve these problems, vManage is designed as a management architecture extension to virtualization infrastructures. The goal is to simplify the creation and execution of *cross-layer coordination* methods that operate without compromising the functionality of individual management layers or components.

The vManage design presented next has the following additional important properties: (i) it enables coordination without exposing internal implementation details across individual layers; (ii) it is sufficiently general and efficient to be able to deal with the range of management actions relevant to the VMM, platform, and application layer, as well as being able to deal with dynamic behaviors like VM migration; (iii) it runs in isolation without noticeable effects on the base VMM and its operation; (iv) it is easily deployed, including dynamic deployments and upgrades; and (v) it supports heterogeneous execution environments (hardware, software, virtualization) with well-defined APIs.

Figure 23 shows the conceptual picture of the vManage cross-layer stack, depicting the following major elements:

- *Management Coordination Channel* is an abstraction layer for seamless exchange of management information across hardware-virtualization-application layers, hiding implementation and platform details. Operationally, channels connect to the sensors and

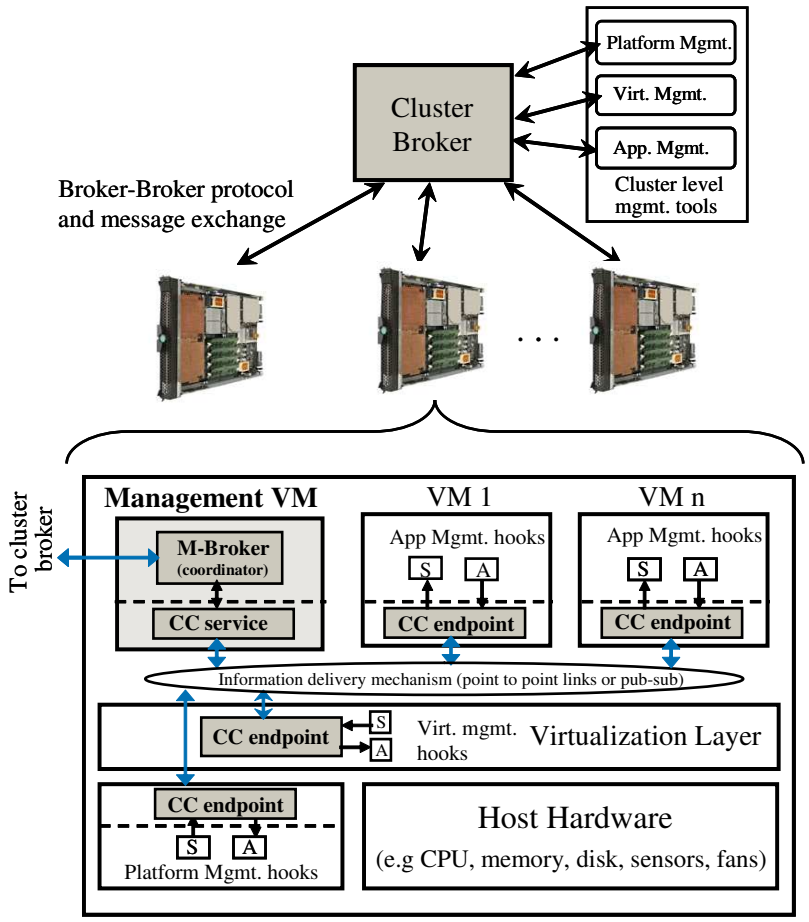

Figure 24: Extending virtualized system for cross layer management. Shaded portions represent vManage components.

actuators of interest in these layers, and they are virtualization-aware in their ability to automatically handle events like VM migration (i.e., via automatic disconnection from the old host and reconnection to the new one). [1]

- *Mediation brokers* implement aggregation and coordination policies, using coordination channels to transport monitoring information and to convey actuation decisions. A *coordinator* instantiated as such a broker can focus on policy decisions within a well-defined coordination structure and framework.

[1]For brevity, we henceforth refer to management coordination channels as 'coordination channels'.

4.2.2 Realization in a Virtualized Environment

Figure 24 depicts the realization of vManage in a virtualized environment. Each physical node has management entities dealing with platform, virtualization, and application management. The key per node element in vManage is the *coordinator*, represented as *M-broker* in Figure 24. This entity exists in a privileged execution domain, termed *Management VM* (MVM), running at higher levels of trust than guest VMs. Each per-node broker can interact with a higher level broker responsible for coordinating among cluster management tools, termed *Cluster Broker* in the figure.

4.2.2.1 Coordination Channels

Coordination policies run by brokers benefit from the MVM-resident service for channel use and establishment, termed *CC service*. This service, coupled with *CC endpoints* embedded in participating management entities, is used to discover entities, create and maintain coordination channels, and ensure their seamless use. More precisely, *CC endpoints* are passive entities running within management domains (platforms, virtualization, application). Each CC endpoint keeps track of the sensors and actuators present in its domain, and it interacts with both using sensor- or actuator-specific protocols. After configuration, the CC endpoint is available to receive calls at a well-known port. The *CC service* is started in the management VM and is responsible for providing the following services to the broker by communicating with the CC endpoints: (i) *Discovery* of CC endpoints across the platform, virtualization, and application layers, by communicating with the well-known port of each CC endpoint and using standard discovery protocols; (ii) *Meta-data registration* from available CC endpoints in the various management domains, thus allowing the broker to be aware of the type and structure of information available across the various management entities; (iii) *Information gathering and transfer*, which involves the actual transfer of monitoring data and actuation commands; and (iv) *Dealing with dynamics*, which is a key attribute of the loosely coupled nature of vManage. This includes dealing with dynamic additions of VMs, hardware updates, dynamic application deployment, and VM migration.

Dynamic additions and removals are handled by periodic discovery and lease-based registrations. VM migration is supported by dynamic disconnection and reconnection between VMs and MVM.

The choice of physical communication media in implementing coordination channels varies. For example, communication with platform hardware elements will likely employ a driver that uses shared memory communication, while communication between per-node broker and cluster broker will employ message passing. The CC service determines the specific information delivery and physical channel during the endpoint discovery process by querying the endpoint. Finally, the CC service exposes a well-defined set of APIs to the broker. Also, while not currently fully completed, the final coordination channel implementation will use and extend standard protocols provided by WBEM [90] so as to use its rich services like naming, discovery, registration, and security.

4.2.2.2 Mediation Brokers (Coordinator)

M-brokers benefit from coordination channels and their capabilities (see Figure 25) to carry out tasks that include discovery, information gathering, and dissemination of actuation commands across the platforms, virtualization, and application layers. Specific examples include gathering power, IPMI [36] sensor, SLA violations, versioning information, etc. Going beyond facilitating such interactions, vManage also provides a structured environment for changing and running coordination policies. Specifically, a policy interface and engine make it easy to plug-in and run specific policies. If a custom policy engine is needed, it can be created by using the provided framework to develop it and then bind it with the CC service libraries to obtain all the CC services, without having to be concerned with implementation and platform details. Finally, as also shown in Figure 25, multiple brokers (e.g., for power, fault management, etc.) can share the same CC service, thus leveraging common elements and avoiding duplication.

The role of a *cluster broker* (CB) (see Figure 24) is interesting because (i) it can run policies that coordinate across multiple per-node brokers, and (ii) it can also interact with and control cluster-level management tools and mechanisms. As a result, the CB runs on a

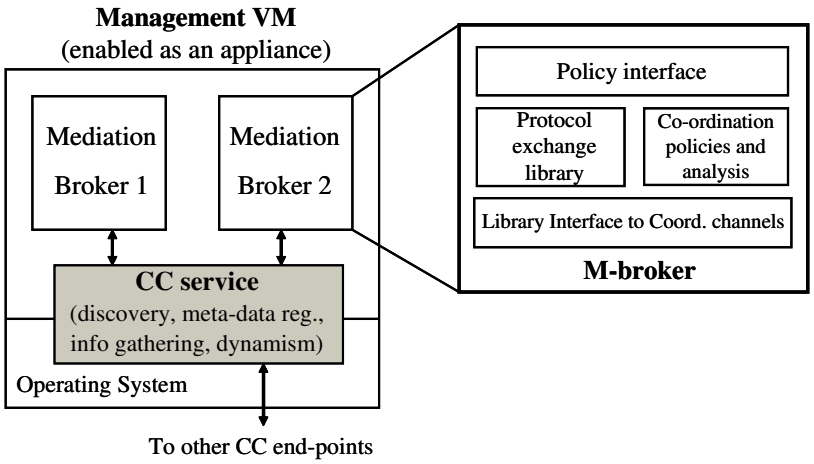

Figure 25: Brokers deployed in MVM. Multiple brokers can share the CC service. Brokers have a well-defined structure simplifying the development of coordination policies.

separate node, called a *Cluster Leader (CL)*. Examples of cluster level platform management tools include HP SIM [30], and those for virtualization level management tools include VMWare VirtualCenter [83].

4.2.2.3 Management VM

The Management VM (MVM) is a privileged VM that is the dedicated point of control for coordinated management tasks on a single host. MVM hosts the broker and CC service and any associated libraries. It has selective privileges to access platforms' hardware including entities such as management processors. Having a separate MVM to run the broker has several significant benefits compared to running the broker inside the VMM. (i) *Reduction in VMM's Trusted Computing Base.* since most of the per-host management code (including various management models, policies, and device drivers) run inside an isolated VM, this reduces the trusted computing base (TCB) of the VMM and in turn, improves its security and reliability. (ii) *Selective Privileges.* Although the broker requires certain privileges for management, this is a subset of the privileges required by other distinguished VMs like Dom-0, ensuring improved robustness to failures compared to solutions in which such functions are integrated into Dom-0. (iii) *Virtual Appliances.* Having a separate virtual machine for management simplifies its deployment and its upgrades, by casting it into the form of a virtual machine appliance. In this form, an MVM can be deployed independently

from the VMM, and it can be dynamically added or removed from a system.

4.3 Using vManage: Case Studies

In this section, we describe the application of the vManage design presented in Section 4.2 for solving a real-world problem in data centers which utilizes the low-level mechanisms developed in previous chapters. We first present the problem and then our vManage-based solution.

4.3.1 Coordinated VM Placement and Dynamic Provisioning

The VM placement problem is to select the most suitable host for a given virtual machine. Traditional solutions like the ones used in VMWare's Virtual Center consider VM metrics such as CPU, memory, network bandwidth, VM priority, etc., but do not take into account platform requirements such as power budget or attributes like platform reliability or trust. Consequently, if a host fails, for example, the availability needs of a VM will not be met, leading to undesirable downtime. Similar issues arise, of course, after initial deployment, which means that runtime management, i.e, dynamic provisioning, is required to ensure VMs' as well as hosts' requirements are continuously met over time.

An integrated approach to solving problems like those described above, would be to create a single infrastructure and tool able to deal with an arbitrary number of metrics, requirements, attributes, and able to deal with/interface with all management subsystems present in the environment. vManage instead, makes the practical assumption that there will be multiple, well-designed and well-tuned methods for performing different management tasks, such as application performance management (e.g., load balancing), virtualization management (e.g., VM scheduling), power management, and others. Each of these management components independently carries out actions that ultimately, trigger certain management events, such as requests for VM migration. vManage, then provides the 'glue' for coordinating such events, to avoid unnecessary workload migration, loss in performance due to unoptimized power-performance trade-offs, and to prevent conflicts created by contradictory decisions made by different subsystems (e.g., migrating a VM onto an idle machine to increase capacity but thereby also increasing power usage). The outcomes are improved

system stability and effectiveness in data center management. Technical specifics in how this is done are explained next.

