IMPROVING DATA CENTER RESOURCE MANAGEMENT, DEPLOYMENT, AND AVAILABILITY WITH VIRTUALIZATION

A Dissertation Presented

by

TIMOTHY WOOD

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2011

Computer Science

IMPROVING DATA CENTER RESOURCE MANAGEMENT, DEPLOYMENT, AND AVAILABILITY WITH VIRTUALIZATION

A Dissertation Presented

by

TIMOTHY WOOD

ACKNOWLEDGMENTS

This thesis would not have been completed without the guidance of my advisor and the support of my friends and family. Professor Prashant Shenoy taught me the key skills needed by any researcher: how to pick a problem, pursue it to completion, and then present my ideas to the world. When deadlines drew close, his ability to help me see the broader picture was invaluable for learning how to prioritize my efforts and produce the best work possible.

I have enjoyed working in such a collaborative department, and I have learned a great deal from my other committee members, Professors Arun Venkataramani, Mark Corner, Jim Kurose, and Mani Krishna. I particularly appreciate Arun Venkataramani's help when I first arrived as a graduate student and Prashant was on sabbatical. I was fortunate to be able to work with Arun on several other projects throughout my time in Amherst and greatly appreciate his guidance through the murky waters that surround Byzantine Fault Tolerance. I was also able to work with Professors Mark Corner, Emery Berger, and David Jensen, each of whom provided me with both new perspectives on my own work and a broader vision of what is possible in Systems research.

I have collaborated with researchers outside of UMass while on internships at HP and AT&T. I would like to thank Lucy Cherkasova for mentoring me during my time at HP Labs, and for helping me explore the modeling aspects of my work. I have had extensive collaborations with KK Ramakrishnan, Jacobus van der Merwe, and Andres Lagar-Cavilla from AT&T Research, and I am grateful for their support and for sharing their time and ideas with me.

My peers within the Computer Science department also played an important role during my time in Amherst. I am grateful to the other members of the LASS group, particularly

Emmanuel Cecchet, Rahul Singh, Upendra Sharma, Peter Desnoyers, Jeremy Gummeson, and Ming Li, who answered my questions and assisted with projects when deadlines were tight. They were both helpful coworkers and good friends. I am also indebted to Tyler Trafford for his assistance managing the server clusters used in my work and to Leeanne Leclerc for keeping my paperwork organized and on track; they were able to solve every problem I had.

My time in Amherst was made even more enjoyable by the incredible group of friends I developed over the years. Gal Niv, Yariv Levy, the Feilds, the Balasubramanians, Bobby Simidchieva, Marc Cartright, Stefan Christov, TJ Brunette, Filip Jagodzinski, Marek Petrik, and too many others to list helped me take a break with company, conversation, and, inevitably, good food and drink. The computer science community was a great place to spend my work and free time, and I will miss you all. I also apologize to all those on the broadcast@cs email list who had to suffer through my email announcements and poor puns.

I might not have gone down the graduate school path if it had not been for my family, who raised me to pursue knowledge and explore my creativity. I also thank Steve Stoft, a family friend who had me changing motherboards at an early age and taught me my first programming language. Finally, a very special thanks goes to my wife, Megan Olsen. Going through the Ph.D. process together with her was infinitely more enjoyable than doing so by myself, and I look forward to continuing our journey together.

ABSTRACT

IMPROVING DATA CENTER RESOURCE MANAGEMENT, DEPLOYMENT, AND AVAILABILITY WITH VIRTUALIZATION

SEPTEMBER 2011

TIMOTHY WOOD

B.S., RUTGERS UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

The increasing demand for storage and computation has driven the growth of large data centers–the massive server farms that run many of today's Internet and business applications. A data center can comprise many thousands of servers and can use as much energy as a small city. The massive amounts of computation power contained in these systems results in many interesting distributed systems and resource management problems. In this thesis we investigate challenges related to data centers, with a particular emphasis on how new virtualization technologies can be used to simplify deployment, improve resource efficiency, and reduce the cost of reliability, all in application agnostic ways.

We first study problems that relate to the initial capacity planning required when deploying applications into a virtualized data center. We demonstrate how models of virtualization overheads can be utilized to accurately predict the resource needs of virtualized applications, allowing them to be smoothly transitioned into a data center. We next study how memory similarity can be used to guide placement when adding virtual machines to a data center, and demonstrate how memory sharing can be exploited to reduce the memory footprints of virtual machines. This allows for better server consolidation, reducing hardware and energy costs within the data center.

We then discuss how virtualization can be used to improve the performance and efficiency of data centers through the use of “live” migration and dynamic resource allocation. We present automated, dynamic provisioning schemes that can effectively respond to the rapid fluctuations of Internet workloads without hurting application performance. We then extend these migration tools to support seamlessly moving applications across low bandwidth Internet links.

Finally, we discuss the reliability challenges faced by data centers and present a new replication technique that allows cloud computing platforms to offer high performance, no data loss disaster recovery services despite high network latencies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Modern data centers are comprised of tens of thousands of servers, and perform the processing for many Internet business applications. Data centers are increasingly using virtualization to simplify management and make better use of server resources. This thesis discusses the challenges faced by these massive data centers, and presents how virtualization can provide innovative solutions.

1.1 Background and Motivation

Internet and business applications are increasingly being moved to large data centers that hold massive server and storage clusters. Current data centers can contain tens or hundreds of thousands of servers, and plans are already being made for data centers holding over a million servers [65]. Some data centers are built to run applications for a single company, such as the search engine clusters run by Google. Other data centers are operated by service providers that are able to rent storage and computation resources to other customers at very low cost due to their large scale. *Cloud computing*, which refers to hosting platforms that rent data center resources to customers, is becoming increasingly popular for running Internet websites or business applications. In all of these data centers, the massive amounts of computation power required to drive these systems results in many challenging and interesting distributed systems and resource management problems.

Virtualization promises to dramatically change how data centers operate by breaking the bond between physical servers and the resource shares granted to applications. Virtualization can be used to “slice” a single physical host into one or more *virtual machines* (VMs) that share its resources. This can be useful in a hosting environment where customers or applications do not need the full power of a single server. In such a case, virtualization provides an easy way to isolate and partition server resources. The abstraction layer between the VM and its physical host also allows for greater control over resource management. The CPU and memory allocated to a virtual machine can be dynamically adjusted, and live migration techniques allow VMs to be transparently moved between physical hosts without impacting any running applications.

As data centers continue to deploy virtualized services, new problems have emerged such as determining optimal VM placements and dealing with virtualization overheads. At the same time, virtualization allows for new and better solutions to existing data center problems by allowing for rapid, flexible resource provisioning. The central theme of this thesis is to explore how virtualization can allow for application agnostic solutions when dealing with challenges related to application deployment, resource management, and reliability. Specifically, we try to answer the following questions:

- How can we transition applications running on native hardware to virtual machines while ensuring they receive sufficient resources despite virtualization overheads?
- On what servers should we deploy new VMs in order to obtain the greatest level of server consolidation?
- How can we efficiently manage server resources despite highly varying application workloads?
- How can we effectively connect and manage multiple data centers?
- How can we ensure application reliability despite unexpected disasters that can bring down entire data centers?

The data center environment makes these challenges particularly difficult since it requires solutions with high scalability and extreme speed to respond quickly to fluctuating

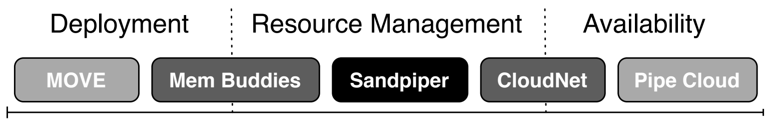


Figure 1.1. The systems described in this proposal explore the challenges and relationships between service deployment, resource management, and reliability.

Internet workloads. By tackling these problems, data centers can be made more efficient and reliable, significantly reducing their hardware costs and energy utilization.

1.2 Thesis Contributions

Many of the challenges described in this thesis are not new problems related solely to virtualized data centers. They are classical resource management problems that are often further compounded by the massive scale of modern data centers. However, in each case, we propose novel techniques that combine the flexibility and speed of virtualization with intelligent control algorithms and modeling techniques.

1.2.1 Contribution Summary

This thesis proposes virtualization based techniques to simplify deployment, automate resource management, and provide greater resilience in modern data centers. The fundamental thesis of this dissertation is that *virtualization can provide powerful, application agnostic techniques to improve data center agility, efficiency, and reliability*. To this end we have designed five key systems which provide new algorithms and mechanisms for advanced virtualized data centers:

- *MOVE*: An automated model generation technique that quantifies the cost of virtualization layer overheads to ease the transition to virtualized data centers [143].

- *Memory Buddies*: A VM placement scheme that maximizes memory sharing between VMs in order to provide greater server consolidation and improve memory management [146].

- *Sandpiper*: Automated hotspot detection and mitigation techniques that utilize dynamic resource allocation and live VM migration to prevent server overload [145, 141].
- *CloudNet*: A virtual network based infrastructure that seamlessly connects multiple data centers and enables optimized migration of virtual machines between geographically separated data centers [139, 144].
- *PipeCloud*: A new approach to data replication that enables cloud based disaster recovery to provide synchronous consistency guarantees while maintaining the same level of performance as asynchronous approaches, despite higher WAN latencies [142, 140].

These systems cover a spectrum of overlapping data center challenges as illustrated in Figure 1.1.

1.2.2 Planning and Placement

Virtualization provides many benefits, but it also incurs a cost in the form of overheads caused by the hypervisor. These costs come from various activities within the virtualization layer such as binary code rewriting, traps caused by OS memory operations, and, most commonly, I/O operation overhead. The actual overhead varies depending on the virtualization platform being used, and different applications can see different types of overhead depending on the nature of the application's activities.

In Chapter 3 we propose the use of virtualization overhead models to help predict how resource requirements of an application will change when it is transitioned to a virtualized environment. We present MOVE, an automated model generation technique that builds general purpose models that map the relationship between a native and virtual platform. This simplifies the deployment of new applications to virtualized data centers since their resource requirements can be easily predicted ahead of time. MOVE creates "generic" models which can be applied to traces of any application which must be transitioned to the virtual environment.

After the resource requirements of a new virtual machine are known, it must be placed on a host within the data center. Minimizing the number of servers required to host a given set of VMs can reduce the hardware and energy costs of a data center. Bin-packing algorithms have been used to determine VM placements, but existing techniques often require knowledge of the applications running within each virtual machine to determine how they should be placed.

Chapter 4 presents how exploiting memory sharing between virtual machines can provide significant server consolidation benefits. We describe the Memory Buddies system which uses a novel Bloom filter based fingerprinting technique to efficiently predict the potential for sharing between large numbers of virtual machines, and uses that information to guide server placement. This improves the energy efficiency of data centers and allows them to be more reactive to changing workloads. Memory Buddies works at the virtualization layer and thus requires no advance knowledge of the applications or operating systems running in each VM.