4.3.2 vManage-based Solution

Overview. Figure 26 shows a vManage-based set of methods whereby the CB assists in making suitable VM placement decisions. It interfaces both with existing cluster level virtualization management solutions (e.g., VMWare Virtual Center [83]) and with platform management solutions (e.g, HP SIM [30]), wherever available. Via its unified monitoring, CB gathers information concerning all relevant requirements and attributes about all of the hosts and VMs in the cluster. It finds a host for the VM by "matching" VMs' and hosts' requirements and attributes so that both are met.

Since VM or host requirements or attributes can change at runtime (e.g., a VM's CPU requirement or host's reliability may change over time), it is necessary to re-provision a VM (e.g., by increasing its CPU reservations or by migrating it to a more reliable host) to continue meeting VM or host needs. This involves message exchanges between MVM brokers with the CB (e.g., to determine a new host) and communications with the MVM brokers who invoke VM migration.

Figure 26 also shows disk reliability brokers hosted in storage nodes. Such a broker can monitor disk for errors, and if errors are predicted to cross reliability thresholds that would make the storage node unable to meet certain VMs' reliability requirements, the broker would pro-actively trigger either a backup or a complete data migration. In the latter case, it communicates with the CB to determine the appropriate disk where the data should be moved, and it then sends commands to available data migration actuators (e.g., Storage VMotion [77], or disk hot-swapping as described in chapter 3), the objective being zero downtime or zero data-loss.

Unified System Monitoring. Coordination requires unified monitoring across the platform, virtualization, and application layers. With vManage's coordination channels, the power and reliability brokers use the CC service to collect monitoring data from the relevant sensors, via CC endpoints. Specific sensors used in our current implementation include

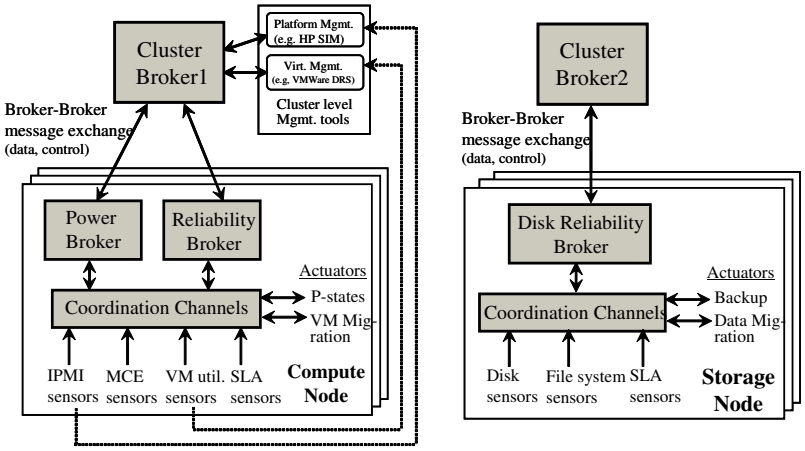

Figure 26: vManage-based Solution for VM placement and dynamic provisioning. The coordination channels and brokers would be hosted in a virtualized environment as described in Figure 24.

the following(also see Figure 26): (i) resource usage sensors in the VMM layer, (ii) power monitoring sensors (IPMI based) in the hardware, e.g., from HP's iLO management processor, (iii) guest VM SLA monitoring sensors, (iv) Processor Machine Check Event (MCE) sensors in the VMM layer [2], and (v) disk S.M.A.R.T. data sensors at the storage node. Since all such data is first collected and processed by per-node brokers, data processing, data filtering, the production of data digests, and similar actions are possible before sharing it with the CB. This distributed architecture also makes the CB more scalable.

Policies. For power management, DVFS and two different power regulation policies are used to evaluate the vManage approach (Section 4.5 for experimental details): (i) the utilization-based policy, similar to the Linux *ondemand* power governor, uses current CPU utilization as an indication of system load; (ii) the SLA-based policy uses SLA violation statistics gathered from the VMs through coordination channels. Whenever a VM experiences an SLA violation, the broker increases the frequency of the CPU running the VM. When the SLA metric drops below a certain threshold, CPU frequency is decreased. Further, whenever the host's power budget is violated because of increased load on the host and local actions (e.g., reducing CPU frequency to reduce power consumption) are insufficient, the MVM broker sends a notification to the CB, asking it to find a relatively less loaded

[2]These sensors use a driver to get the actual data from processor.

host to which one or more VMs can be migrated.

Our ongoing work with reliability management is experimenting with an approach in which the reliability of a host is defined in terms of five reliability states, varying from extremely reliable to extremely unreliable, with different probabilities of failures. This is implemented with a reliability broker that scans the MCE and IPMI sensor logs for any errors (e.g., bus error, fan failure etc.) and uses a reliability model to determine the current reliability state of the system. The model is known *a priori* or learned dynamically using techniques like Bayesian machine learning algorithm [18] or offline monitoring [37]. Accurate predictions of the reliability state, i.e., of impending failures, can be used to pre-emptively migrate VMs to more reliable hosts, using VM-specified and vManage-captured reliability requirements, thus improving overall availability of VMs.

4.3.3 Dynamic Coordination Assessment

A key goal of coordination is to prevent redundant or unnecessary management actions. vManage attains this goal by assessing the effects of coordination before actuation. Specifically, with VM placement and provisioning, the assessment metric is *stability*, where due to the high cost of VM migration, coordination should prevent unnecessary migration, thereby reducing the number of migrations over time. Stated more precisely, when choosing a destination host for a VM (either for initial placement or for migration), dynamic assessment encourages decisions that lead to a more stable system in which newly migrated VMs do not cause further violations on the new host, thereby causing more migrations, etc. Similarly, when choosing a target VM for migration (e.g., because of a host power budget violation), dynamic assessment is used to bring power consumption below the budget with a minimum number of migrations. it may also not be suitable to migrate VMs in the face of some transient SLA or power violations. Hence, the CB's decisions on when, which VM, and where to migrate must go through a dynamic stability assessment in the decision process. Details of the vManange dynamic assessment approach are presented next.

System Model. Dynamic assessment uses a probabilistic model of application requirements and system resources, where VMs' and platform's resource and attribute requirements like CPU, memory, power, reliability, trust, etc. may change over time. For a particular VM, the variation corresponding to each resource or attribute is represented with $\mathbf{v}_i(t)$. The variation of all such requirements $\mathbf{v}_i(t)$ is then represented with $\mathbf{V}$.

We distinguish between resource ($\mathbf{V}^R \subset \mathbf{V}$) and attribute ($\mathbf{V}^A \subset \mathbf{V}$) requirements. A *resource requirement* is used to specify quantities that result in the reduction of such resources available to other VMs deployed on the same platform (e.g., a CPU or memory requirement), while an *attribute requirement* is used to specify properties, numerical or otherwise, that the hosting platform is expected to provide at minimum (e.g., a reliability requirement). The resource and attribute requirements $\mathbf{v}_i(t)$ are modeled as time dependent functions. For a resource requirements, since its variation is not known exactly, we model it using a time varying mean $\mu_{\mathbf{v}_i}(t)$, a standard deviation $\sigma_{\mathbf{v}_i}(t)$, and a probability distribution function $f_{\mathbf{v}_i}(x,t)$, the latter representing the distribution of the resource requirement at any time instant t based on $\mu_{\mathbf{v}_i}$ and $\sigma_{\mathbf{v}_i}$. The attribute requirements are modeled as time varying step functions, and it is assumed that for attributes, one can calculate a binary match or a non-match between a VM and a platform.

The resource requirements ($\mu_{\mathbf{v}_i}(t)$ and $\sigma_{\mathbf{v}_i}(t)$) and probability distribution function (pdf) depend on the type of workload. To find out a typical workload behavior, we analyzed the real requests traces obtained from our industry collaborators [97]. The trace is shown in Figure 27.

In the figure, we see that the request rate varies significantly, but that there is a repetitive pattern in the request rate over a period of 24 hours (1 day). This indicates that we can estimate workload characteristics ($\mu_{\mathbf{v}_i}(t)$ and $\sigma_{\mathbf{v}_i}(t)$) with very good accuracy by monitoring it for a certain period of time.

We also model the resources and attributes, $\mathbf{h}_i(t)$, of the underlying platform, which represent the resource or attribute value of the platform at any time t when no VMs are deployed on the platform. The model of the underlying platform is such that the VM requirement or attribute $\mathbf{v}_i(t)$ corresponds to the platform resource or attribute represented

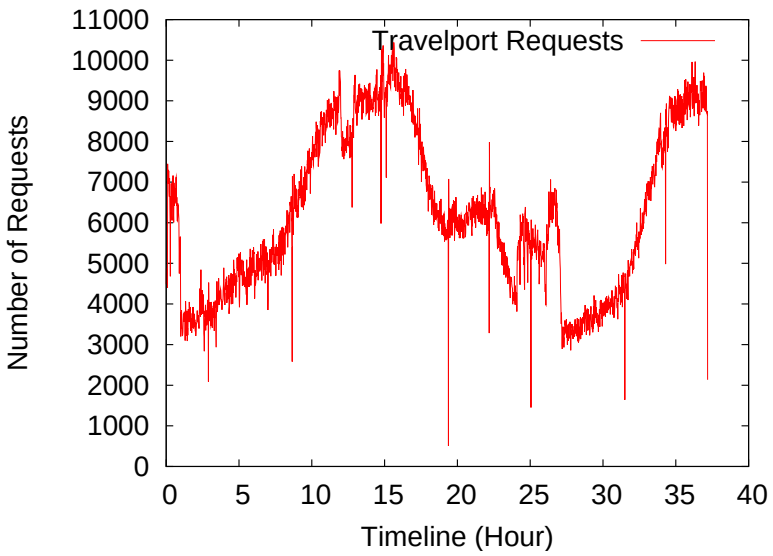

Figure 27: Travelport web trace

by $\mathbf{h}_i(t)$. We model the platform resources and attributes as a function of time because these can change due to events like CPU frequency scaling, failure of a disk in a disk array, etc.

Stability Determination. Dynamic assessment for stability is interested in determining the average probability with which a certain platform can provide sufficient resources to a set of VMs (represented by $\mathbf{L}$), for a given time T into the future, and independently satisfy their individual attribute requirements. To compute this, we start by calculating the cumulative resource requirements for all VMs deployed on the platform. We assume that the resource requirements of various VMs in $\mathbf{L}$ are independent of each other. This is a valid assumption for VMs running independent workloads. This, however, may not be a valid assumption for VMs that are dependent on each other (e.g., a multi-tier application where each tier is running in a separate VM). Under this independence assumption, let $f_{\mathbf{v}_i^L}(x,t)$ represent the pdf for the net requirement of the ith resource by all the VMs in $\mathbf{L}$. Then the corresponding mean $\mu_{\mathbf{v}_i^L}(t)$ and the standard deviation $\sigma_{\mathbf{v}_i^L}(t)$ for the pdf can be calculated by summing the corresponding quantities from the VMs in $\mathbf{L}$.

$$\mu_{\mathbf{v}_i^L}(t) = \sum_{\forall \mathbf{v}_i(t) \in \mathbf{L}} \mu_{\mathbf{v}_i}(t)$$
$$\sigma^2_{\mathbf{v}_i^L}(t) = \sum_{\forall \mathbf{v}_i(t) \in \mathbf{L}} \sigma^2_{\mathbf{v}_i}(t)$$

In the remainder of this section, we assume that the pdf $f_{\mathbf{v}_i^L}(x,t)$, for any time instant t, follows the normal distribution. Now, since the sum of several normal distributions is again a normal distribution [92], the cumulative distribution function for the pdf $f_{\mathbf{v}_i^L}(x,t)$ can be calculated using Equation 1. Note that if $f_{\mathbf{v}_i^L}(x,t)$ were found to follow some other distribution the calculations for the cumulative distribution function may become very involved, and beyond the scope of this paper. The results in Section 4.5 show that the assumption of normal distribution works well for the workloads used in the experiments and shows good results.