1.2.3 Data Center Resource Management

The dynamic workloads seen by many of the applications running within data centers mean that before long, the initial placement and resource shares given to a virtual machine may become insufficient for its growing demand. Hotspots form within a data center when the resource requirements of one or more VMs on a physical host exceed the host's capacity. The large scale nature and the speed with which workloads can change means that data centers require automated resource management techniques to prevent these hotspots.

Chapter 5 describes how to dynamically adjust resource shares and migrate virtual machines between hosts in order to balance load. Our Sandpiper system includes techniques to automatically detect hotspot formation, calculate new resource shares required for overloaded VMs, and either initiate migrations to balance load or adjust resource allocations to meet each VM's needs. We consider both a completely black-box approach and one which exploits application level information to improve decision making.

As more data centers are built across the world, it becomes increasingly desirable to seamlessly connect them into large jointly managed resource pools. Having the flexibility to group multiple data centers and dynamically move applications between them would enable new approaches to cross data center resource management.

We propose the CloudNet infrastructure in Chapter 6 to seamlessly and securely connect multiple data center sites into flexible resource pools. We then study how virtual machine migration techniques can be adapted to work over the WAN, and present optimizations to allow transparent migration of arbitrary applications over low bandwidth, high latency links.

1.2.4 Reliability and Disaster Recovery

Data centers must provide not only performance guarantees, but reliability ones as well. Disaster Recovery (DR) services attempt to protect applications by continuously replicating state to a secondary data center that can be switched to in the event of catastrophic data center failure. These services come at high cost both economically and in terms of application performance, particularly if no-data-loss consistency guarantees are required.

Chapter 7 proposes a new replication approach which offers synchronous consistency guarantees but uses speculative execution to provide performance on par with asynchronous approaches. Our PipeCloud system uses this replication protocol to protect virtual machines in a black-box manner that makes no assumptions about the applications or operating system inside. PipeCloud replicates VM disks to a backup cloud data center and automates the failover and recovery process when a disaster is detected.

1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 provides background on data centers and virtualization to set the context of our work. The thesis then starts with the challenges faced during the planning and deployment phases of running a modern data center. Chapter 3 describes how to predict the way that resource requirements will change when transitioning an application to a virtualized data center. This is followed in Chapter 4 with an explanation of how memory sharing can be used to guide VM placement when deploying or consolidating servers within a data center. Chapter 5 discusses resource management challenges in data centers, and describes the use of VM migration to handle server hotspots caused by changes in application resource requirements. In Chapter 6 we broaden our view to consider multiple data centers and describe the infrastructure and tools required to move live applications between them. Chapter 7 discusses our work on reliability and using cloud platforms for disaster recovery. Finally, Chapter 8 summarizes the full thesis contributions and discusses potential future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter presents background material on virtualization technologies and data centers to set the context for our contributions. More detailed related work sections are also provided in the remaining chapters.

2.1 Data Centers

Data centers have grown in popularity as the processing power required by businesses exceeds what they can maintain within their own corporate infrastructure. Data centers have sprung up to act as large server farms that use economy of scale to provide computation and storage resources to one or more businesses with greater efficiency. A business may own and operate its own data centers, or a data center may be operated by a service provider that in turn rents shares of its resources to other businesses.

Data center operators face challenges from the initial capacity planning stages of deploying and provisioning new applications, to efficiently allocating resources to meet the application performance guarantees of live systems. At the same time, they must deal with the problems of maintaining the hardware reliability, cooling, and energy needs of running thousands of servers. All of these issues are compounded by both the massive scale of modern data centers, and the fast paced workload dynamics of many data center applications.

Table 2.1 lists four key problem areas for data center operators. Administrators first must deal with infrastructure challenges such as determining data center architectures and providing sufficient cooling and power for large numbers of servers. A current trend in

Area	Challenges
Infrastructure	server & network architecture, cooling & power management
Deployment	capacity planning, **service placement**, **application modeling**
Resource Mgmt	storage, **server, & network provisioning**, monitoring
Reliability	**high availability, fault tolerance**, security
Applications	clustering frameworks, performance, configuration management

Table 2.1. Administrators face challenges related to data center infrastructure, deployment, resource management, and reliability. This proposal covers aspects of the issues listed in bold.

data center architecture is the use of large scale, modular data centers composed of shipping containers filled with servers [65], but more radical proposals range from micro data centers placed inside condominium closets [28] to floating barges filled with servers running off of power generated from ocean currents [31]. The increasing energy consumption of data centers is a growing concern, and work is being done in the "green computing" area to better manage the power and cooling systems in data centers [25, 77, 100].

Next, data center operators must deal with the deployment and planning problems related to estimating a data center's capacity and initial provisioning for new applications [149]. This may require models of an application's resource requirements [113, 148], and an understanding of how they are impacted by different hardware configurations [114]. As data centers attempt to improve resource utilization through server consolidation, it also becomes necessary for data center operators to understand how the placement of applications impacts performance and resource consumption [124].

Efficient resource management is a key concern for data center operators looking to both meet application Service Level Agreements (SLAs) and reduce costs. Shared hosting platforms attempt to multiplex physical resources between multiple customer applications [24, 125]. However, without virtualization, it is difficult to provide strong isolation between applications, and operating systems must be modified to fairly allocate resources [134, 126].

Reliability becomes an important concern when running mission critical applications within data centers. The large scale of modern data centers means that hardware components fail on a constant basis [97], requiring both low level fault tolerance techniques like RAID, and high level reliability mechanisms within applications. Security is also an issue, especially for shared data center environments, leading to much research on isolating services and building trusted platforms [45, 40].

Finally, the massive scale of data centers has led to new distributed application architectures. Clustering of web servers and databases becomes necessary when a single commodity server cannot meet customer demands [89, 75]. Large scale data mining is also an increasingly popular use for data centers, with search engines becoming some of the largest consumers of data center resources. These systems employ clustering frameworks like MapReduce and Dryad to distribute work across many hundreds or thousands of nodes [33, 58].

This work focuses on three of these areas: deployment, resource management, and reliability, with an emphasis on how virtualization can provide improved solutions.

2.2 Server Virtualization

Virtualization is not a new technology, but it has regained popularity in recent years because of the promise of improved resource utilization through server consolidation. The virtualization of commodity operating systems in [21] has led to the wide range of both commercial and open source virtualization platforms available today [10, 131, 72, 55, 95].

Virtualization can be performed at the application level (e.g., Oracle's VirtualBox or VMware's Player software), within an operating system (e.g., KVM or Linux Vservers), or even below the operating system (e.g., VWware ESX). In this thesis we focus on the Xen and VMware full system virtualization platforms. VMware's ESX platform is a full virtualization technique that provides a "bare-metal" hypervisor that manages a set of virtual machines running unmodified operating systems [131]. Xen uses a paravirtualization technique that requires small changes to be applied to the operating systems within each VM and a host OS to run device drivers, but allows for a simpler hypervisor layer [10]. Both systems support fine grain management of memory and CPU resources, as well as the ability to transparently migrate running virtual machines from one physical server to another [30, 91, 133]. In this thesis we make frequent use of the resource management and migration tools provided by each virtualization platform.

2.3 Virtualization in Data Centers

Server virtualization has become popular in data centers since it provides an easy mechanism to cleanly partition physical resources, allowing multiple applications to run in isolation on a single server. Virtualization helps with server consolidation and provides flexible resource management mechanisms, but can introduce new challenges.

Determining where to run applications in a shared environment remains a challenge [124], and virtualization adds new difficulties due to the variable virtualization overheads seen by different applications and platforms [27]. Our work explores a new factor to consider when placing VMs, the potential for memory sharing, and helps build models that characterize VM overheads.

Some commercial systems now exist for automating the management of VM resources [138, 37], and a variety of research projects have proposed schemes for management of processing [43, 105] and memory [150, 80] resources. Our work was some of the first to combine automated management of VM resources with dynamic migration to balance load within commercial data centers.

Reliability is an important feature for data center applications, and virtualization has been used to provide increased resiliency in the face of crash failures [32, 132]. Our work extends these ideas to provide disaster recovery services across data centers, allowing applications to fail over from one to another with no data loss. We propose a new replication approach based on the ideas of external synchrony [92], which uses speculative execution to combine the best aspects of synchronous and asynchronous approaches.

CHAPTER 3

TRANSITIONING TO VIRTUAL DATA CENTERS

Virtualization technologies promise great opportunities for reducing energy and hardware costs through server consolidation. However, to safely transition an application running natively on real hardware to a virtualized environment, one needs to estimate the additional resource requirements incurred by virtualization overheads. In this chapter we propose the use of automated model generation systems to characterize the relationship between native and virtual platforms, easing the transition to virtualized data centers.

3.1 Background and Motivation

Modern data centers employ server virtualization to slice larger, underutilized physical servers into smaller virtual ones. By allowing for greater resource multiplexing, virtualization can decrease energy utilization and hardware costs. While many businesses would like to lower costs by moving their applications from running on physical hardware to virtual machines, they must ensure that this transition will not disrupt application performance by incorrectly estimating the resource requirements of the virtualized application. A naïve solution is to simply monitor the workload of native applications and attempt to provision the virtual servers based on the observed peak resource requirements. However, this does not account for the different types of overhead caused by the virtualization layer, and can lead to either over- or under-provisioning depending on the nature of the application.

In this chapter we present MOVE[1], an automated model generation system which determines the relationship between the native and virtual platforms being used. The overhead

[1]**M**odeling **O**verheads of **V**irtual **E**nvironments

of the virtual platform is characterized by running a series of microbenchmarks on both platforms and building a regression model that relates the resource requirements of one platform to the other. Although it is created using data from synthetic benchmarks, the result is a general model which can be applied to traces from any other application in order to predict what its resource requirements will be on the virtual platform.

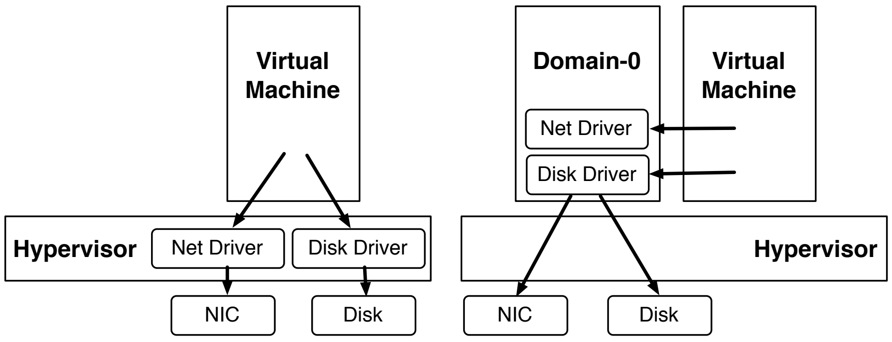
3.2 Virtualization Overheads

Server consolidation is an approach to reduce the total number of servers in response to the problem of server sprawl, a situation in which multiple, under-utilized servers take up more space and consume more resources than can be justified by their workloads. A typical approach for evaluating which workloads can be efficiently consolidated together is based on multi-dimensional "binpacking" of resource usage traces. Under such an approach, each application is characterized by its CPU, I/0 and memory usage over time. Then a binpacking algorithm finds a combination of workloads with resource requirements which do not exceed the available server resources. After the initial workload placement, specialized workload management tools are used [56, 52] to dynamically adjust system resources to support the required application performance.