$$F_{\mathbf{v}_i^L}(x,t) = \frac{1}{2}(1 + erf\frac{x - \mu_{\mathbf{v}_i^L}(t)}{\sigma_{\mathbf{v}_i^L}(t)\sqrt{2}}) \tag{1}$$

Now, by substituting $x = \mathbf{h}_i(t)$, we can calculate the probability that the net required resources are less than or equal to the resources available with the platform, at any time t. To calculate the average probability $\overline{p}_i(t_0, t_0 + T)$ that the net required resources, over the time interval t_0 to $t_0 + T$, are less than or equal to the available platform resources, we use the following formulation:

$$\overline{p}_i(t_0, t_0 + T) = \frac{\int_{t_0}^{t_0+T} F_{\mathbf{v}_i^L}(\mathbf{h}_i(t), t)dt}{T} \tag{2}$$

However, since the erf (the error function) cannot be evaluated in closed form, we discretize the quantities involved in Equation 1 over the time dimension. This transforms Equation 2 to:

$$\overline{p}_i(t_0, t_0 + T) = \frac{\sum_{t_0}^{t_0+T} F_{\mathbf{v}_i^L}(\mathbf{h}_i(t), t)}{T} \tag{3}$$

Assuming that all the VMs (contained in the set $\mathbf{L}$) have their attribute requirements matched to the platform, then the probability $\overline{P}(t_0, t_0 + T)$ that the VMs will continue to have the required resources can be found using the following:

$$\overline{P}(t_0, t_0 + T) = \prod_{\forall i: \mathbf{v}_i \in \mathbf{V}^R} \overline{p}_i(t_0, t_0 + T) \quad (4)$$

$$= \prod_{\forall i: \mathbf{v}_i \in \mathbf{V}^R} \frac{\sum_{t_0}^{t_0+T} F_{\mathbf{v}_i^L}(\mathbf{h}_i(t), t)}{T} \quad (5)$$

Dynamic assessment: adding a VM to a cluster. To illustrate the utility of the formulation developed above, consider the addition of a new VM to a cluster. In this case, the CB uses these formulations to find the most suitable host for the VM satisfying stability. Specifically, the VM specifies its various requirements in terms of resources (CPU, memory, etc.), and attributes (reliability, etc.). Once the CB determines a host that satisfies these basic resource and attribute requirements, it uses Equation 5 to calculate the probability of the host satisfying the VM's resource requirements over time T in future. If the probability is above a certain threshold, the search ends and the selected host is used to deploy the VM. Otherwise, the search continues and at the end, the host with maximum probability is selected to deploy the VM.

4.4 Implementation

We have implemented vManage in Xen environment to provide a solution for the case study described in Section 4.3. We first describe the implementation of the design components and then of the case study.

Management VM. The management VM (MVM) has been implemented as a separate privileged VM, termed *Dom-M*. Xen has been extended with the notion of a *Dom-M* so that Xen can provided special privileges to it. It has privileged access to selective management hardware, such as HP's iLO management processor. When Xen (or Dom-0) receives requests from *Dom-M*, it authenticates *Dom-M* and after successful authorization, performs the action requested. *Dom-M* hosts the CC service and mediation brokers. The coordination channels support dynamic disconnection and reconnection, so that one Dom-M can be

dynamically replaced with another, transparently to any DomU and to the platform endpoints. This functionality also supports dynamic deployment and is essential for deploying *Dom-M* as a virtual appliance.

Coordination Channels. This is implemented as CC service in *Dom-M* and as CC endpoints in Dom-0 (for virtualization management), DomUs (for application management), and iLO (for platform management). Some of the platform management information is also provided through Dom-0 (e.g., S.M.A.R.T. data is obtained by invoking a driver in Dom-0). Currently, in-house protocols are used for discovery, lease registration, and meta-data gathering.

Multiple underlying physical channels are used for information delivery depending on the nature of communication. (1) For monitoring VM-specific SLA statistics, we use a Xen-provided shared memory channel (Xenbus) over which VMs and *Dom-M* can communicate with each other. Since the coordination channels are virtualization-aware, they automatically handle VM migration events. In other words, whenever a VM migration happens, the CC service in the old *Dom-M* automatically disconnects itself from the VM endpoint, and the new *Dom-M* automatically connects with the arriving VM's endpoint. (2) For performing DVFS and accessing MCE, SMART, and IPMI data, we utilize the drivers already present in Dom-0. The CC service establishes a Xen-provided shared memory channel (Xenbus) with a CC endpoint in Dom-0. The CC service sends requests to perform DVFS, VM migration, and collect MCE/IPMI data to Dom-0 and Dom-0 performs these actions on behalf of Dom-M. These actions can also be performed by directly communicating with Xen, but that requires significant modifications to the MCE/IPMI and DVFS drivers, and we found that communicating through Dom-0 has negligible overheads compared to directly communicating with Xen. (3) For interacting with virtualization management, Dom-M either communicates directly with Xen through the hypercall interface providing it privileged resource utilization data, or it communicates with Dom-0 similar to (2). The implementation also supports network socket communication to interact with cluster brokers.

Mediation Broker. M-brokers are implemented as applications that run inside *Dom-M* and the Cluster Leader. They are multi-threaded processes where one or more threads

perform the monitoring, protocol exchange, coordination, and actuation tasks. Unified management via a single broker has the advantage that various monitoring time constants can be coordinated. For example, the power budgeting thread can coordinate with the power regulation thread to ensure timely handling of power violations. Coordination policies are written using an in-house policy engine template, and they bind to the coordination channels libraries. The cluster broker is similar to per-node broker described above, the difference being that this broker runs cluster-wide policies.

Case Study specifics. For initial VM placement, a new VM specifies its requirements with respect to resources (CPU, memory, network bandwidth, storage capacity) and reliability to the cluster broker (CB). The CB runs the "matching" algorithm which is based on the dynamic coordination assessment algorithm described in Section 4.3.3 and finds a matching host for a VM. Once the decision is made at the cluster broker, it sends a message to the *Dom-M* of the selected compute host to deploy the VM. The *Dom-M* broker asks Dom-0 to create the VM and allocate all its required resources. *Dom-M* is thus doing the additional functionality of provisioning in our implementation.

Dom-M bootstrapping takes place by the host sending a request to a cluster deployment server (currently co-hosted with the cluster broker) to install the appropriate Dom-M on the host (Dom-M is enabled as an appliance). Two brokers are currently used on the compute nodes – one for power which uses IPMI sensor data and another for reliability which uses IPMI and Machine Check Events (MCEs) for error statistics to determine current reliability of the host. A separate storage host runs a broker that monitors the reliability of disks using S.M.A.R.T. data. The various brokers implement the policies described in Section 4.3.

4.5 Evaluation

This section evaluates the vManage architecture and its application for the case study described in Section 4.3. Specifically, we first present an architecture evaluation of vManage and subsequently, using the vManage architecture, we evaluate SLA and utilization based power regulation, SLA-aware power budgeting, and reliability management. We also demonstrate the stability of our VM migration decisions. Results demonstrate that (1)

vManage is sufficiently rich to support the solution of complex management problems like the coordinated management of VMs and hosts; (2) policies implemented with vManage are effective, such as the novel cross-layer SLA-based power regulation policy which is more power-efficient than traditional utilization-based power management; (3) SLA- and migration-aware power budgeting along with power regulation can reduce power consumption and at the same time, improve system reliability by reacting quickly to power violations; (4) there are opportunities for reliability management, to significantly improve VM availability by predicting host failures and migrating VMs to other reliable hosts in advance; and (5) dynamic action assessment is an effective method, in this case concerning the effectiveness of our stability algorithm in reducing the number of VM migration by choosing target hosts more intelligently.

4.5.1 Experimental Setup and Methodology

Our experimental testbed consists of 4 Dell PowerEdge 1950 hosts connected through a gigabit network. One host works as the storage node providing virtual disks to all the VMs. The storage node also runs the cluster broker. The other three nodes are compute hosts running VMs. The hosts are dual-core dual-socket (total four cores) machines containing Intel Xeon 5150 processors with three different frequencies (2.66 GHz, 2.33 GHz, and 2.0 GHz) and 4 GB of memory. Experimentation with a larger testbed is planned when additional resources are acquired in late summer 2008.

Multiple workloads are used to emulate a typical data center environment. They include the RUBiS online auctioning application, the Nutch search engine, WebServer serving static files, and batch mode applications which run some standalone workload without any timing requirements. RUBiS is a three tier application with an Apache webserver, Tomcat application server, and a MySQL database server. Two application servers are used between webserver and db server for load balancing. All servers are run inside VMs. Hence, one RUBiS instance consists of 4 VMs. We have used two instances of Nutch, RUBiS, and static WebServers each in this "mini" data center, resulting in a total of 12 VMs (8 RUBiS VMs, 2 Nutch VMs, and 2 WebServer VMs). All VMs are uniprocessor VMs and have different

requirements for CPU cycles and memory.

We used two actual workload traces to generate workload for the VMs: (i) EPA-HTTP web traffic trace from the LBL Repository [22] to generate workload for the RUBiS applications and (ii) request traces from the Travelport [97] website to generate workload for Nutch and static WebServers. The two web traces contain traffic for more than an entire day. However, for purpose of experimentation, we scaled down the trace to run in 24 minutes (instead of 24 hours) while preserving the shape of the workload. The Travelport trace is discussed in Section 4.3.3 and is shown in Figure 27. All the workloads are primarily CPU bound.

Since these hosts do not have real-time power metering capabilities, we use CPU utilization as a proxy for power consumption. Specifically, we assume power consumption varies according to P = K*U + I, where P is consumed power, K is a constant, U is the CPU utilization and I is the idle power. K also depends on the current operating frequency (i.e., p-state) of the CPU. We determine the values of K and I offline, by calibrating the hosts using a power meter.

4.5.2 Architecture Evaluation

This section contains both a quantitative and a qualitative evaluation of the vManage architecture, with micro-benchmarks presented first. Table 5 summarizes the evaluation and is explained below.

Table 5: vManage Architecture Evaluation

Physical Channel Latency	Xen SHM	7 us
	Network Socket	14 us
Unified Monitoring	Max. CPU Usage	2%
	VM/VMM Monitoring	1.5 ms
	IPMI Monitoring	5 sec
Mediation Broker	Stress expt. Overhead (CPU)	15%
	MVM-CB Latency	1 ms
	MVM LoC	2670
	Xen/Dom-0 LoC	1085

Unified Monitoring Evaluation. Monitoring overheads via coordination channels depend on multiple factors: number of VMs on the host, monitoring data collection frequency,

and the resources allocated to the MVM. Table 5 shows latencies for runtime unified monitoring overhead (after initial setup) as the number of VMs is varied from 1 to 32. Initial channel setup takes place through a socket, and the monitoring data transfer is via shared memory. We see that MVM's max CPU usage and VM/Xen monitoring latency are very modest. The IPMI sensor data records take about 5 second to read. This indicates that the vManage monitoring infrastructure is very lightweight. Results also show that we are not likely to encounter scalability problems for MVM for a typical frequency of monitoring data (often 1 sample every few seconds) and for up to a few 10s of VMs per host.

Broker Evaluation. The costs of running an MVM directly depend on the complexities and frequencies of the analysis used to make coordination decisions. For intuition, we report the overhead of the MVM when the broker executes heavy analysis (stress experiment). Table 5 also shows that the max latency overhead of indirection between MVM broker and cluster broker (CB) is 1 ms. Finally, in our prototype, Xen/Dom-0 runs 1085 new/modified lines of code (LoC), while the MVM runs 2670 new/modified lines of code (LoC). Hence, in this implementation, MVM is able to isolate 71% of the new/modified code of the vManage prototype from Xen, clearly showing that having a separate MVM has significantly less impact on TCB of Xen and consequently has very little adverse impact on its reliability/security.

Scalability of Cluster Broker. To measure the scalability of the CB, we simulate and vary a cluster of hosts from 100 to 2500 hosts and measure the total CPU consumption and the time taken in making a VM placement/migration decision. Table 6 shows the scalability of cluster broker for the host update frequency of 1 per second. We see that as the number of hosts increase, the CPU usage, the search time for a host for VM migration, and the total response time for migration request from the host increase linearly. This shows that the cluster broker scales well for sizes typical for small data centers. For larger systems, a data center may be divided into multiple smaller size, separately managed clusters, with all clusters coordinating in a tree-like structure.

Other qualitative benefits. There are several other qualitative benefits for vManage: (i) *Automation:* we take a structured approach to designing cross-layer coordination,

Table 6: Cluster Broker Scalability

No. of hosts	CPU usage	Search time	Migration resp. time
100	0.5%	1 ms	1 ms
500	1.0%	2 ms	3 ms
1000	2.0%	3 ms	4 ms
1500	4.0%	4 ms	5 ms
2000	6.0%	5 ms	6 ms
2500	8.0%	6 ms	7 ms

thereby improving on manual or ad-hoc approaches. The general adoption of vManage, however, will require some support from community, but with our current effort to use standards-based protocols and leverage existing CIMOM deployments, we believe this to be plausible. (ii) *Framework extensibility and flexibility:* while the coordination channel infrastructure provides us with flexibility and easy portability across multiple platforms and implementations, the broker framework provides for ease of programmability, extensibility, and the potential for reusability.

4.5.3 Case Study Evaluation

We next evaluate the scenario represented in the case study (see Section 4.3) and show the benefit of the vManage-based architecture.

4.5.3.1 Overall Benefits

We first demonstrate the overall benefits obtained with respect to data center efficiency (power savings) and VM guarantees (SLA violations) with various coordination policies. This is summarized in Table 7.

Table 7: Summary of Benefits

Coordination Algorithms	Avg. Power	SLA Violations	VM Migrations
No Coord.	249.5W	20	-
Util. based	247.0W	25	-
SLA + Basic VM Placement	232.2W	26	89
SLA+Intelligent VM Placement	231.2W	20	62

With no coordination, there is no power management, and the CPUs always run at the highest frequency. The utilization-based algorithm uses current CPU utilization for power regulation, while the SLA-based algorithm uses SLA violation notifications for this purpose. The basic VM placement algorithm considers current CPU utilization to choose a target host for VM placement/migration, while the intelligent VM placement algorithm uses dynamic assessment and the workload characteristics of the VMs to make more stable VM placement/migration decisions and consequently reduce the total number of VM migrations. In summary, SLA-driven policies can provide additional benefits. Further, dynamic assessment helps reduce the number of migrations (i.e., intelligent VM placement), and these improvements will increase with larger equipment/testbed configurations (additional experimentation is in progress). We next explore experimental results in more detail.

4.5.3.2 SLA vs. Utilization Based Power Management

There are many possible cross-layer coordination alternatives. The specific ones compared in this case study are SLA-based and utilization-based power regulation policies, implemented with vManage. The utilization-based power regulation broker in the MVM attempts to save power by performing DVFS of the CPUs according to the current CPU utilization. If the utilization exceeds a certain threshold (80%), it increases the CPU frequency by one level and when the CPU utilization drops below a certain threshold (20%), it decreases the CPU frequency by one level. The SLA-based broker attempts to save power through DVFS of the CPUs according to the SLA violations experienced by the VM applications. The SLA of the application servers is defined in terms of the execution response time of the requests. SLA monitoring tools inside VMs capture the average execution time periodically (every 1 second) and a notification is to the MVM broker whenever the response time exceeds the SLA threshold or when the slack between response time and threshold is more than 50% of the threshold. When the broker receives a SLA violation notification from a VM, it increases the frequency of the CPU running the VM and when it receives a notification that the response time slack is more than 50%, it decreases the frequency of the CPU running the VM. For this experiment, we only used one instance of RUBiS and one instance of the

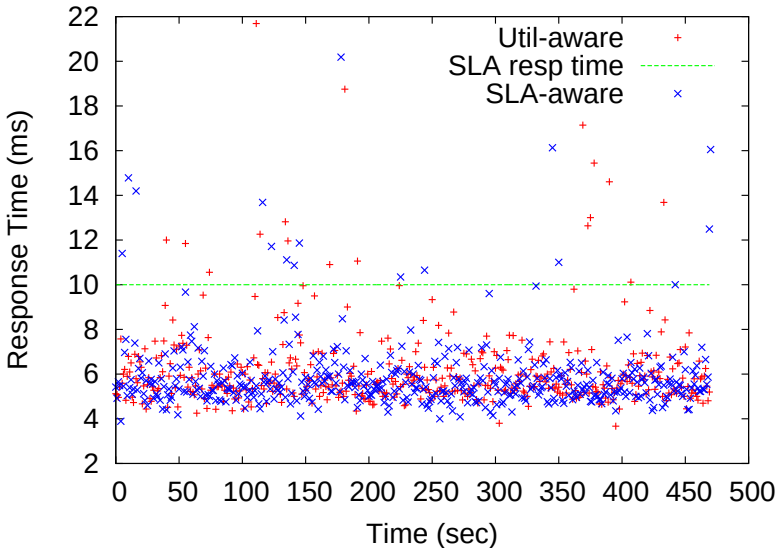

Figure 28: Comparison of execution times of Nutch search engine for Utilization based and SLA based power regulation policies

Nutch server on a single host.

Figures 28 and 29 shows the response time of Nutch application, for both power regulation policies. The SLA threshold is set at 10 ms, so any request execution time above 10 ms is considered an SLA violation. We see that the request execution time and SLA violation characteristic are similar for both policies. RUBiS' behavior is similar. Concerning power consumption, Figure 30 depicting the consumption trace over time for both policies, demonstrates that host power consumption under the SLA-based policy is less than under the utilization-based policy. The SLA-based policy reduces power consumption by 8%, although average CPU utilization is similar under both policies. The simple reason for this fact is that high utilization is not an appropriate metric for SLA violations. As a result, power regulation directly done based on SLA violations can often run the CPU at lower frequencies even under high CPU utilization. This constitutes a straightforward example of the benefits attained from vManage's ability to extend the cross-layer stack beyond VMs to the application level. The next two subsections use the SLA-based approach to demonstrate other useful elements of the vManage approach.

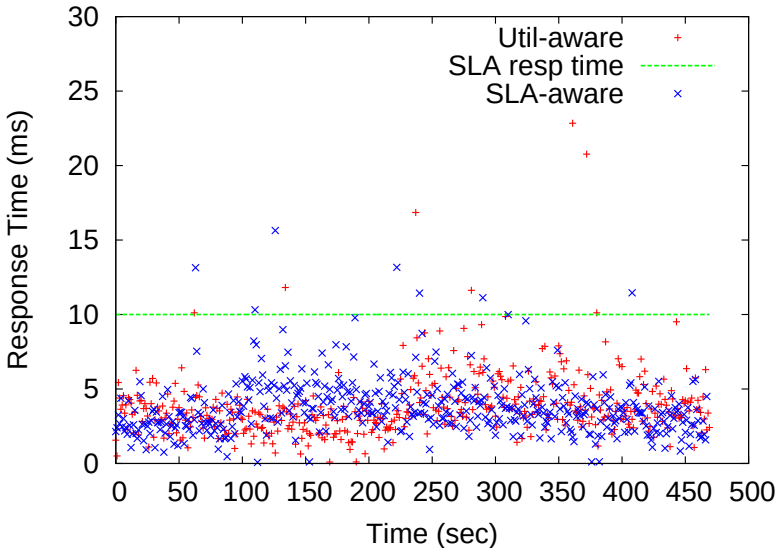

Figure 29: Comparison of execution times of RUBiS requests for Utilization based and SLA based power regulation policies

4.5.3.3 SLA-Based Coordinated Power Regulation And Power Capping

The lack of coordination between performance management and power capping can lead to conflicts between the two. To demonstrate how vManage can be used to solve this problem, we conduct an experiment similar to the one in Section 4.5.3.2. In addition, we run two batch mode applications in two other VMs on the host to increase the power consumption so that it violates the power budget (360 watts) set for the hosts. Under certain high load conditions, we may run into a situation where performance management conflicts with power capping. For example, when the SLA of an application is violated, the power regulator increases the CPU frequency to handle the load but this also increases the power consumption which may in turn violate the power budget of the host. To handle power violation, the power capper reduces the CPU frequency, which again causes additional SLA violations, thereby resulting in a lack of stability.

Figure 31 shows the result of unstable operation. The left side y axis show the response time of Nutch server, and right side y axis shows the power consumption as the experiment progresses over time. We see that often, SLA violations are caused right after power violations (because of power capper reducing CPU frequency), and power violations are caused

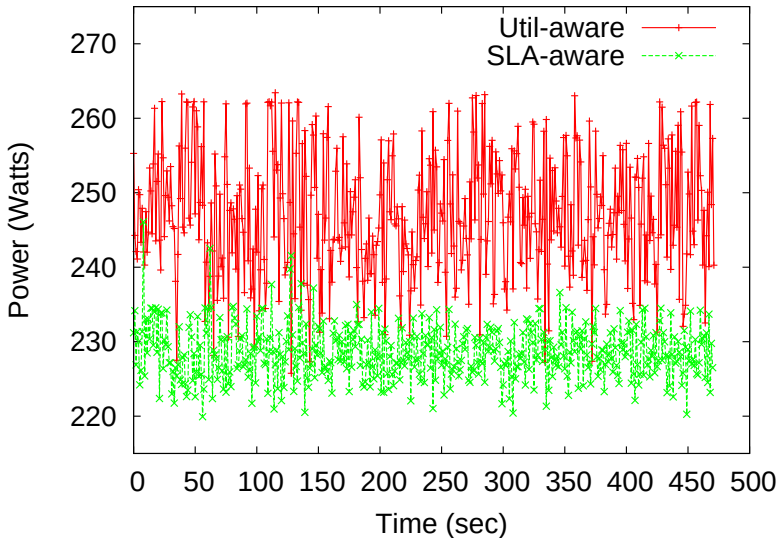

Figure 30: Comparison of power consumption of the host for Utilization based (avg. 247 watts) and SLA based (avg. 228.6 watts) power regulation policies

right after SLA violations (because of power regulator increasing the CPU frequency). Current power management schemes do not have the visibility into such SLA violations and keep aggressively reducing power consumption, causing more SLA violations and oscillation. In vManage, we recognize that under such overload conditions, per host management techniques may be insufficient, and power regulator and power capper must coordinate to migrate one or more VMs to relatively less loaded host(s) to reduce load and increase stability. We have implemented such a broker which coordinates between SLA management and power capping. Indeed, vManage makes it easy to implement such diverse cross layer monitoring and actuation.

Re-running the above experiment with our coordinated broker in place, when the frequency of power budget violations or SLA violations crosses a certain threshold, the MVM broker notifies the CB of the event. The CB chooses a VM (in this case, an application server of the RUBiS application) from the host (named H1), chooses a relatively less loaded host (H2), and then triggers a migration of the VM to host H2, thereby resolving the oscillation between SLA and power violations.