In our work, we are concerned with the initial workload placement phase that requires as an input the application resource usage traces. We assume that all applications are currently running on a native platform, and thus we must be able to predict how resource requirements (in particular, CPU requirements) will increase due to virtualization overheads. It is important to know what an application's resource needs are going to be prior to transitioning it to the virtual environment. If these overheads are not accounted for during initial planning, an application could be deployed to a server with insufficient resources, resulting in unacceptable application performance.

Virtualization overheads depend on the type and implementation specifics of the virtualization solution [115, 137, 67, 10]. Often, the "amount" of CPU overhead is directly proportional to the “amount” of performed I/O processing [27, 45], however different types of I/O and different virtualization platforms may incur different overheads.

Xen and VMware ESX Server demonstrate the two popular I/O models for VMs. In ESX (and Xen in its original design [10]), the hypervisor itself contains device driver code and provides safe, shared access for I/O hardware (see Figure 3.1 a). Later, the Xen team proposed a new architecture [39] that allows unmodified device drivers to be hosted and executed in isolated “driver domains” (see Figure 3.1 b).



(a) VMware I/O Model (b) Xen I/O Model

Figure 3.1. Two popular Virtualization I/O models.

In Xen, the management domain (Domain-0) hosts unmodified Linux device drivers and plays the role of the driver domain. This I/O model results in a more complex CPU usage model with two components: CPU consumed by the guest virtual machine and CPU consumed by Dom-0 which performs I/O processing on behalf of the guest domain. We demonstrate our approach using Xen running paravirtualized VMs because it presents the additional challenge of modeling both the virtualized application and the driver domain (Dom-0) separately.

Given resource utilization traces of an application running natively, we aim to estimate what its resource requirements would be if the application were transitioned to a virtual environment on a given hardware platform. For example, let a collection of application resource usage profiles (over time) in the native system be provided as shown in Figure 3.2 (top): *i)* CPU utilization, *ii)* transferred and received networking packets, *iii)* read and written disk blocks.

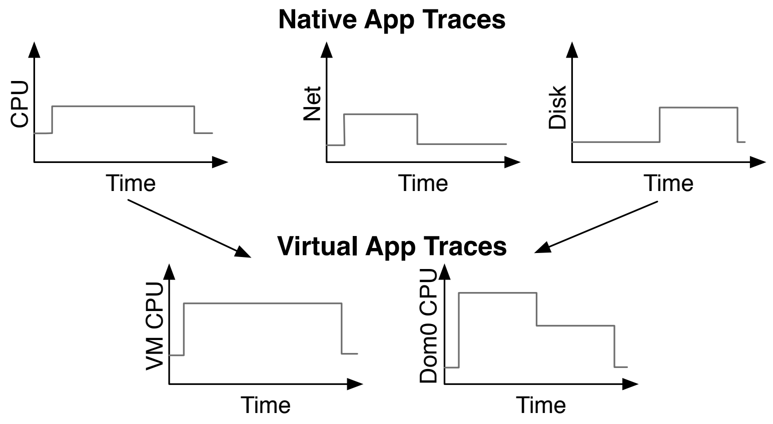


Figure 3.2. Using native application traces to predict resource needs in virtual environments.

The goal of MOVE is to estimate the CPU requirements of the virtual machine running the application and of Dom-0 which performs I/O processing on behalf of the guest VM. Intuitively, we expect that CPU utilization of the VM is highly correlated and proportional to the native CPU usage profile of the application, while Dom-0 CPU utilization is mostly determined by a combination of I/O profiles (both network and disk).

We focus on estimating only CPU utilization since other metrics (such as disk and network request rates) are not directly impacted by the virtualization layer–running an application in a virtualizated environment will not cause more packets to be sent over the network or more disk requests to be generated. Instead, the virtualization layer incurs additional processing overheads when I/O is performed; it is these overheads which our models seek to capture. [2]

Our Approach: MOVE is an automated model generation system which determines the relationship between the native and virtual platforms being used. The overhead of the

[2]Virtualization also incurs a memory overhead. Both Xen and ESX Server require a base allocation for Dom-0 or the Service Console, plus a variable amount per VM.

virtual platform is characterized by running a series of microbenchmarks on both platforms and building a model that relates the resource requirements on one platform to the other. Although it is created using data from synthetic benchmarks, the result is a general model which can be applied to traces from any other application in order to predict what its resource requirements will be on the virtual platform.

3.3 Profiling Native & Virtual Platforms

In order to characterize the overhead of a specific virtualization platform, we propose running a set of microbenchmarks which define a *platform profile*. The same microbenchmarks are then run on a native hardware system in order to produce a second profile. As the microbenchmarks run, resource utilization traces are gathered to define the platform profile. These profiles are later used to define the model that relates the native and virtual platforms.

3.3.1 Microbenchmark Requirements

The microbenchmarks used to generate platform profiles must meet two main criteria:

- *Microbenchmarks must be able to apply a range of workload intensities.*
- *Microbenchmarks should function nearly-identically in native and virtual environments.*

The first requirement allows the microbenchmarks to mimic the variable loads commonly seen by enterprise applications, while the second ensures that the activities occurring on both the native and virtual platforms are identical during the microbenchmarks, allowing us to relate the resource consumption on one to the other. Therefore, we concentrate on creating a set of microbenchmarks that can generate *repeatable* workloads at *varying intensities*.

3.3.2 Microbenchmark Workloads

The selected microbenchmarks have to create a set of workloads that utilize different system resources and have a different range of workload intensities. MOVE uses a client-server style setup for its benchmarks. In general, a client machine issues a set of requests to the benchmark server running on the system being profiled. The clients adjust the rate and type of requests to control the amount of CPU computation and I/O activities performed on the test system. At a high level, our microbenchmarks are comprised of three basic workload patterns that either cause the system to perform CPU intensive computation, send/receive network packets, or read/write to disk.

- Our *computation intensive* workload calculates Fibonacci series when it receives a request. The number of terms in the series is varied to adjust the computation time.
- The *network intensive* workload has two modes depending on the type of request. In transmit mode, each incoming request results in a large file being sent from the system being tested to the client. In receive mode, the clients upload files to the benchmark application. The size of transferred files and the rate of requests is varied to adjust the network utilization rate.
- The *disk intensive* workload has read and write modes. In both cases, a random file is either read from or written to a multilevel directory structure. File size and request rate can be adjusted to control the disk I/O rate.

Each workload is created by adjusting the request type sent to the server from the client machines. We split each of the basic benchmark types, CPU-, network-, and disk-intensive, into five different intensities ranging from 10% load to 90% load. The maximum load that a server can handle is determined by increasing the throughput of benchmark requests until either the virtual machine or Dom-0 CPU becomes saturated during testing. To create more complex and realistic scenarios, we use a *combination* workload that exercises all three of the above components. The combination workload simultaneously sends requests of all types to the benchmarked server. The relative intensity of each request type is varied in order to provide more realistic training data which does not focus exclusively on a single form of I/O.

The microbenchmarks are implemented as a set of PHP scripts running on an Apache web server in the benchmarked server. Basing the microbenchmarks on Apache and PHP has the benefit that they can be easily deployed and executed on a wide range of hardware platforms within a software environment which data center administrators are already familiar with. MOVE's microbenchmark suite allows us to generate a diverse set of simple and more complex workloads that exercise different system components.

The client workloads are generated using *httperf* [88] and Apache JMeter [5]. These tools provide flexible facilities for generating variable and fixed rate HTTP workloads. The workloads can then be easily "replayed" in different environments. Both tools can emulate an arbitrary number of clients accessing files on a webserver.

3.3.3 Platform Resource Usage Profiles

MOVE generates *platform profiles* by gathering resource utilization traces while the microbenchmarks are running. Within the native system, we currently gather information about *eleven* different resource metrics related to CPU utilization, network activity, and disk I/O. The full list of metrics is shown in Table 3.1. These statistics can all be gathered easily in Linux with the sysstat monitoring package [119].

CPU	**Network**	**Disk**
User Space %	Rx packets/sec	Read req/sec
Kernel %	Tx packets/sec	Write req/sec
IO Wait %	Rx bytes/sec	Read blocks/sec
	TX bytes/sec	Write blocks/sec

Table 3.1. Resource Utilization Metrics

We monitor three CPU related metrics since different types of activities may have different virtualization overheads. For example, user space processing such as simple arithmetic operations performed by an application are unlikely to have much overhead in current virtualization platforms. In contrast, tasks which occur in kernel space, such as context switches, memory management, and I/O processing, are likely to have a higher level of overhead since they can require traps to the hypervisor.

We measure both the packet rates and byte rates of the network interfaces since different platforms may handle I/O virtualization in different ways. For example, prior to Xen version 3.0.3, incoming network packets were passed between Dom-0 and the guest domain by flipping ownership of memory pages, thus the overhead associated with receiving each packet was independent of its size [45]. Newer versions of Xen directly copy packets from Dom-0 to the guest domain rather than using page flipping, thus the overhead is also related to the number of bytes received per second, not just the number of packets. We differentiate between sending and receiving since these paths may have different optimizations. We split disk measurements into four categories based on similar reasoning.

A resource usage trace is gathered for each benchmark set containing values for all metrics listed in Table 3.1, plus the time interval, and benchmark ID. After the resource metrics have been gathered on the native system, the Dom-0 and VM CPU utilizations are measured for the identical benchmark on the virtualized platform.

3.4 Generating Overhead Models

This section describes how to create models which characterize the relationship between a set of resource utilization metrics gathered from an application running natively on real hardware and the CPU requirements of the application if it were run on a virtual platform. Two models are created: one which predicts the CPU requirement of the virtual machine running the application, and one which predicts the Dom-0 CPU requirements when it performs I/O processing on behalf of the guest domain.

The model creation employs the following *three key components*:

- A *robust linear regression* algorithm that is used to lessen the impact of outliers.

- A *stepwise regression* approach that is employed to include only the most statistically significant metrics in the final model.
- A *model refinement* algorithm that is used for post-processing the training data to eliminate or rerun erroneous benchmarks and to rebuild a more accurate model.

Model Creation

To find the relationship between the application resource usage in native and virtualized systems we use the resource usage profile gathered from a set of microbenchmarks run in both the virtual and native platforms of interest (see Section 3.3.3).

Using values from the collected profile, we form a set of equations which calculate the Dom-0 CPU utilization as a linear combination of the different metrics:

$$\begin{aligned} U^1_{dom0} &= c_0 + c_1 * M^1_1 + c_2 * M^1_2 + ... + c_{11} * M^1_{11} \\ U^2_{dom0} &= c_0 + c_1 * M^2_1 + c_2 * M^2_2 + ... + c_{11} * M^2_{11} \\ .... & \qquad .... \end{aligned} \tag{3.1}$$

where

- M^j_i is a value of metric M_i collected during the time interval j for a benchmark executed in the native environment;
- U^j_{dom0} is a measured CPU utilization for a benchmark executed in virtualized environment with the corresponding time interval j.