Figure 32 shows the response time of Nutch and the power consumption of host H1 as the experiment progresses. At around 52nd second, H1 experiences multiple power violations

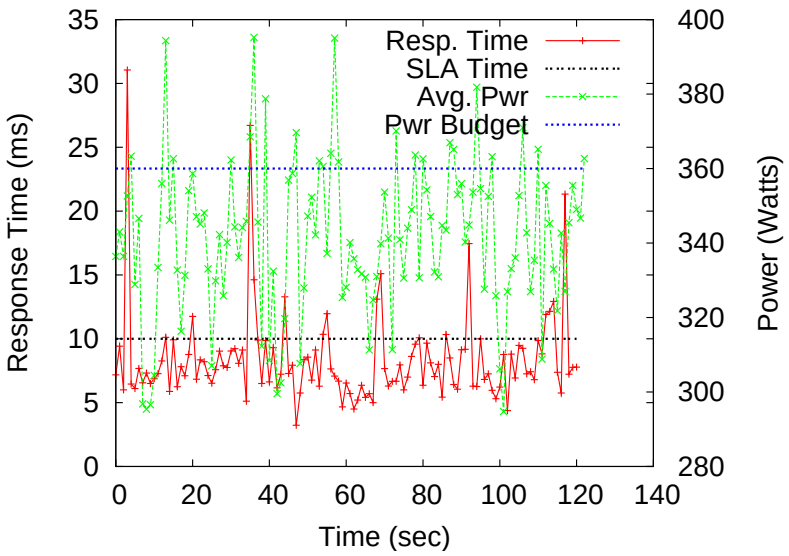

Figure 31: A demonstration of conflict between VM's SLA requirements and hosts power requirements. VMs and hosts both suffer because of lack of coordination between SLA management and power capping

which triggers a VM migration of RUBiS AppServer. This finally brings down the power consumption and reduces the SLA violation of the Nutch server significantly. Figure 33 shows the response time of the RUBiS application as its application server is migrated from H1 to H2. We see that the migration lasts for about 8 seconds and during this period, the RUBiS application experiences very high response time because: (1) the shadow mode is enabled during live VM migration [16], (2) the VM is suspended briefly during the last phase of VM migration, and (3) a significant amount of network bandwidth is used for migrating the VM itself. These facts again demonstrate the importance of dynamic assessment for vManage coordination actions. More generally, the experiment demonstrates that with the vManage architecture, we can develop automated solutions for coordinated performance management, power regulation, and power budgeting.

4.5.3.4 Intelligent VM Placement/Migration Algorithm (Dynamic Coordination Assessment)

To demonstrate the benefits of the VM placement algorithm described in Section 4.3.3, we only consider the CPU requirements of various VMs, and we assign various applications (Nutch, RUBiS, WebServer) CPU requirements that approximate the request rate of the

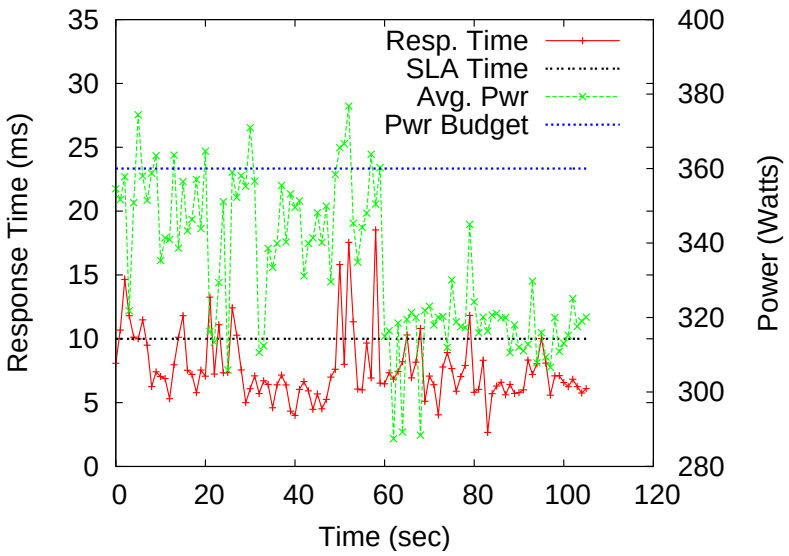

Figure 32: Coordination between SLA based power regulation and power budgeting. Excessive power violation causes RUBiS AppServer to be migrated which brings down the power consumption of the host and the response time of Nutch server

Travelport traces in Figure 27. Figure 34 shows the CPU requirements of all six workloads (RUBiS1, RUBiS2, Nutch1, Nutch2, WS1, and WS2). These workloads are shown over a 24 minute period and repeat after every 24 minutes. Due to discretization, the CPU requirement may change at the start of every 1 minute period, but it remains the same for the 1 minute period. The workloads are also time shifted so that they peak at different times, a typical behavior for servers operating in different time zones. This also allows us to over-provision the system. We have used *poisson* distribution to generate the workload requests for the VMs.

For simplicity, the RUBiS CPU requirements are equally divided among the 4 VMs. Figure 35 shows the three hosts present in the testbed and the cumulative CPU requirement of the VMs running on them. Initially, RUBiS1, Nutch1, and WS1 are running on Host1, RUBiS2 and Nutch2 are running on Host2 and WS2 is running on Host3. The experiment shows intelligent VM migration decision based on excessive SLA violations for this initial VM placement configuration.

Whenever the SLA violation frequency increases beyond a certain threshold (because of increased load), the MVM broker requests CB to migrate the VM to a relatively less

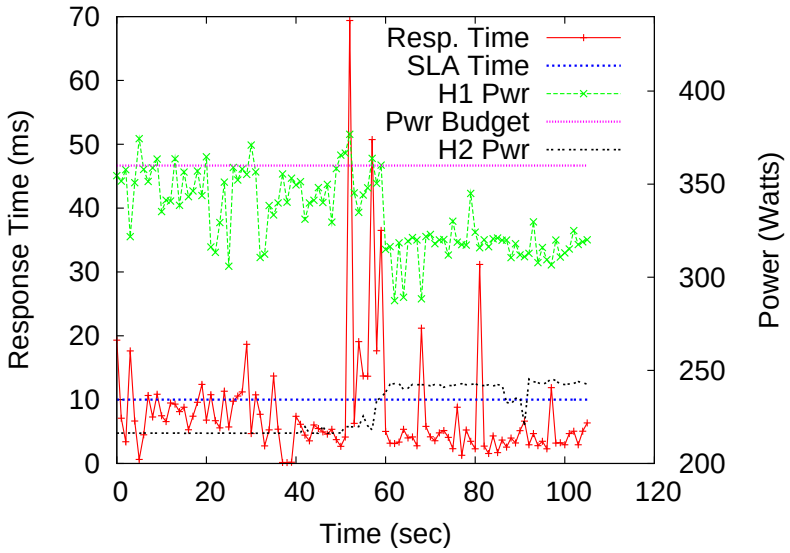

Figure 33: Execution response time of RUBiS before, during and after VM migration. Shows the power consumption traces on source and destination machines.

loaded host. CB searches its list of hosts and finds a suitable destination for the migration. The basic algorithm attempts to find a host by looking at the current CPU utilization and chooses the host which has the least CPU utilization. However, taking a decision based on current utilization may be short sighted because the CPU utilization may go up after VM migration, which may cause further VM migrations to happen.

Figures 36, 38, and 37 show the CPU utilization traces and VM migration instances by using the basic matching algorithm. As the experiment starts, Host2 quickly becomes overloaded and consequently, RUBiS2 and Nutch2 servers experience excessive SLA violations, triggering a VM migration (at about 290 seconds). CB chooses Host1 as the target for VM migration because its CPU utilization is less than the CPU utilization of Host3. The AppServer of RUBiS2 is migrated to Host1. However, Host1's CPU utilization quickly goes up because of the increase in its workload causing excessive SLA violations to occur. At this time (627 seconds), the AppServer of RUBiS2 is migrated to Host3 from Host1 because its CPU utilization is the lowest. Hence, the previous VM migration decision was a short-sighted one, because CB had to migrate the AppServer twice in quick succession. The two migrations (at 878 and 1075 seconds) migrate the AppServer of RUBiS1 from Host1 to Host2 and Nutch1 from Host1 to Host3. The experiment causes a total of 4 migrations in

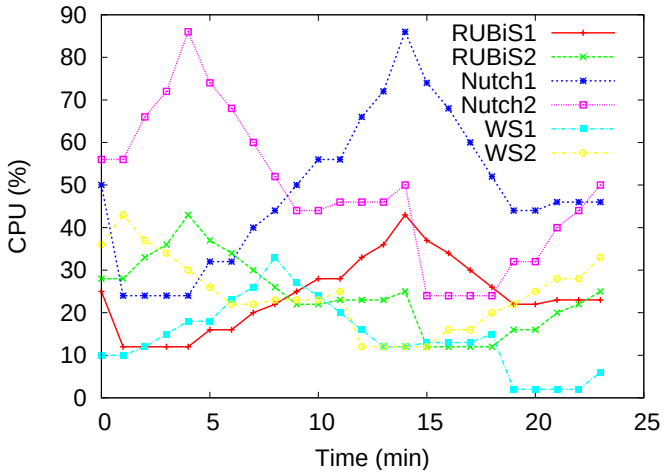

Figure 34: Mean CPU requirements for various applications over time

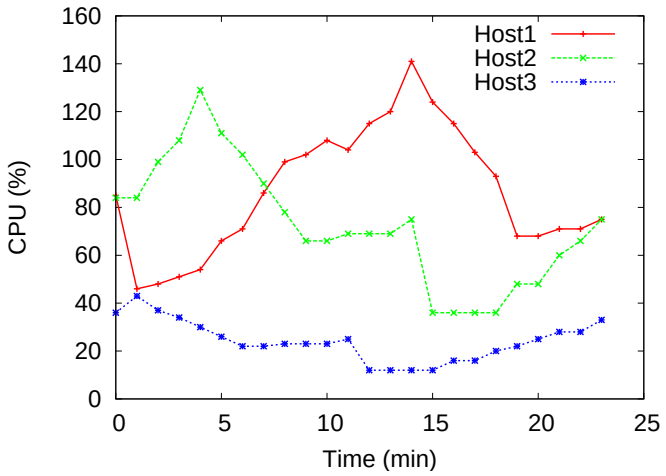

Figure 35: Mean CPU requirements for various hosts over time for a certain VM placement configuration

the 24 minute period.

With the intelligent algorithm developed in Section 4.3.3, CB keeps track of mean CPU utilization and variance of every VM over time. Hence, during the selection of a target host for VM migration, CB can evaluate the stability of its decision by calculating the probability of a Host satisfying the VM requirements over some future time period. Here, CB calculates the probability of a host satisfying CPU requirements for 15 minutes in the future when making migration decision.

Figures 39, 40, and 41 show the CPU utilization traces of Host1, 2, and 3 and VM

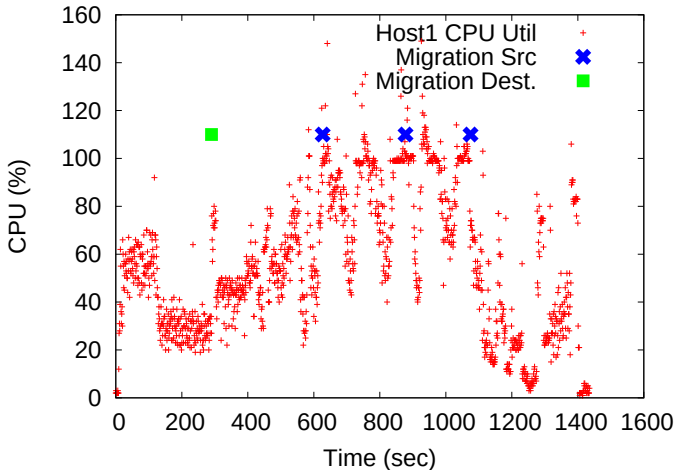

Figure 36: CPU utilization traces for Host H1 over time without intelligent placement

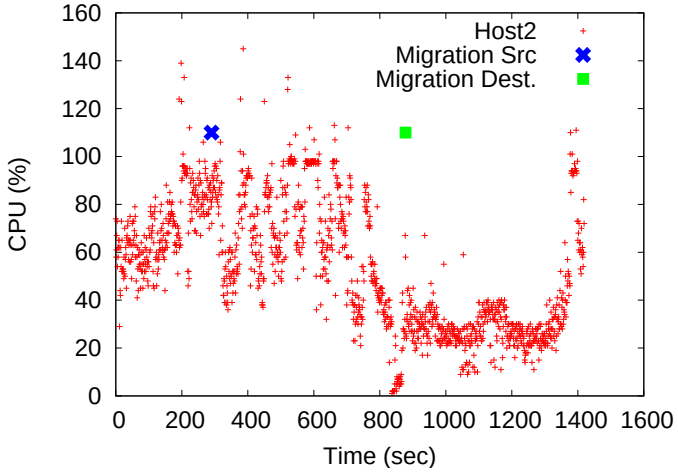

Figure 37: CPU utilization traces for Host H2 over time without intelligent placement

migration instances while using the intelligent matching algorithm. During this experiment, the first VM migration is triggered at 275 second on Host2. The intelligent algorithm, however, chooses Host3 instead of Host1 as the destination because Host3 has a higher probability of satisfying the AppServer's requirements over the next 15 minutes. The next two migrations (at 810 seconds and 1000 seconds) are the same as the last two migrations of the basic algorithm.

This experiment demonstrates that using the intelligent algorithm developed in Section 4.3.3, CB can make more stable VM migration decisions, in this case reducing the total number of VM migrations from 4 to 3. If we consider a larger data center environment

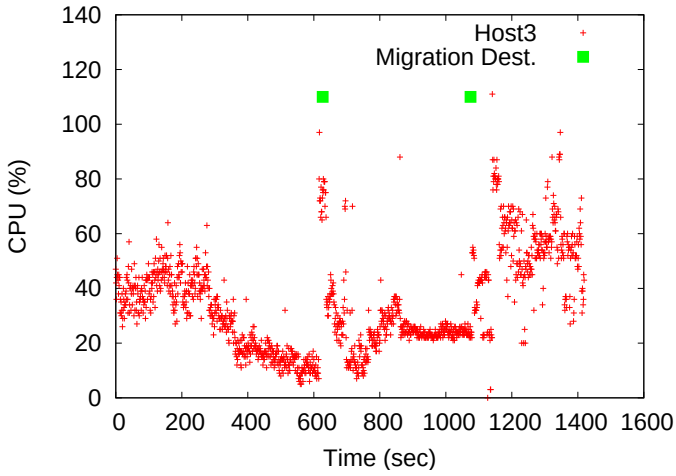

Figure 38: CPU utilization traces for Host H3 over time without intelligent placement

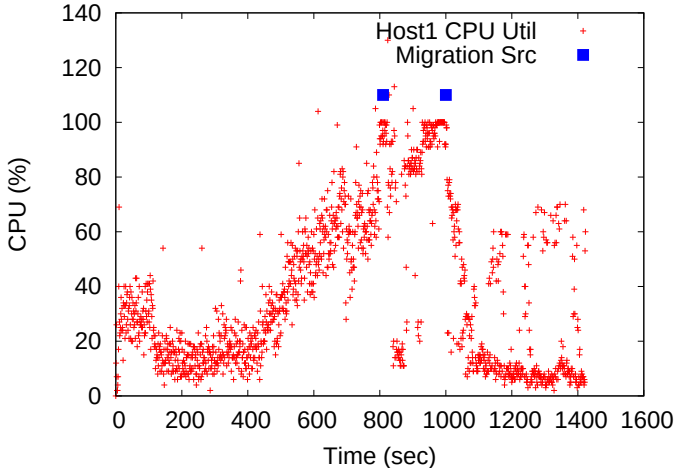

Figure 39: CPU utilization traces for Host H1 over time with intelligent placement

with thousands of machines and VMs, this modest reduction can expand into significant reductions in the total number of VM migrations over a large period of time, preserving significant network bandwidth, CPU cycles, and improves VMs' levels of availability. To validate our claims, we repeated the above experiment on a bigger testbed and for a longer period. We used 28 VMs (3 RUBiS, 10 Nutch, 6 Web Servers) running on 7 hosts. We ran the experiment for 20 hours each with and without intelligent VM placement algorithm. We observed the total number of VM migrations without the intelligent VM placement algorithm was 89 while with the intelligent VM placement algorithm, the number of VM migrations was significantly reduced to 62.

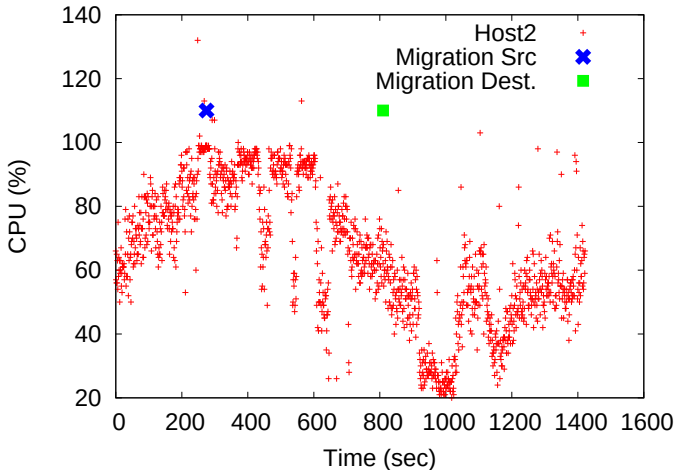

Figure 40: CPU utilization traces for Host H2 over time with intelligent placement

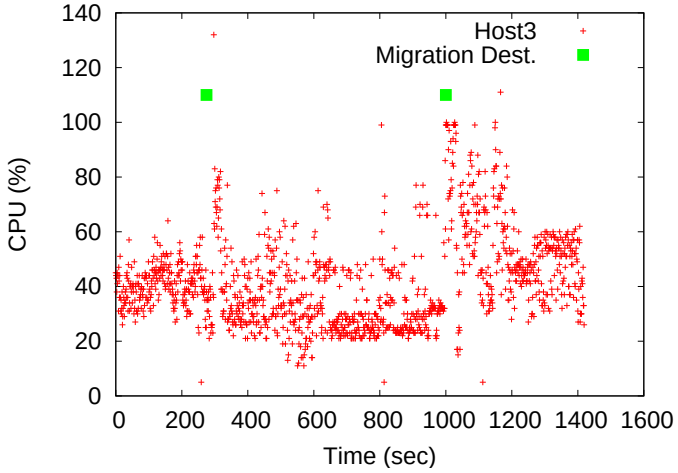

Figure 41: CPU utilization traces for Host H3 over time with intelligent placement

4.5.3.5 SLA-Reliability Coordinated Management

A brief foray into reliability management serves the purpose of further demonstrating the generality of the vManage approach. Specifically, we demonstrate the improvement in availability of VMs using reliability management through VM migration. In the experiment, a synthetic reliability model is used to determine the current reliability state of a host by monitoring MCE events. The MCE errors could be either correctable or uncorrectable. The correctable errors are automatically fixed by hardware, and the host can survive such errors. The uncorrectable errors, however, cannot be corrected, and in this experiment, we mark the host as unavailable after an uncorrectable MCE error. Since MCE errors happen rarely,

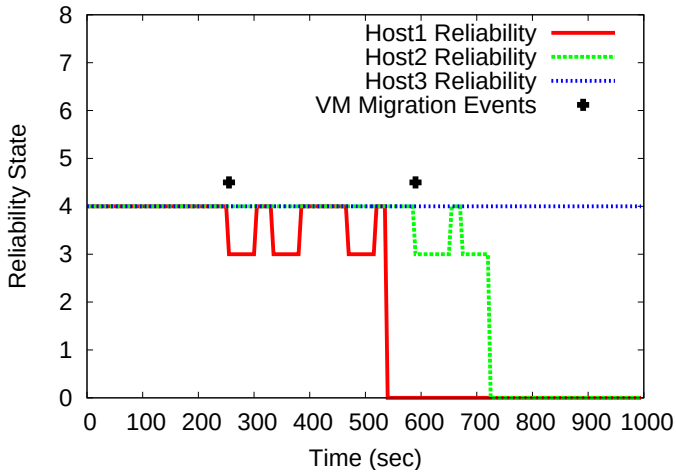

Figure 42: Reliability management of VMs using vManage

we injected artificial MCE errors to demonstrate this functionality. A uniform probability distribution function is used, with the probability of correctable errors higher than that of uncorrectable errors.

Figure 42 shows the result of the experiment. The three hosts have different reliability, i.e., the probability of MCE errors decreases from Host1 to Host3. Reliability is defined as states from 0 to 4, with 0 being unavailable (crashed) and 4 being most reliable. The Nutch VM's reliability requirement specifies that it can be run only on the most reliable machine (state 4). The experiment starts with 3 hosts in state 4, and the VM is running on Host1. At 325 seconds, a correctable MCE on Host1 reduces its reliability state to 3. This triggers a VM migration, because Host1 is no longer able to satisfy the VM's reliability requirement. Host2 is chosen as the destination by CB and the VM is migrated. When no more errors happen for the next 1 minute, the host's reliability is restored to 4. However, Host1 experiences further errors and finally it encounters an uncorrectable error and becomes unavailable (reliability 0). At 650 secs., Host2 encounters a correctable MCE error which causes the VM to migrate to Host3. Host2 also eventually encounters an uncorrectable error and becomes unavailable. The experiment shows that vManage, through its monitoring of reliability events along with other resource monitoring, can be used to greatly improve the availability of VMs in a virtualized data center. The results, of course, depend on our ability to predict impending failures and migrate VMs before the

failure occurs. Here, we use the occurrence of correctable MCE errors as an indication of an impending uncorrectable error in the future. We also note that, similar to compute node reliability, we can monitor storage disk reliability (e.g., monitoring S.M.A.R.T. data) and predict impending disk failures. vManage, then, as described in Section 4.3, can perform proactive backup or use techniques like VMWare's Storage VMotion [77] or Disk Hot-swapping [44] to significantly improve VM's and its data availability.

4.6 Related Work

Virtualization management solutions include industry solutions such as VMWare's Lifecycle Manage and Virtual Center [83]. These solutions, however, provide VM provisioning and dynamic management based on only traditional resources such as CPU, memory, priority etc. vManage, on the other hand, through its unified monitoring infrastructure, can provide VM provisioning and management based on novel metrics such as reliability, trust, and others. For example, Virtual Center and Sandpiper [96] provide dynamic VM migration for performance management, while vManage can additionally improve VM's availability by managing its reliability. Similarly, platform management solutions such as HP SIM [30] and application management solutions [31], respectively, focus on host and application metrics only.

Within coordination and unification, related work includes agent systems [91, 95], cross layer adaptation [2], and autonomic computing [39, 40]. While our approach borrows from prior work in these domains, we differ in (i) our light weight implementation approach suited to the VMM and system levels we target; and (ii) the instantiation of the coordination architecture across the "hardware-software-virtualization boundary". Past work on coordination in data centers has focused on specific domain policies and ad-hoc individual interfaces generating point solutions [65, 53]. Even the solutions focussing on coordination, provide solutions which are specific to some management actions, e.g., power and performance management [48]. vManage, on the other hand, provides a generic coordination architecture using which any number of cross-layer management solutions can coordinate with each other in an automated manner for improved data-center management.