Let $c^{dom0}_0, c^{dom0}_1, ..., c^{dom0}_{11}$ denote the coefficients that approximate the solution for the equation set (3.1). Then, an approximated utilization $\hat{U}^j_{dom0}$ can be calculated as

$$\hat{U}^j_{dom0} = c^{dom0}_0 + \sum_{i=1}^{11} M^j_i \cdot c^{dom0}_i \tag{3.2}$$

To solve for c^{dom0}_i $(0 \leq i \leq 11)$, one can choose a regression method from a variety of known methods in the literature. A popular method for solving such a set of equations is Least Squares Regression that minimizes the error:

$$e = \sqrt{\sum_j (\hat{U}^j_{dom0} - U^j_{dom0})^2_j}$$

The set of coefficients $c_0^{dom0}, c_1^{dom0}, ..., c_n^{dom0}$ is the model that describes the relationship between the application resource usage in the native system and CPU usage in Dom-0 on behalf of the virtualized application.

We form a set of equations similar to Eq. 3.1 which characterize the CPU utilization of the VM by replacing U_{dom0}^i with U_{vm}^i. The solution $c_0^{vm}, c_1^{vm}, ..., c_n^{vm}$ defines the model that relates the application resource usage in the native system and CPU usage in the VM running the application. To deal with outliers and erroneous benchmark executions in collected data, and to improve the overall model accuracy, we apply a more advanced variant of the regression technique as described below.

Robust Stepwise Linear Regression: To decrease the impact of occasional bad measurements and outliers, we employ iteratively reweighted least squares [50] from the Robust Regression family. The robust regression technique uses a bisquare weighting function which lessens the weight and the impact of data points with high error.

In order to create a model which utilizes only the statistically significant metrics and avoids "overfitting" the data, we use stepwise linear regression to determine which set of input metrics are the best predictors for the output variable [34]. Step-wise regression starts with an "empty" model that includes none of the eleven possible metrics. At each iteration, a new metric is considered for inclusion in the model. The best metric is chosen by adding the metric which results in the lowest mean squared error when it is included. Before the new metric is included in the model, it must pass an F-test which determines if including the extra metric results in a statistically significant improvement in the model's accuracy. If the F-test fails, then the algorithm terminates since including any further metrics cannot provide a significant benefit. The coefficients, for the selected metrics are calculated using the robust regression technique described previously. The coefficient for each metric not included in the model is set to zero.

Model Refinement: MOVE's use of robust linear regression techniques helps lessen the impact of occasional bad data points, but it may not be effective if all measurements within a microbenchmark are corrupt (this can happen due to unexpected background processes on the server, timing errors at the client, or network issues). If some microbenchmarks have failed or collected data were corrupted then it can inevitably impact the model accuracy.

In order to automate the model generation process and eliminate the need for manual analysis of these bad data points, we must automatically detect erroneous microbenchmarks and either rerun them or remove their data points from the training set. At runtime, it can be very difficult to determine whether a benchmark is executed correctly, since the resource utilization cannot be known ahead of time, particularly on the virtual platform which may have unpredictable overheads. Instead, we wait until all benchmarks have been run and an initial model has been created to post process the training set and determine if some benchmarks have anomalous behavior.

First, we compute the mean squared error and standard deviation of the squared errors when applying the calculated model to the full microbenchmark set. We then apply the calculated model to the data from each benchmark run individually; if its error rate is significantly larger than the mean square error across all the benchmarks then it is likely that the benchmark did not run correctly. MOVE's automated system detects this and attempts to rerun the failed benchmark and regenerate the model.

Model Usage: Once a model has been created, it can then be applied to resource utilization traces of other applications in order to predict what their CPU requirements would be if transferred to the virtual environment. Resource usage traces of the application are obtained by monitoring the application in its native environment over time. The traces must contain the same resource metrics as presented in Table 3.1, except that CPU utilizations of VM and Dom-0 are unknown and need to be predicted. Applying the model coefficients $c_0^{dom0}, c_1^{dom0}, ..., c_{11}^{dom0}$ and $c_0^{vm}, c_1^{vm}, ..., c_n^{vm}$ to the application usage traces in native environment (using Equation 3.1), we obtain two new CPU usage traces that estimate the application CPU requirements in Dom-0 and the virtual machine.

3.5 Experimental Evaluation

In this section, we first try to justify a set of our choices presented in earlier Sections 3.3 and 3.4: why these *metrics*? why these *microbenchmarks*? why this *model creation process*? After that, we evaluate the effectiveness of our models under several realistic web application workloads on two different hardware platforms.

3.5.1 Implementation Details

Our implementation and evaluation has centered on the Xen virtualization platform. In our evaluation, both the native systems and virtual machines run the Red Hat Enterprise Linux 5 operating system with Linux kernel 2.6.18-8. We use paravirtualized Xen version 3.0.3-rc5.

Monitoring resource utilization in the native environment is done with the sysstat package [119] commonly used in Linux environments. The virtual CPU utilizations are measured using xentop and xenmon, standard resource monitoring tools included with the Xen distribution. Statistics are gathered for 30 second monitoring windows in both environments. We have experimented with both finer grain and longer intervals and found similar results. The system is configured that Dom-0 resides on a separate CPU.

We evaluate our approach using two realistic web applications:

- *RUBiS* [23] is an auction site prototype modeled after eBay.com. A client workload generator emulates the behavior of users browsing and bidding on items. We use the Apache/PHP implementation of RUBiS version 1.4.3 with a MySQL database.

- *TPC-W* [122] represents an e-commerce site (modeled after Amazon.com) implemented with Java servlets running on Tomcat with a MySQL database.

Both applications have an application and a database tier. We profile and predict the resource requirements of the application server tier; the databases are hosted on a separate server which is sufficiently provisioned so that it will not become a bottleneck.

We have tested our approach on two different hardware platforms:

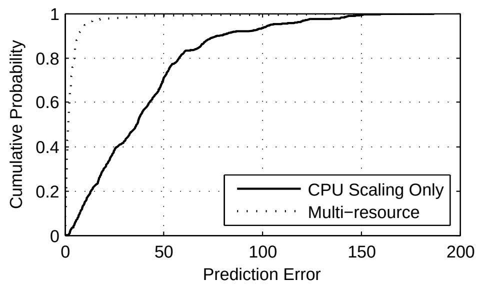
- HP ProLiant DL385, 2 processors: AMD Opteron model 252 2.6GHz with 1MB L2 single-core, 64-bit; 2 x 2GB memory; 2 x 1 Gbit/s NICs, 72 GB 15K U320 Disk.
- HP ProLiant DL580 G2, 4 processors: Intel Xeon 1.6 GHz with 1MB L2 cache, 32-bit; 3 x 2GB memory; 2 x 1 Gbit/s NICs, 72 GB 15K U320 Disk.

3.5.2 Importance of Modeling I/O

MOVE generates models based on up to eleven different resource utilization metrics, here we evaluate whether such complexity is warranted, or if a simple model based solely on scaling CPU requirements is a viable approach. In the simplified approach, a model is created using the same model generation techniques as described in Section 3.4, except that instead of using all eleven metrics, only a single Total CPU metric is used to predict the CPU needs in virtual environment. We produce a model using each technique to predict the CPU requirements and demonstrate it using the CPU needs of the guest domain, since, intuitively, it is more likely that the simplified model will perform better when predicting VM CPU needs than when predicting Dom-0 since the latter is scheduled almost exclusively for handling I/O.

Since our models are created with stepwise regression, not all of the eleven possible metrics are included in MOVE's final model. The Dom-0 model uses five metrics: Kernel CPU, I/O Wait, Rx Packets/sec, Tx Packets/sec, and Disk Write Req/sec. Dom-0's CPU utilization is dominated by I/O costs, so a large number of I/O related metrics are important for an accurate model. In contrast the virtual machine model uses only three metrics: User Space CPU, Kernel CPU, and RX Packets. We compare MOVE's multi-resource VM model to the CPU-Scaling based model which uses only the Total CPU metric (equal to the sum of User Space and Kernel CPU).

We evaluate the performance of these two models by training them on our microbenchmark set and then comparing the error when the models are applied back to the training data. Figure 3.3 (a) shows the error CDF for each model, showing the probability that our predictions were within a certain degree of accuracy for the virtual machine.



(a)

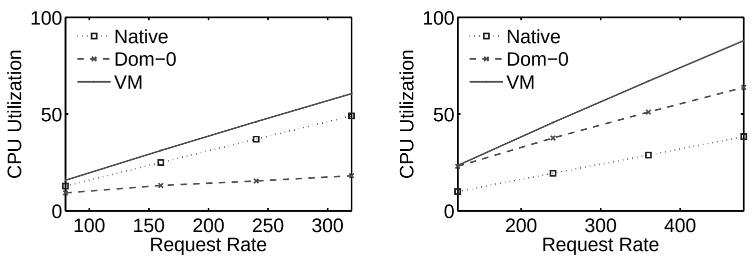
		Test Set Median Error %		
		CPU	Net	Disk
Training Set	CPU	0.36	670	13
	Net	11	3.4	1
	Disk	7.1	1798	1.2
	All	0.66	1.1	2.1

(b)

Figure 3.3. (a) Using CPU as the only prediction metric leads to high error. (b) Using a subset of benchmarks leads to poor accuracy when applied to data sets with different type of I/O.

MOVE's multiple resource model performs significantly better than the CPU scaling approach; the 90^{th} error percentile using our approach is 5% while the scaling approach is 65%. Without information about I/O activities, the simple model cannot effectively distinguish between the different types of benchmarks, each of which has different levels of overhead. Even though the VM model only includes one I/O metric, splitting CPU into User and Kernel time acts as a surrogate for detecting high levels of I/O. Our results suggest that I/O activity can cause significant changes in the CPU requirements of both Dom-0 and the guest domain: Dom-0 since it must process the I/O requests, and the guest because of the increased number of hypercalls required for I/O intensive applications.

Figure 3.4 presents profiles of some of our CPU and network intensive microbenchmarks. The CPU intensive application exhibits only a small virtualization overhead occurring for the VM CPU requirements and Dom-0 also has relatively low CPU needs. In contrast, the network intensive application has a significantly higher requirement in Dom-0 as well as a much larger increase in VM CPU requirements relative to the native CPU utilization. This further demonstrates why creating a model using only the native CPU metric is incapable of capturing the differences in overhead caused by I/O requests.



(a) CPU Intensive (b) Network Intensive

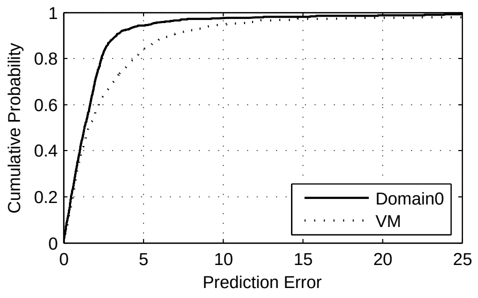
Figure 3.4. I/O intensive applications exhibit higher virtualization overheads.

3.5.3 Benchmark Coverage

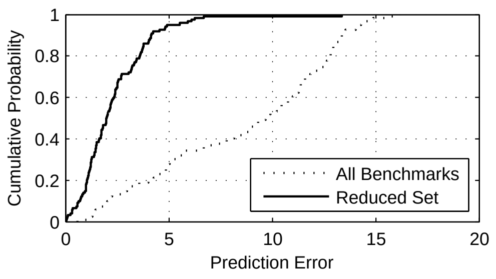
In this experiment we examine how the three different benchmark types each add useful information and examine the training set error of our model. Figure 3.3 (b) illustrates how using only a single type of microbenchmark to build a model can produce very high error rates when applied to applications with different workload characteristics.

For example, training the model solely with the CPU intensive microbenchmarks provides accuracy within 1% when applied back to the same kind of CPU intensive workloads, but the median error rises to 670% when applied to the network intensive data. This happens because the CPU benchmark includes only very low network rates. When a model based solely on that data tries to predict the CPU needs of the network intensive applications, it must extrapolate well beyond the range of data it was trained with, resulting in wildly inaccurate numbers. The bottom row in the table corresponds to using *all* of the benchmark data to create a model. This provides a high degree of accuracy in all cases – while a specialized model may provide higher accuracy on data sets very similar to it, we seek to build a general model which will be effective on workloads with a range of characteristics.

Figure 3.5(a) shows the error CDF when *all* of our benchmark data is used to create a model and then the model is validated by applying back to the training set. The error is quite low, with 90% of the predictions being within 3% for Dom-0 and 7% for the virtual



(a) Training Error



(b) BM Elimination

Figure 3.5. (a) CDF error of the training set on the Intel 4 -CPU machine. (b) Automatic benchmark elimination can increase model accuracy

machine. This confirms our hypothesis that a single linear model can effectively model the full range of training data.

3.5.4 Benchmark Error Detection

Our profiling system runs a series of microbenchmarks with identical workloads on both the native and virtual platforms. This experiment tests our anomalous benchmark detection algorithm. To be effective, it should be able to detect which benchmarks did not run correctly so that they can be either rerun or eliminated from the training set. If the detection scheme is too rigorous, it may eliminate too many data points, reducing the effectiveness of the model.

We first gather a set of training data where 10 percent of the benchmarks are corrupted with additional background processes. Figure 3.5(b) shows the change in model accuracy after the error detection algorithm eliminates the malfunctioning microbenchmarks. We then gather a second training set with no failed benchmarks and run the error detection algorithm on this clean data set. We find that the model performance before and after the error detection algorithm is identical since very few data points are eliminated.

While it is possible for these errors to be manually detected and corrected, our goal is to automate the model creation procedure as much as possible. The error detection algorithm reduces the human interaction required to create high quality models.

3.5.5 Model Accuracy

To test the accuracy of a model, we use MOVE to generate a model for our Intel native and virtualized platform. We then use this generic model to predict the resource needs of the RUBiS and TPC-W web applications.

We first create a variable rate workload for RUBiS by incrementally spawning clients over a thirty minute period. The system is loaded by between 150 and 700 simultaneous clients. This workload is repeated twice to evaluate the amount of random variation between experiments. We record measurements and make predictions for 30 second intervals. Figure 3.6 compares the actual CPU utilization of the RUBiS application to the amount predicted by MOVE's model. Note that the virtual machine running RUBiS is allocated two virtual CPUs, so the percent utilization is out of 200.

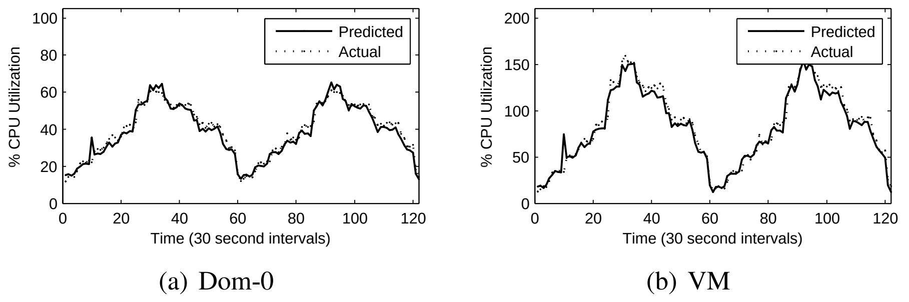
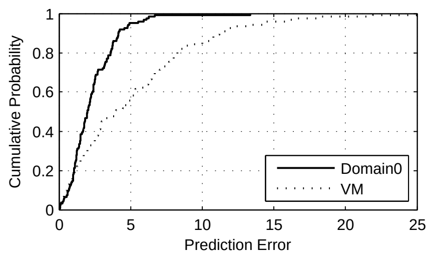
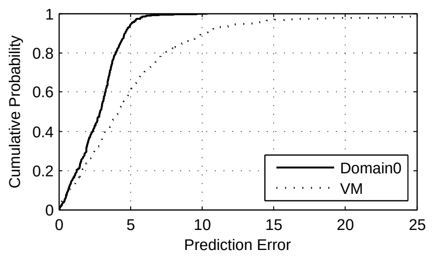


Figure 3.6. Prediction accuracy of the RUBiS web application.

Figure 3.7(a) shows a CDF of MOVE's prediction error. We find that 90% of our predictions for Dom-0 are within 4% accuracy, and within 11% for predicting the virtual machine's CPU utilization. Some of this error is due to model inaccuracy, but it can also be due to irregularities in the data used as input to the model. For example, there is a spike in the predicted CPU requirements of both Dom-0 and the VM around time interval 10. This spike was caused by a background process running for a short period when RUBiS was run in the native environment. Since the predicted values are based on these native measurements, they mistakenly predict the virtual CPU requirements to spike in the same way.



(a) RUBiS



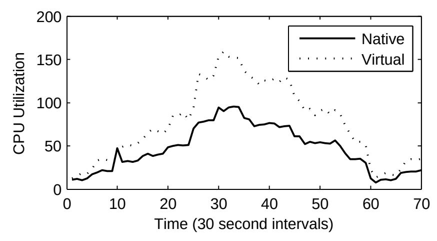
(b) TPC-W

Figure 3.7. Error rates on the Intel platform.

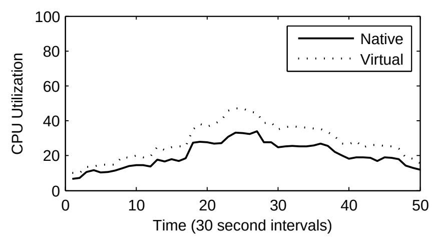
We have also validated MOVE's model on the TPC-W application. We create a changing workload by adjusting the number of emulated clients from 250 to 1100 in a random (but repeatable) pattern. Figure 3.7(b) presents the error distribution for TPC-W. The error for this application is almost identical to RUBiS, with 90^{th} percentile error rates of 5% and 10% for Dom-0 and the virtual machine respectively.

3.5.6 Cross Platform Modeling

In many server consolidation scenarios, the transition from a native to a virtual platform is accompanied by a change in the underlying hardware. However, using a single model for multiple hardware platforms may be ineffective if they have different overhead costs. Attempting to apply the model for the Intel system to the AMD system results in high error rates as shown in Figure 3.9(a). To investigate why these two platforms exhibit such a large difference, we compare the CPU required by the RUBiS application in the native and virtual environments on both platforms in Figure 3.8. Not including the Dom-0 requirements, the Intel system requires approximately 1.7 times as much CPU in the virtual case as it does natively. On the AMD system, the increase is only about 1.4 times. The different scaling between the native and virtual traces in each platform suggest that a single model cannot be used for both platforms.



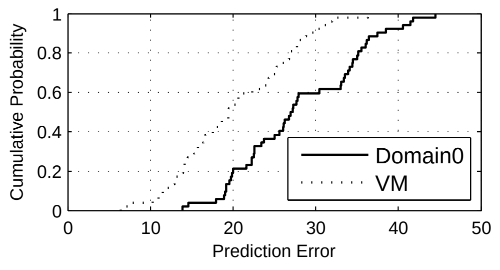
(a) Intel



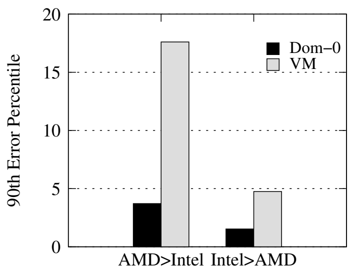
(b) AMD

Figure 3.8. Comparison of CPU overhead on different hardware platforms.

We test MOVE's ability to determine the relationship between native and virtual systems running on different hardware platforms by executing an identical set of microbenchmarks on the Intel and AMD platforms in both the native and virtual environments. Using this data, we create two models, one which relates a native usage profile of the Intel platform to a virtual usage profile of the AMD system and one which relates the native AMD system to the virtualized Intel system.



(a)



(b)

Figure 3.9. (a) Using a single model for different architectures is ineffective, (b) but cross platform models are feasible.

Figure 3.9(b) presents the 90^{th} error percentiles when these cross platform models are used to predict the CPU needs of both the TPC-W and RUBiS workloads. The cross platform models are very effective at predicting Dom-0 CPU needs, however the VM prediction error is higher, particularly for the AMD to Intel model. We propose two factors which may cause this jump in error. First, the AMD system has a significantly faster CPU than the Intel system, so translating the CPU component from one platform to the other requires a significant scale up factor. As a result, small variations in the CPU needs of the AMD system can result in larger fluctuations in the predicted CPU for the Intel system, leading to higher absolute error values. Secondly, cross platform models for predicting virtual machine CPU are typically more difficult than Dom-0 models. This is because Dom-0 models are predominantly based on I/O metrics such as packet reception rates and disk operations, which have similar costs on both platforms. In contrast, the VM model is primarily based on the CPU related metrics which may not have a linear relationship between the two platforms due to differences in the processor and cache architectures. However, it should be noted that in many cases, the AMD to Intel model performs better than the 90^{th} error percentile indicates; the median error is only 5%, and all of the points with high error occur at the peaks of the RUBiS workload where the virtual CPU consumption exceeds 160%.

3.6 Discussion

In this section, we discuss the impact of the application behavior on the accuracy of the prediction results and challenges introduced by dynamic frequency scaling.

Impact of application behavior on resource use: The timing for an application's operations in the native and virtualized environments may be slightly different if the application has a strong "feedback loop" behavior.

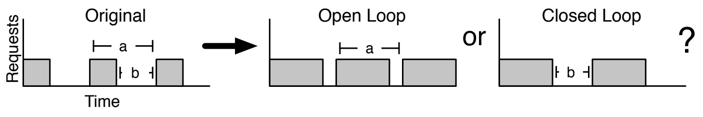


Figure 3.10. Resource requirements in different environments is influenced by the amount of feedback in an application's workload.