Multiple management standards have been defined to manage resources in distributed environments, such as SNMP, WBEM, CIM, SMASH, DASH, PMCI, etc. We leverage these wherever appropriate and adapt them to meet the efficiency and dynamism support we require. An important aspect of vManage is to model system behavior based on monitoring data to take better management decisions. A significant amount of work has been done where statistical and machine learning techniques are used to build system behavior models at runtime [18, 46]. Other techniques use offline analysis of historical traces to determine system behavior, e.g., using Disk failure traces to determine disk failure models [76, 60]. vManage can use these techniques to understand and predict system behavior models, which would further improve the efficiency of its coordination actions.

4.7 Summary

Management and automation are important issues in future virtualized data centers. A critical and relevant issue today is the isolation of management solutions in different silos across the platforms, virtualization, and application layers, leading to inefficient management in data centers. In this chapter, we present the vManage architecture for achieving coordinated cross-layer management and show its mapping onto a virtualized system. vManage achieves loose coupling of management solutions using coordination channels and mediation brokers. Further, through its dynamic coordination assessment, vManage enables probabilistic analysis of management actions to improve stability and efficiency in virtualized enterprises. We have presented the basic design building blocks, the specific realization in virtualized environments, and have shown its usage through case study example.

vManage is used to implement dynamic VM and host provisioning so that both VM's and host's requirements are satisfied. It does per-node provisioning as well as extends the solution to cluster and data center level, including by using VM migration. It also uses the dynamic coordination assessment to evaluate the efficiency of VM migration actions to reduce total number of VM migrations. The architecture has been implemented in Xen, and evaluation results show that a vManage-based system provides improved overall management at acceptable overhead.

In the broader context, vManage improves virtualization of enterprise systems by efficiently managing them. It uses the lower-level mechanisms developed in previous chapters, e.g., Netchannel for efficient live migration and can also use Sidecore enabled VMMs for better core utilization and provisioning in a many-core system.

CHAPTER V

RELATED WORK

There have been many efforts, both in the past and ongoing, that attempt to improve the performance and manageability of virtualized environments. For example, attempts are ongoing, both in software and hardware, to improve the I/O performance for multicore systems in virtualized environments. Hardware attempts include developing high bandwidth, low latency interconnects, e.g., AMD's HyperTransport [33], Intel's CSI, and PCI Express, developing high bandwidth device controllers, virtualization extensions to processor architectures [87, 4], the introduction of IOMMUs to chipsets, etc. Software attempts include paravirtualization [21], self-virtualized devices [66], soft devices [89], VMM bypass [50], improving network throughput [41], etc. Similarly, multiple attempts like McRT [71], Accelerator [80], etc., propose novel ways to use cores in a many-core system. On the management front, there are rich software solutions like HP's SIM [30], IBM's Tivoli [34], VMWare's VirtualCenter [83], etc., for managing large data-center environments.

The idea of specialized/dedicated cores has been previously used in traditional operating systems both for homogeneous cores [69, 14, 101] as well as heterogeneous cores [28, 19, 9]. These solutions, however, become ineffective in virtualized environments because of the VMM layer between the OS and the cores. The Sidecore mechanism, on the other hand, operates at the VMM layer enabling VMMs to efficiently utilize and manage the cores as well as share them with VMs. It significantly improves the dynamic provisioning of various computing resources in the virtualized systems.

I/O virtualization is another important component of virtualized enterprises because (i) I/O virtualization incurs highest performance overhead significantly affecting VM performance, and (ii) lack of location transparency between virtual and physical I/O devices leads to inflexibility and inefficiency in VM migration which reduces the manageability of VMs. Previous attempts at high I/O throughput via pass-through access, e.g., Nomad [32] or live

VM migration solutions [84, 16] heavily depend on specialized and costly hardware support for I/O devices (e.g, intelligent Infiniband controller, Fiber channel SAN, etc.). Netchannel mechanism proposed by this dissertation, on the other hand, supports pass-through access and live migration of I/O devices (including device hot-swapping) in the VMM-layer making it independent of the specialized hardware support and thus reducing the cost wherever it is not warranted.

Further, the additional flexibility and functionality provided by above solutions increases the challenges of managing large virtualized enterprises. To address these challenges, there has been significant amount of past work on managing both non-virtualized as well as virtualized enterprises. Examples include power management solutions [65, 53], VM management solutions [83], resource provisioning solutions [46, 98, 88, 47], etc.

While these independent attempts have significantly improved the performance and manageability of computing systems (both virtualized and non-virtualized), the lack of coordination across multiple such solutions can lead to reduced performance and inefficient management. For example, lack of coordination between VM migration and reliability management solutions reduces a VM's availability in virtualized environments. This dissertation attempts to bridge the gap between existing solutions by first developing some of the required low-level mechanisms (Sidecore and Netchannel) and then providing a coordination architecture (vManage), so that different solutions can coordinate with each other, thereby improving the overall performance and utility of virtualized many-core systems.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This dissertation addresses the problem of efficiently using and managing virtualized environments, ranging from single many-core platforms to entire data-centers. Towards this end, we have developed a coordination architecture, termed *vManage*, which enables coordination between multiple management applications resulting in better management actions. However, to make *vManage* more effective, we first develop VMM level mechanisms to enhance the virtualization of processing cores and I/O devices in future many-core systems. For cores, we have developed the *Sidecore* approach, where one or more of the cores (both homogeneous and heterogeneous) can be specialized and/or dedicated to perform certain VMM or VM processing. Sidecore improves VMM and VM performance and provides improved flexibility in using and managing the diverse cores present on large many-core platforms. For I/O devices, we have developed the *Netchannel* mechanism, which provides *location transparency* of I/O devices for I/O virtualization. Netchannel enables *virtual device migration*, *device hot-swapping*, and *transparent device remoting* for virtualized as well as pass-through devices. Netchannel helps vManage by providing VMM level mechanism for seamless live VM migration and disk hot-swapping. Finally, *vManage*'s dynamic coordination assessment analyzes the stability of VM migration actions using probabilistic methods to reduce the total number of VM migrations in a data-center.

In summary, this dissertation provides hard evidence for the thesis statement, specifically, *novel VMM level techniques (at low levels of virtualization support as well as for higher-level management) should be developed to efficiently use large many-core systems and data-center environments.*

As an ongoing effort, we are currently working on more realistic reliability experiments, specifically, gathering failure related data from real systems for both compute and storage nodes and providing better reliability management by analyzing this data. Further, to make

the *vManage* architecture more scalable, we are also working on developing a hierarchical architecture where a large data-center can be divided into smaller clusters with cluster brokers coordinating with each other in a tree-like structure. As part of future work, besides working with *a priori* known models of system behavior (e.g., workload requirements), we plan to use machine learning techniques like *Bayesian networks* and *continuous time Markov chains* for learning system behavior dynamically. Such models can then be used for better dynamic coordination assessment without relying on prior knowledge of system behavior. We also plan to generalize the dynamic coordination assessment technique to analyze other management decisions, (e.g., hardware provisioning for the data-center, admission control of VMs on a host, etc.). Finally, we would like to extend the *vManage* architecture to support multiple VMMs, which requires VMMs to support common and standardized interfaces.

REFERENCES

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

[2] Adve, S., Harris, A., Hughes, C., Jones, D., Kravets, R., Nahrstedt, K., Sachs, D., Sasanka, R., Srinivasan, J., and Yuan, W., "The illinois grace project: Global resource adaptation through cooperation," 2002.

[3] "Amazon Elastic Compute Cloud." `http://aws.amazon.com/ec2/`, as retrieved on 06/30/2008.

[4] "AMD Pacifica Virtualization Technology." `http://www.amd.com/virtualization/`, as retrieved on 06/30/2008.

[5] Ammons, G., Appavoo, J., Butrico, M., Silva, D. D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Hensbergen, E. V., and Wisniewski, R. W., "Libra: a library operating system for a jvm in a virtualized execution environment," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, (New York, NY, USA), pp. 44–54, ACM, 2007.

[6] Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Silva, D. M. D., Krieger, O., Auslander, M. A., Edelsohn, D. J., Gamsa, B., Ganger, G. R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J., "Enabling autonomic behavior in systems software with hot swapping," *IBM Systems Journal*, vol. 42, no. 1, pp. 60–76, 2003.

[7] Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S., "The influence of operating systems on the performance of collective operations at extreme scale," in *CLUSTER*, 2006.

[8] "Removing the Big Kernel Lock." `http://kerneltrap.org/Linux/Removing_the_Big_Kernel_Lock`, as retrieved on 06/30/2008.

[9] Braun, F., Lockwood, J., and Waldvogel, M., "Protocol wrappers for layered network packet processing in reconfigurable networks," *IEEE Micro*, vol. 22, no. 1, 2002.

[10] Brecht, T., Janakiraman, J., Lynn, B., Saletore, V., and Turner, Y., "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O," in *Proc. of EuroSys*, 2006.

[11] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S., "The state of the art in locally distributed Web-server systems," in *ACM Computing Survey, 34(2):263311*, June 2002.

[12] Cecchet, E., Marguerite, J., and Zwaenepoel, W., "Performance and Scalability of EJB Applications," in *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 246–261, November 2002.

[13] "The Cell Architecture." `http://www.research.ibm.com/cell/`, as retrieved on 06/30/2008.

[14] Chakraborty, K., Wells, P. M., and Sohi, G. S., "Computation spreading: employing hardware migration to specialize cmp cores on-the-fly," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 283–292, ACM Press, 2006.

[15] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A., "Sla decomposition: Translating service level objectives to system level thresholds," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, (Washington, DC, USA), p. 3, IEEE Computer Society, 2007.

[16] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A., "Live migration of virtual machines," in *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA.*, May 2005.

[17] Clark, N., Hormati, A., and Mahlke, S., "VEAL: Virtualized execution accelerator for loops," in *Proc. of the International Symposium on Computer Architecture*, (Beijing, China), June 2008.

[18] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S., "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[19] "NVIDIA CUDA (Computer Unified Device Architecture)." `http://www.nvidia.com/object/cuda_home.html`, as retrieved on 06/30/2008.

[20] Dahlin, M., Wang, R., Anderson, T. E., and Patterson, D. A., "Cooperative caching: Using remote client memory to improve file system performance," in *Operating Systems Design and Implementation*, pp. 267–280, 1994.

[21] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R., "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[22] "EPA-HTTP - a day of HTTP logs from EPA WWW server." `http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html`, as retrieved on 06/30/2008.

[23] et. al., J. H., *Feedback Control of Computing Systems.* John Wiley & Sons, 2004.

[24] Gavrilovska, A., *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors.* PhD thesis, Georgia Institute of Technology, 2004.

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

[26] Gupta, D., Cherkasova, L., and Vahdat, A., "Enforcing performance isolation across virtual machines in xen," in *Proceedings of the ACM/IFIP/USENIX Middleware Conference*, Nov. 2006.

[27] Gupta, V. and Xenidis, J., "Cellule: Virtualizing cell/b.e. for lightweight execution," in *Georgia Tech STI Cell/B.E. Workshop*, 2007.

[28] Hady, F. T., Bock, T., Cabot, M., Chu, J., Meinechke, J., Oliver, K., and Talarek, W., "Platform Level Support for High Throughput Edge Applications: The Twin Cities Prototype," *IEEE Network*, July/August 2003.

[29] Hirofuchi, T., Kawai, E., Fujikawa, K., and Sunahara, H., "Usb/ip - a peripheral bus extension for device sharing over ip network," in *USENIX 2005 Annual Technical Conference, FREENIX Track*, Apr. 2005.

[30] "HP Systems Insight Manager." `http://h18002.www1.hp.com/products/servers/management/hpsim/index.html`, as retrieved on 06/30/2008.

[31] "HP Management Software." `http://www.hp.com/go/software`, as retrieved on 06/30/2008.

[32] Huang, W., Liu, J., Koop, M., Abali, B., and Panda, D., "Nomad: migrating os-bypass networks in virtual machines," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, 2007.