Figure 3.10 illustrates the difference between an application with (closed loop) and without (open loop) feedback. In the original application trace, a series of requests arrive, with their processing time indicated by the width of the rectangles. The value of a represents the time from the start of one request until the start of the next, while b is the time from the end of one request to the start of the next. When the same application is run on a different platform, the time to process a request may increase due to virtualization overhead. The two move/figures on the right represent how the trace would appear if the application does or does not exhibit feedback. With an open loop, the time between the start of each request will remain a, even if the request processing time increases. This would occur if the requests are being submitted by a client on another machine sending at a regular rate. For an application with feedback, requests are processed then a constant delay, b, occurs before the next request is processed. The figure illustrates that when request processing times increase, applications with feedback may process fewer requests in a given time interval (due to a slowdown), i.e., its CPU overhead is "spread" across a longer time period, resulting in lower average CPU utilization.

It is impossible to tell if an application's workload has a feedback loop just by looking at resource utilization traces of the original application. So the estimated resource utilization produced by our model for the application with a "feedback loop" might be higher than in reality since such an application might consume CPU resources in virtualized environment "slower" than in native one due to the increased latency on the application's critical path.

Understanding Application Performance: While our models can accurately predict the changes in resource requirements for a virtualized application, they cannot directly model how application performance (ie. response time) will change. Unfortunately, this is a difficult challenge, akin to making performance predictions under different hardware platforms. Our approach tells system administrators the minimum amount of resources which must be allocated to a VM in order to prevent significantly reduced performance due to resource starvation. The application may still see some performance penalty due to the longer code path as requests go through the virtualization layer. To accurately predict this performance change would necessitate carefully tailored, application specific models.

Our approach helps in estimating the resource requirements that are necessary for the initial application placement in a virtualized environment. After the initial workload placement, specialized workload management tools may be used [56, 52] to dynamically adjust system resources to support the required application performance.

3.7 Related Work

Virtualization Overheads: Virtualization is gaining popularity in enterprise environments as a software-based solution for building shared hardware infrastructures. VMware and IBM have released benchmarks [129] for quantifying the performance of virtualized environments. These benchmarks aim to provide some basis for comparison of different hardware and virtualization platforms in server consolidation exercises. However, they both are lacking the ability to characterize virtualization overhead compared to a native platform.

Application performance and resource consumption in virtualized environments can be quite different from its performance and usage profile on native hardware because of additional virtualization overheads (typically caused by I/O processing) and interactions with the underlying virtual machine monitor (VMM). Several earlier papers which describe various VMM implementations include performance results that measure the impact of virtualization overhead on microbenchmark or macrobenchmark performance (e.g., [11, 82, 133, 115, 3, 137, 67, 115, 27, 94]). The reported virtualization overhead greatly depends on the hardware platform that is used in such experiments. For example, previously published papers [11, 39] evaluating Xen's performance have used networking benchmarks in systems with limited network bandwidth and high CPU capacity. However, there are cases where throughput degrades because CPU processing is the bottleneck instead of the network [86, 45]. In many virtualization platforms, the "amount" of CPU overhead is directly proportional to the "amount" of performed I/O processing [27, 45]. For example, it has been shown that networking packet rates are highly correlated with the measured CPU overhead [45]. Recent work attempts to reduce the performance penalty of network I/O by bypassing parts of the virtualization layer [79, 135] or optimizing it [106]. However, since these optimizations typically target only one source of virtualization overhead (network I/O), our modeling system can still be employed to provide useful information about the level of overhead incurred by a wider range of activities.

This extensive body of previous work has motivated us to select a set of microbenchmarks that "probe" system resource usage at different I/O traffic rates (both networking and disk) and then employ these usage profiles for predicting variable CPU overhead of virtualized environments.

Trace-based Approaches: In our work, we chose to represent application behavior via resource usage traces. Many research groups have used a similar approach to characterize application behavior and applied trace-based methods to support what-if analysis in the assignment of workloads to consolidated servers [126, 104, 109, 41]. There are a few commercial tools [51, 130, 57] that employ trace-based methods to support server consolidation exercises, load balancing, ongoing capacity planning, and simulating placement of application workloads to help IT administrators improve server utilization. Since many virtualization platforms introduce additional virtualization overhead, the trace-based capacity planning and management solutions provide a capability to scale the resource usage traces of original workloads by a specified CPU-multiplier. For some applications it might be a reasonable approach, however, in general, additional CPU overhead highly depends on system activities and operations performed by the application. Simplistic trace-scaling may result in significant modeling error and resource over-provisioning.

System Profiling: Finally, there is another body of work [84, 112, 20, 108] that is closely related to our thinking and the approach presented in this chapter. This body of works goes back to 1995, when L. McVoy and C. Staelin have introduced the *lmbench* – a suite of operating system microbenchmarks that provides a set of portable programs for use in cross-platform comparisons. Each microbenchmark was purposely created to capture some unique performance problem present in one or more important applications. Although such microbenchmarks can be useful in understanding the end-to-end behavior of a system, the results of these microbenchmarks provide little information to indicate how well a particular application will perform on a particular system. In [20, 108], the authors argue for an application-specific approach to benchmarking. The authors suggest a vector-based approach for characterizing an underlying system by a set of microbenchmarks (e.g., *lmbench*) that describe the behavior of the fundamental primitives of the system. The results of these microbenchmarks constitute the *system* vector. Then they suggest to construct an *application* vector that quantifies the way that the application makes use of the various primitives supported by the system. The product of these two vectors yields a relevant performance metric. There is a similar logic in our design: we use a set of microbenchmarks to characterize underlying system and virtualization solution. Then we apply the derived model (analogy to a *system* vector) to the application usage traces (analogy to the *application* vector) and use it for predicting the resource requirements of applications when they are transferred to a virtual environment.

3.8 MOVE Conclusions

This chapter has motivated the need for improved estimates of application resource requirements when they are consolidated to virtual environments. To this end, we designed MOVE, an automated approach for profiling different types of virtualization overhead on a given platform with regression-based models that map the native system profile into a virtualized one. This model can then be used to accurately assess the required resources and make workload placement decisions in virtualized environments.

Although MOVE uses data only from synthetic benchmarks, the result is a general model which can be applied to traces from any other application in order to predict what its resource requirements will be on the virtual platform. Our evaluation has shown that MOVE effectively characterizes the different virtualization overheads of two diverse hardware platforms and that the models have median prediction error of less than 5% for both RUBiS and TPC-W.

CHAPTER 4
MEMORY SHARING GUIDED VM PLACEMENT

The previous chapter discussed how to predict the resource requirements of an application when it is moved to a virtual environment. The next step in deploying applications within a virtualized data center is to determine where to run each virtual machine and how to colocate them together. One of the primary bottlenecks that limits how many VMs can be grouped on a single server is the available memory. In this chapter we present how memory sharing between virtual machines can be used to guide VM placement and consolidate applications onto a smaller number of physical hosts.

4.1 Background and Motivation

Modern hypervisors use a technique called *content-based page sharing (CBPS)* [133, 69] to intelligently share RAM across VMs. In this technique, duplicate copies of a page resident on a host are detected and a single copy of the page is shared, thereby reducing the memory footprint of resident VMs. Today this technique is widely deployed in VMware ESX, with experimental support in Xen [87, 69]. The potential benefits of content-based page sharing are well documented; for instance, the original VMware ESX paper [133] reports memory savings of as much as 33% in measured production environments. Support for memory sharing at finer, sub-page granularity can save more than 65% [47].

However, a CBPS mechanism by itself only shares redundant pages *after* a set of VMs have been placed onto a physical host—the mechanism does not address the problem of *which* VMs within the data center to colocate onto each host so that page sharing can be maximized. This leaves this responsibility to system administrators who must make placement decisions based on knowledge of the applications and operating systems running within each virtual machine. Unfortunately, relying on this expert knowledge cannot scale to the massive size of modern data centers. Thus, to fully realize the benefits of this mechanism, a data center should implement an intelligent colocation strategy that automatically identifies virtual machines with high sharing potential and then maps them onto the same host. Such a colocation strategy can be employed both during the initial placement of a new VM as well as during a server consolidation phase in order to consolidate existing VMs onto a smaller number of physical hosts.

In this chapter we present *Memory Buddies*, a system for intelligent VM colocation within a data center to aggressively exploit page sharing benefits. The key contribution of this work is a memory fingerprinting technique that allows Memory Buddies to quickly identify VMs with high page sharing potential. The memory fingerprints are compact representations of the memory contents of virtual machines; these fingerprints may be compared to determine the number of redundant pages between VMs and thus the potential for memory savings.

Our second contribution is an intelligent VM colocation algorithm that utilizes our memory fingerprinting techniques to identify VMs with high page sharing potential and colocate them onto the same host. Finally, we have gathered a large set of real memory usage data from nearly two dozen Linux and Mac OS X servers, laptops, and desktops in our department to help study the true potential for sharing between diverse sets of machines.

4.2 Problem and System Overview

Consider a typical virtualized data center where each physical server runs a hypervisor and one or more virtual machines. Each VM runs an application or an application component and is allocated a certain slice of the server's physical resources such as RAM and CPU. All storage resides on a network file system or a storage area network, which eliminates the need to move disk state if the VM is migrated to another physical server [30].

The hypervisor uses a content-based page sharing mechanism, which detects duplicate memory pages in resident VMs and uses a single physical page that is shared by all such VMs. If a shared page is subsequently modified by one of the VMs, it is unshared using copy-on-write [133]. Thus, if VM_1 contains M_1 unique pages, and VM_2 contains M_2 unique pages, and S of these pages are common across the two VMs, then page sharing can reduce the total memory footprint of two VMs to $M_1 + M_2 - S$ from $M_1 + M_2$. The freed up memory can be used to house other VMs, and enables a larger set of VMs to be placed on a given cluster.

Problem formulation: Assuming the above scenario, the VM colocation problem is one where each VM must be colocated with a set of other "similar" VMs with the most redundant pages; the best placement strategy will allow the set of VMs to be packed onto the smallest number of servers. Several instantiations of the smart colocation problem arise during: (i) *initial placement*, (ii) *server consolidation* and (iii) *offline planning*.

During initial placement the data center servers must map a newly arriving VM onto existing servers so as to extract the maximum page sharing. For server consolidation, VMs need to be repacked onto a smaller number of servers (allowing the freed up servers to be retired or powered off). Offline planning is a generalization of initial placement where a set of virtual machines must be partitioned into subsets and mapped onto a set of physical server to minimize the total number of servers.

In each case, the problem can be reduced to two steps: (i) identify the page sharing potential of a VM with several candidate VM groups and (ii) pick the group/server that provides the best sharing/memory savings. In scenarios such as server consolidation, live migration techniques will be necessary to move each VM to its new home (server) without incurring application down-time [91, 30].

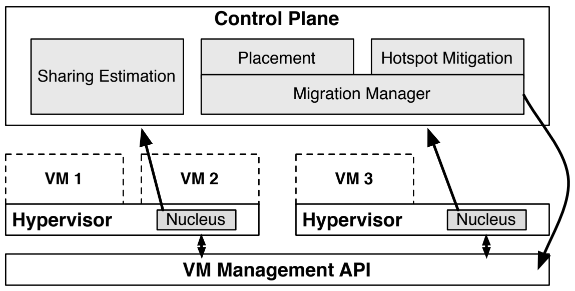


Figure 4.1. Memory Buddies System Architecture. The Nucleus sends memory fingerprint reports to the Control Plane, which computes VM placements and interacts with each hosts to place or migrate VMs according.

Finally, any data center that aggressively exploits page sharing should also implement hotspot mitigation to address any significant *loss* of page sharing due to application termination or major application phase changes—such loss of page sharing can create memory pressure and cause swapping. Hotspot mitigation techniques offload VMs to other servers to reduce memory pressure.

System Overview: Low-level page sharing mechanisms only detect and share duplicate pages belonging to resident VMs—they do not address the problem of *which* VMs to colocate on a host to maximize sharing. *Memory Buddies* detects sharing potential between virtual machines and then uses the low-level sharing mechanisms to realize these benefits.

The Memory Buddies system, which is depicted in Figure 4.1, consists of a *nucleus*, which runs on each server, and a *control plane*, which runs on a distinguished control server. Each nucleus generates a memory fingerprint of all memory pages within the VMs resident on that server. This fingerprint represents the page-level memory contents of a VM in a way which allows efficient calculation of the number of pages with identical content across two VMs. In addition to per-VM fingerprints, we also calculate aggregate per-server fingerprints which represent the union of the VM fingerprints of all VMs hosted on the server, allowing us to calculate the sharing potential of candidate VM migrations.

The control plane is responsible for virtual machine placement and hotspot mitigation. To place a virtual machine it compares the fingerprint of that VM against server fingerprints in order to determine a location for it which will maximize sharing opportunities. It then places the VM on this server if there are sufficient resources available. The control plane interacts with VMs through a VM management API such as VMware's Virtual Infrastructure or the libvirt API [76].

The following sections describe the memory fingerprinting and control plane algorithms in detail.

4.3 Memory Fingerprinting

The nucleus runs on each physical server, computing memory fingerprints for each VM resident on that server, as well as for the server as a whole. Ideally the nucleus would be implemented at the hypervisor level, allowing re-use of many mechanisms already in place to implement content-based page sharing. Our experiments were performed with VMware ESX Server, however, and so lacking source code access[1] we have implemented the fingerprinting aspect of the nucleus within each VM, as a paired guest OS kernel module and user-space daemon.

4.3.1 Fingerprint Generation

Content-based page sharing implementations for both Xen and VMware ESX use hashes of page contents in order to locate pages with identical content which are thus candidates for sharing. In the Memory Buddies nucleus Hsieh's SuperFastHash algorithm [53] is used to generate 32 bit hashes for each 4KB page. In order to measure potential sharing *be-*

[1]Our initial efforts had focused on the open-source Xen platform, where it was possible to make experimental modifications to the hypervisor. However, Xen's page sharing implementation is experimental and not compatible with its live migration mechanism; since our work requires both mechanisms, the work presented in this chapter makes use of VMware ESX Server.

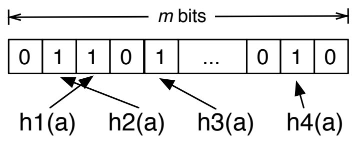


Figure 4.2. Bloom filter with four hash functions, containing a single key a.

tween VMs, rather than *self-sharing*, i.e., sharing within a single VM[2], we gather the set of unique page hashes for a VM's pages to generate the raw memory fingerprint. Maintained in sorted order, such a fingerprint may be easily compared against the fingerprint of another VM or server, yielding a count of the pages duplicated between the two VMs and thus the potential memory sharing between them.

4.3.2 Succinct Fingerprints

The memory fingerprints we have described consist of a list of page hashes; the intersection between two such fingerprints may be computed exactly, but they are unwieldy to use. Not only are they large—e.g. 1 MB of fingerprint for each 1 GB of VM address space—but they must be sorted in order to be compared efficiently. To reduce this overhead, we also provide a *succinct fingerprint* which represents this set of hashes using a Bloom filter. [14, 71]. A Bloom filter is a lossy representation of a set of keys, which may be used to test a value for membership in that set with configurable accuracy. The filter parameters may be set to trade off this accuracy against the amount of memory consumed by the Bloom filter.

As shown in Figure 4.2, a Bloom filter consists of an m-bit vector and a set of k hash functions $H = h_1, h_2, h_3, ..., h_k$ ($k = 4$ in the figure). For each element a, the bits corresponding to positions $H(a) = h_1(a), h_2(a), ..., h_k(a)$ are set to 1; to test for the presence of a, we check to see whether all bits in $H(a)$ are set to 1. If this test fails we can be certain

[2]Our results indicate that a single virtual machine often contains significant numbers of duplicated pages, a fact that we exploit in Chapter 6 to optimize VM migration.

that a is not in the set. However, we observe that the test may succeed—i.e. result in a *false positive*—if all bits in $H(a)$ were set by the hashes of some other combination of variables. The probability of such errors depends on the size of the vector m, the number k of bits set per key, and the probability that any bit in the vector is 1.

If the number of elements stored is n, the probability 'p_e' of an error when testing a single key against the filter is given by $p_e = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$. Thus given n, the number of pages belonging to a VM, proper choice of the size of the bit vector m and the number of hash functions k can yield a sufficiently small error probability. Two Bloom filters may be compared to estimate the size of the intersection of their key sets; this is covered in more detail in Section 4.3.3. In addition, multiple Bloom filters can be combined by taking the logical OR of their bit vectors; Memory Buddies uses this to create aggregate fingerprints for all VMs running on each host.

While Memory Buddies supports both fingerprint mechanisms, we note that in practice neither full hash lists nor succinct fingerprints will produce an absolutely accurate prediction of page sharing behavior, for several reasons. First, the fingerprints are snapshots of time-varying behavior, and lose accuracy as the actual memory contents change after the fingerprint was taken. A second reason is that comparison of snapshots only indicates how many pages *could* be shared; for various reasons the page sharing logic within the hypervisor may fail to share some pages which might otherwise be sharable. Finally, additional sharing occurs due to multiple copies of a single page within a VM; we ignore this self-sharing since the hypervisor will be able to detect and utilize it regardless of where the VM is placed.

4.3.3 Fingerprint Comparison

To estimate page sharing potential, we need to compare the memory fingerprints of two or more virtual machines and compute their intersection: i.e. the number of identical pages between the two. With raw memory fingerprints consisting of the full list of memory page

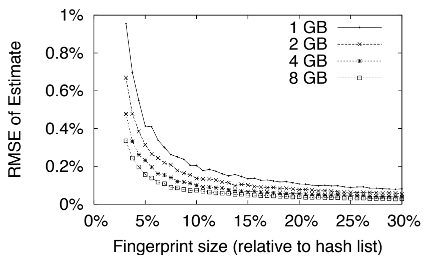


Figure 4.3. Root Mean Squared Error (RMSE) of page sharing estimate for different memory and Bloom filter sizes.

hashes, this may be done by sorting the lists, comparing them, and counting the number of matches. Comparing two concise fingerprints is somewhat more complicated, although faster.

To calculate the size of this intersection, we examine the case of two Bloom filters holding u_1 and u_2 unique entries each, plus c entries common between the two. We then take the bitwise AND of the two filter vectors; the elements of this resulting vector will be 1 for (a) each bit set by the c common elements, and (b) each bit set by one or more keys in each of the unique sets. We omit the mathematical derivation, which can be found in related work [19], but note that the expected number of shared elements is [81]:

$$share = \frac{ln(z_1 + z_2 - z_{12} - ln(z_1 * z_2) + ln(m)}{k(ln(m) - ln(m-1))} \tag{4.1}$$

where z_1 and z_2 are the numbers of zeros in the two Bloom filters, z_{12} is the number of zeros in the AND of the two filters, m is the size of each of the filter vectors, and k is the number of hash functions used.

The estimate contains a correction for the expected number of false matches between the two vectors, and is thus considerably more accurate than the test of a single key against the same Bloom filter. No closed-form solutions for the accuracy of this estimate have been derived to date; however we are able to measure it for particular cases via Monte Carlo simulation. In Figure 4.3 we see error results for different numbers of keys and

Bloom filter sizes; the number of keys is expressed as a total amount of memory (e.g., 1 GB = 256K page hashes or keys) and the filter size is expressed as a fraction of the size needed for the full hash list (e.g., 256K hashes requires 1024KB at 4 bytes per hash). The error rate is the percent of total pages which are incorrectly considered to be shared or unshared by the Bloom filter. We see that with a filter as small as 5% of the size of the hash list—i.e. only slightly more than 1 bit per page—the expected error is less than 0.5%, allowing us to estimate sharing quite precisely with succinct fingerprints. In addition to the savings in communication bandwidth for reporting these succinct fingerprints, they are also much faster to compare, as they are both much smaller and, unlike hash lists, require no sorting before comparison.

4.4 Sharing-aware Colocation

The VM and server fingerprints are periodically computed and transmitted to the control plane by each nucleus; the control plane thus has a system-wide view of the fingerprints of all VMs and servers in the data center. The control plane implements a colocation algorithm that uses this system-wide knowledge to identify servers with the greatest page sharing potential for each VM that needs to be placed.

The control plane provides support for three types of placement decisions: initial placement of new VMs, consolidation strategies for live data centers, and offline planning tools for data center capacity planning.

4.4.1 Initial Placement

When a new virtual machine is added to a data center, an initial host must be selected. Picking a host based simply on current resource utilization levels can be inefficient. The placement algorithm in Memory Buddies instead attempts to deploy VMs to the hosts which will allow for the greatest amount of sharing, reducing total memory consumption, allowing more VMs to be hosted on a given number of servers.

Each new VM is initially placed on a staging host where its resource usage and memory fingerprint can stabilize after startup and be observed. Each VM periodically reports memory fingerprints as well as the resource usages on each server. Monitored resources include memory, CPU, network bandwidth and disk; both the mean usage over the measurement interval as well as the peak observed usage are reported. The placement algorithm uses these reported usages to identify the best candidates for placing each new VM.

The algorithm first determines the set of feasible hosts in the data center. A feasible host is one that has sufficient available resources to house the new VM—recall that each VM is allocated a slice of the CPU, network bandwidth and memory on the host, and only hosts with at least this much spare capacity should be considered as possible targets. Given a set of feasible hosts, the algorithm must estimate the page sharing potential on each host using our fingerprint comparison technique—the fingerprint for the VM is compared with the composite fingerprint of the physical server directly using hash lists, or the number of shared pages is estimated using Equation 4.1 if compact Bloom filters are being used. The algorithm then simply chooses the feasible server that offers the maximum sharing potential as the new host for that VM.