[33] "HyperTransport Interconnect." `http://www.hypertransport.org/`, as retrieved on 06/30/2008.

[34] "IBM Tivoli Software." `http://www-306.ibm.com/software/tivoli/`, as retrieved on 06/30/2008.

[35] Intel Corporation, *Intel Network Processor Family.* developer.intel.com/design/-network/producs/npfamily/.

[36] "Intelligent Platform Management Interface." `http://www.intel.com/design/servers/ipmi/index.htm`, as retrieved on 06/30/2008.

[37] Jung, G., Joshi, K., Hiltunen, M., Schlichting, R., and Pu, C., "Generating adaptation policies for multi-tier applications in consolidated server environments," in *ICAC '08: Proceedings of the Fifth International Conference on Autonomic Computing*, (Chicago, USA), 2008.

[38] Keir Fraser and Steven Hand and Rolf Neugebauer and Ian Pratt and Andrew Warfield and Mark Williamson, "Reconstructing i/o," tech. rep., University of Cambridge, Computer Laboratory, August 2004.

[39] Kephart, J. O., "Research challenges of autonomic computing," in *ICSE '05*, 2005.

[40] Kephart, J. O., Chan, H., Chan, H., Das, R., Levine, D. W., Tesauro, G., Rawson, F., and Lefurgy, C., "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, (Washington, DC, USA), p. 24, IEEE Computer Society, 2007.

[41] Koh, Y., Pu, C., Bhatia, S., and Consel, C., "Efficient packet processing in user-level os: A study of uml," in *31st IEEE Conference on Local Computer Networks (LCN)*, 2006.

[42] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Silva, D. D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V., "K42: building a complete operating system," *SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 133–145, 2006.

[43] Kumar, S., Gavrilovska, A., Schwan, K., and Sundaragopalan, S., "C-core: Using communication cores for high performance network services," in *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, (Washington, DC, USA), pp. 171–178, IEEE Computer Society, 2005.

[44] Kumar, S. and Schwan, K., "Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (New York, NY, USA), pp. 31–40, ACM, 2008.

[45] Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K., "Resource-Aware Distributed Stream Management using Dynamic Overlays," *Proc. of ICDCS-25*, 2005.

[46] Kumar, V., Schwan, K., Iyer, S., Chen, Y., and Sahai, A., "A state space approach to sla based management," in *IEEE/IFIP Network Operations & Management Symposium (NOMS)*, 2008.

[47] Kusic, D. and Kandasamy, N., "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems," *Cluster Computing*, vol. 10, no. 4, pp. 395–408, 2007.

[48] Kusic, D., Kephart, J., Hanson, J., Kandasamy, N., and Jiang, G., "Power and performance management of virtualized computing environments via lookahead control," in *ICAC '08: Proceedings of the Fifth International Conference on Autonomic Computing*, (Chicago, USA), 2008.

[49] LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B., and Heiser, G., "Pre-virtualization: Slashing the cost of virtualization," Tech. Rep. Technical Report 2005-30, Fakultt fr Informatik, Universitt Karlsruhe (TH), 2005.

[50] Liu, J., Huang, W., Abali, B., and Panda, D. K., "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.

[51] "LMbench - Tools for Performance Analysis." `http://www.bitmover.com/lmbench`, as retrieved on 06/30/2008.

[52] "Gartner TCO Reports. 2005, 2006, 2007." `http://www.gartner.com/`, as retrieved on 06/30/2008.

[53] Nathuji, R. and Schwan, K., "Virtualpower: Coordinated power management in virtualized enterprise systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[54] "The Future of Ethernet I/O Virtualization is Here Today." `http://www.netxen.com/technology/pdfs/FutureofEthernet.pdf`, as retrieved on 06/30/2008.

[55] Oleson, V., Schwan, K., Eisenhauer, G., Plale, B., Pu, C., and Amin, D., "Operational Information Systems - An Example from the Airline Industry," in *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[56] "PCI Express IO Virtualization and IO Sharing Specification." `http://www.pcisig.com/members/downloads/specifications/pciexpress/specification/draft/IOV_spec_draft_0.3-051013.pdf`, as retrieved on 06/30/2008. Available to members only.

[57] Petrini, F., Fernandez, J., Moody, A., Frachtenberg, E., and Panda, D., "NIC-based Reduction Algorithms for Large-scale Clusters," *International Journal of High Performance Computing and Networking (IJHPCN)*, 2005.

[58] Pfaff, B., Garfinkel, T., and Rosenblum, M., "Virtualization aware file systems: Getting beyond the limitations of virtual disks," in *NSDI*, 2006.

[59] Pietzuch, P. and Bhola, S., "Congestion Control in a Reliable, Scalable Message-Oriented Middleware," in *Proc. of Middleware 2003*, (Rio de Janeiro, Brazil), 2003.

[60] Pinheiro, E., Weber, W.-D., and Barroso, L. A., "Failure trends in a large disk drive population," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2007.

[61] "Plan 9 Remote Resource Protocol." `http://v9fs.sourceforge.net/rfc/`, as retrieved on 06/30/2008.

[62] Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A., "Xen 3.0 and the Art of Virtualization," in *Proc. of the Ottawa Linux Symposium*, 2005.

[63] "IBM System p Servers." `www.ibm.com/systems/p`, as retrieved on 06/30/2008.

[64] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., "Optimistic incremental specialization: streamlining a commercial operating system," in *In proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP)*, (New York, NY, USA), 1995.

[65] Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X., "No "power" struggles: coordinated multi-level power management for the data center," in *ASPLOS-XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[66] Raj, H. and Schwan, K., "High performance and scalable i/o virtualization via self-virtualized devices," in *HPDC*, 2007.

[67] Raman, B. and Katz, R., "An Architecture for Highly Available Wide-Area Service Composition," *Computer Communications Journal*, May 2003.

[68] Rangarajan, M. and Bohra, A., "Tcp servers: Offloading tcp/ip processing in internet servers. design, implementation, and performance," Tech. Rep. DCS-TR-481, Rutgers University, Department of Computer Science, 2002.

[69] Regnier, G., Minturn, D., McAlpine, G., Saletore, V., and Foong, A., "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," *IEEE Micro*, vol. 24, no. 1, pp. 24–31, 2004.

[70] "SteelEye Technologies: Data replication." `http://www.steeleye.com/products/ext_mirroring.html`, as retrieved on 06/30/2008.

[71] Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J., "Enabling scalability and performance in a large scale cmp environment," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (New York, NY, USA), pp. 73–86, ACM, 2007.

[72] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., "Design and implementation of the Sun Network Filesystem," in *Proc. Summer 1985 USENIX Conf.*, (Portland OR (USA)), pp. 119–130, 1985.

[73] Sapuntzakis, C., Chandra, R., Pfaff, B., Chow, J., Lam, M. S., and Rosenblum, M., "Optimizing the migration of virtual computers," in *OSDI*, 2002.

[74] "Virtualization with the HP BladeSystem." `http://h71028.www7.hp.com/enterprise/downloads/virtualization%20on%20the%20HP%20BladeSystem.pdf`, as retrieved on 06/30/2008.

[75] Schantz, R. E., Loyall, J. P., Rodrigues, C., and Schmidt, D. C., "Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1189–1208, 2006.

[76] Schroeder, B. and Gibson, G. A., "Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, (Berkeley, CA, USA), p. 1, USENIX Association, 2007.

[77] "VMWare Storage VMotion." `http://www.vmware.com/products/vi/storage_vmotion.html`, as retrieved on 06/30/2008.

[78] Sugerman, J., Venkitachalam, G., and Lim, B.-H., "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2001.

[79] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M., "Recovering device drivers," in *OSDI*, 2004.

[80] Tarditi, D., Puri, S., and Oglesby, J., "Accelerator: using data parallelism to program gpus for general-purpose uses," *SIGPLAN Notices*, vol. 41, no. 11, pp. 325–335, 2006.

[81] "Terascale Computing: Intel Platform Research." `http://www.intel.com/research/platform/terascale/`, as retrieved on 06/30/2008.

[82] "Intel QuickAssist Technology." `http://www.intel.com/technology/magazine/45nm/quickassist-0507.htm`, as retrieved on 06/30/2008.

[83] "VMWare Virtual Center." `http://www.vmware.com/products/vi/vc/`, as retrieved on 06/30/2008.

[84] "VMWare's VMotion Technology." `http://www.vmware.com/products/vi/vc/vmotion.html`, as retrieved on 06/30/2008.

[85] "The VMWare ESX Server." `http://www.vmware.com/products/esx/`, as retrieved on 06/30/2008.

[86] "VMware Lifecycle Manager." `http://www.vmware.com/products/lcm`, as retrieved on 06/30/2008.

[87] "Intel Virtualization Technology Specification for the IA-32 Intel Architecture." `http://www.intel.com/cd/ids/developer/asmo-na/eng/197666.htm`, as retrieved on 06/30/2008.

[88] Wang, X., Lan, D., Wang, G., Fang, X., Ye, M., Chen, Y., and Wang, Q., "Appliance-based autonomic provisioning framework for virtualized outsourcing data center," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, (Washington, DC, USA), p. 29, IEEE Computer Society, 2007.

[89] Warfield, A., Hand, S., Fraser, K., and Deegan, T., "Facilitating the development of soft devices," in *USENIX 2005 Annual Technical Conference, General Track*, Apr. 2005.

[90] "Web-Based Enterprise Management." `http://www.dmtf.org/standards/wbem/`, as retrieved on 06/30/2008.

[91] Weiss, G., ed., *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, 1999.

[92] Weisstein, E., "Normal Sum Distribution." `http://mathworld.wolfram.com/NormalSumDistribution.html`, as retrieved on 06/30/2008.

[93] Witchel, E., Larson, S., Ananian, C. S., and Asanovic, K., "Direct addressed caches for reduced power consumption," in *34th International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[94] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "Smart Pointers: Personalized Scientific Data Portals in Your Hand," in *Proc. of Supercomputing 2002*, Nov. 2002.

[95] Wolf, T. D. and Holvoet, T., "Towards Autonomic Computing: agent-based modelling, dynamical systems analysis, and decentralised control," in *Proceedings of the First International Workshop on Autonomic Computing Principles and Architectures*, 2003.

[96] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M., "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[97] "Worldspan by Travelport." `http://www.worldspan.com`, as retrieved on 06/30/2008.

[98] Zhang, Q., Cherkasova, L., and Smirni, E., "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, (Washington, DC, USA), p. 27, IEEE Computer Society, 2007.

[99] Zhang, X., Bhuyan, L. N., and chun Feng, W., "Anatomy of UDP and M-VIA for Cluster Communications," *Journal on Parallel and Distributed Computing*, 2005.

[100] Zhao, Y. and Storm, R., "Exploiting Event Stream Interpretation in Publish-Subscribe Systems," in *Proc. of ACM Symposium on Principles of Distributed Computing*, Aug. 2001.

[101] Zilles, C. B., Emer, J. S., and Sohi, G. S., "The use of multithreading for exception handling.," in *MICRO*, pp. 219–229, 1999.

[102] "IBM zSeries Mainframe." `www.ibm.com/systems/z`, as retrieved on 06/30/2008.

VITA

Sanjay Kumar is a PhD candidate at the College of Computing, Georgia Institute of Technology. He is also a member of the NSF-sponsored Center for Experimental Research in Computer Systems (CERCS) which conducts research in the domains of Enterprise, High Performance, and Embedded Systems. His research interests include virtualization, many-core systems, large-scale enterprise systems and autonomic computing. He holds a Master of Science in Computer Science from Georgia Institute of Technology and a Bachelor of Technology in Computer Science from Indian Institute of Technology, Roorkee, India.