4.4.2 Server Consolidation

Memory Buddies' server consolidation algorithm opportunistically identifies servers that are candidates for shutting down and attempts to migrate virtual machines to hosts with high sharing opportunities. In doing so, it attempts to pack VMs onto servers so as to reduce aggregate memory footprint and maximize the number of VMs that can be housed in the data center. Once the migrations are complete, the consolidation candidates can be retired from service or powered down until new server capacity is needed, thereby saving on operational (energy) costs. The consolidation algorithm comprises three phases:

Phase 1: *Identify servers to consolidate.* The consolidation algorithm runs periodically (e.g., once a day) and can also be invoked manually when needed. A list of hosts which are

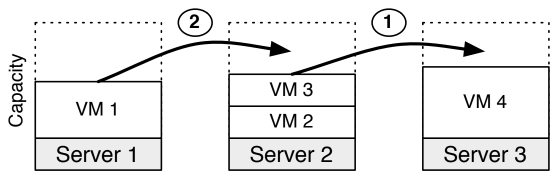


Figure 4.4. A 2-step migration: VM_3 is first migrated from Server 2 to Server 3 to free up space for VM_1. VM_1 can then be migrated from Server 1 to Server 2.

candidates for consolidation is determined by examining memory utilization statistics for each host; a server becomes a candidate for consolidation if its mean usage remains below a low threshold for an extended duration.[3] Currently our system only considers memory usages when identifying consolidation candidates; however, it is easy to extend it to check usages of all resources to identify lightly loaded servers.

Phase 2: *Determine target hosts.* Once the set of consolidation candidates has been identified, the algorithm must determine a new physical server to house each VM. To do so, we order VMs in decreasing order of their memory sizes and consider them for migration one at a time. For each VM, the algorithm first determines the set of feasible servers in the data center as described in Section 4.4.1. The host which will provide the greatest level of sharing (while still ensuring sufficient resources) is then selected for each VM.

In certain cases, it is possible that there are no feasible servers for a VM. This can happen if the VM has a large CPU, network or memory footprint and existing servers in the data center are heavily utilized. In this case, the consolidation algorithm must consider a multi-way move, where one or more VMs from an existing server are moved to other servers to free up additional capacity and make this server feasible for the VM under consideration, as illustrated in Figure 4.4. As migration does impose some overhead, the algorithm attempts to minimize the number of moves considered in multi-way planning.

[3]In addition, the system can also check that the peak usage over this duration stayed below a threshold, to ensure that the server did not experience any load spikes during this period.

Phase 3: *Migrate VMs to targets.* Once new destinations have been determined for each VM on the consolidation servers, our algorithm can perform the actual migrations. Live migration is used to ensure transparency and near-zero down-times for the application executing inside the migrated VMs.

To ensure minimum impact of network copying triggered by each migration on application performance, our algorithm places a limit on the number of concurrent migrations; once each migration completes, a pending one is triggered until all VMs have migrated to their new hosts. The original servers are then powered off and retired or moved to a shutdown pool so they can be reinitialized later if memory requirements increase.

4.4.3 Offline Planning Tool for Smart VM Colocation

The Memory Buddies system can also be used for offline planning to estimate the required data center capacity to host a set of virtual machines. The planning tool can be used to answer "what if" questions about the amount of sharing potential for different VM configurations, or to generate initial VM placements.

The offline planner takes as input:

1. A list of the data center's hosts and their resource capacities.
2. Resource utilization statistics (CPU, network, and disk) for each system to be placed.
3. Memory fingerprints for each system.

The systems to be placed within the data center may either already be running as virtual machines, or may be sets of applications currently running on physical hosts which are to be moved to a virtualized setting (e.g. desktop virtualization). If the systems to be hosted in the data center are not yet running on virtual machines, then additional modeling techniques may be required to estimate the resource requirements of the applications after virtualization overheads are added [143]. In either case, the memory fingerprints are gathered by deploying the memory tracer software (our kernel module implementation of the nucleus) on each system to be moved to the data center.

The planning tool can be used to analyze "what if" scenarios where a data center administrator wants to know about the resource consumption of different sets of VMs hosted together. The tool can output the amount of both inter-VM and self-sharing likely to occur for a given set of VMs. This provides valuable information about the expected amount of memory sharing from colocating different applications or operating systems.

The offline planner can generate VM placements that match each VM to a host such that the resource capacities of the host are not violated, while maximizing the amount of sharing between VMs. This is analogous to a bin packing problem where the resource constraints define the size of each bin. A variety of heuristics can be used for this sort of problem. Memory Buddies uses a dynamic programming technique which determines what subsets of VMs will fit on each host to maximize sharing while respecting constraints. These constraints may be simple resource consumption thresholds such as not using more than 80% of the CPU or requiring a portion of the server's memory to be kept free to prevent changes in sharing or memory requirements causing hotspots. Constraints can also be used to enforce business rules such as only colocating a single customer's VMs on a given host or to ensure fault tolerance by preventing multiple replicas of an application from being placed together. The tool's output provides a list of which VMs to place on what hosts, as well as the total memory consumption and expected rate of sharing.

4.5 Hotspot Mitigation

The Memory Buddies hotspot mitigation technique works in conjunction with the consolidation mechanism to provide a sharing-aware mechanism for resolving memory pressure caused by changes in virtual machine behavior. Our system must detect such hotspots when they form and mitigate their effects by re-balancing the load among the physical hosts. We note that memory hotspots are only one form of such overload; other kinds of hotspots can occur due to demand for CPU, network, and disk resources. Such overloads are best dealt with by other techniques [145] and are not considered in this work.

A memory hotspot may arise for several reasons. First, it may be due to increased demand for memory by one or more virtual machines. Changing behavior on the part of applications or the guest OS (e.g. the file system buffer cache) may result in a need for more memory, which the hypervisor will typically attempt to meet by contracting the memory balloon and returning memory to the guest. The second possible cause is due to a loss of page sharing. If changes in virtual machine behavior cause its memory contents to change (a so-called “phase change”) in such a way as to reduce memory sharing, then overall memory usage on a physical server may increase even though the amount of memory seen by each guest OS remains constant.

The control plane relies on statistics reported by the Memory Buddies nucleus to detect memory hotspots. If implemented at the hypervisor level, the nucleus would have direct access to information on the availability of physical memory; in our prototype we must instead infer this information from guest behavior and externally reported hypervisor statistics. In particular, we monitor both the level of swap activity on each guest OS, as well as the number of shared pages reported by the hypervisor.

When swap activity rises above a certain threshold, a hotspot is flagged by the control plane, which then attempts to resolve it by re-distributing VMs among physical servers. In choosing a VM to move and a destination for that VM, we use the same algorithm as for initial placement. In particular, we examine all VMs on the overloaded system, and for each VM calculate the maximum gain in sharing which could be obtained by migrating that VM to another feasible host. We then choose to migrate the VM which provides the highest absolute gain in sharing—i.e. which provides the maximum system-wide increase in available memory.

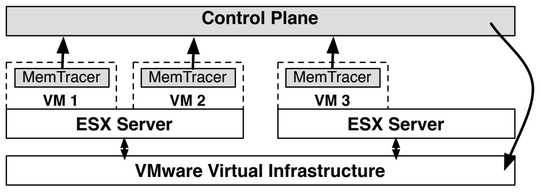


Figure 4.5. Our implementation uses VMware ESX since it supports migration and page sharing. As it is a closed source hypervisor, the nucleus is implemented as a memory tracer component running within each VM.

If there are no feasible destinations for the virtual machines on the overloaded host, a server must be brought in from the shutdown pool so that it can host one or more of the VMs.

4.6 Implementation

The Memory Buddies implementation uses VMware ESX for the virtualization layer as it supports both page sharing and virtual machine migration. While the ESX hypervisor already gathers page hashes to determine sharable pages [133], this information is unavailable to our software because of ESX's closed nature. As a result, our system implementation moves the nucleus component from the hypervisor into each virtual machine in the form of a memory tracing kernel module that supports Linux, Windows, and Mac OS X. This software gathers the lists of hashes and sends them (or compact Bloom filters) to the control plane. Figure 4.5 illustrates the specifics of our implementation. We have deployed our tracer and control plane on a testbed for evaluation.

Memory Tracer: We have developed memory analysis tools that run on Linux, Mac OS X, and Windows. The Linux tracer supports both 2.4 and 2.6 kernels, the Mac tracer runs under OS X 10.5 on Intel and G4 systems, and the Windows tracer supports XP Service Pack 2 and 32-bit Vista systems. All of the tracers work by periodically stepping through the full memory of the machine being traced, generating 32 bit hashes for each page in memory. When used in our testbed, the tracer is run within each virtual machine, and the resulting hash lists (or Bloom filters) are sent to the control plane for processing every few minutes. An alternate version of our tracer has been designed solely for gathering memory traces, and has been distributed to volunteers within our department to be run on a variety of physical machines. We analyze the sharing potential in Section 4.7.2.

Under memory pressure, a VMware ESX host may make use of the *balloon driver* [133] to reclaim memory from running VMs. We note that as the guest OS in unaware of the balloon driver activity, our memory tracer may analyze such reclaimed memory pages. In practice, however, this is not a concern, as reclaimed memory will uniformly appear zeroed, and thus will not affect fingerprints based on unique page hashes.

Control Plane: The control plane is a Java based server which communicates with the VMware Virtual Infrastructure management console via a web services based API. The API is used by the control plane to discover which hosts are currently active and where each virtual machine resides. Extra resource statistics are retrieved from the VMware management node such as the total memory allocation for each VM. This API is also used to initiate virtual machine migrations between hosts. The control plane primarily consists of statistic gathering, sharing estimation, and migration components which comprise about 3600 lines of code.

Memory Buddies Testbed: The testbed is a cluster of P4 2.4GHz servers connected over gigabit ethernet which combines the Control Plane with a set of virtual machines, each running the Memory Tracer. Each server runs VMware ESX 3.0.1 and the VMware Virtual Infrastructure 2.0.1 management system is on an additional node.

4.7 Experimental Evaluation

We have evaluated the Memory Buddies system to study the benefits of exploiting page sharing information when determining virtual machine placement.

Section 4.7.1 discusses our evaluation workloads and experiment specifications. Section 4.7.2 analyzes the sharing characteristics of the memory traces we have gathered. Our first case study measures the benefits of Memory Buddies for Internet Data Centers (section 4.7.3) on both our testbed and through trace driven simulation. Section 4.7.4 evaluates the hotspot mitigation algorithm on our testbed and we explore offline planning with a desktop virtualization case study in section 4.7.5. Finally, Section 4.7.6 shows the performance tradeoffs of the fingerprinting techniques available in Memory Buddies.

4.7.1 Experimental Workloads

We conduct two sets of experiments to evaluate Memory Buddies. First, we demonstrate the performance of our consolidation and hotspot mitigation algorithms on a small prototype Memory Buddies data center running realistic applications. Second, to demonstrate that these results apply to larger data centers and to real-world applications, we gather memory traces from live machines in our department and use these traces to evaluate the efficacy of our techniques.

Our prototype data center experiments are based on the following applications:

- RUBiS [23] is an open source multi-tier web application that implements an eBay-like auction web site and includes a workload generator that emulates users browsing and bidding on items. We use the Apache/PHP implementation of RUBiS version 1.4.3 with a MySQL database.

- TPC-W [122] models an Amazon style e-commerce website implemented with Java servlets and run on the Jigsaw server with a DB2 backend.

- SpecJBB 2005 [111] is a Java based business application benchmark which emulates a 3-tier system with a workload generator.

- Apache Open For Business (OFBiz) [99] is an open source suite of enterprise web applications with accouting, finance, and sales functionality used by many businesses. We utilize the eCommerce component and a workload generator based on the JWebUnit testing framework to emulate client browsing activities.

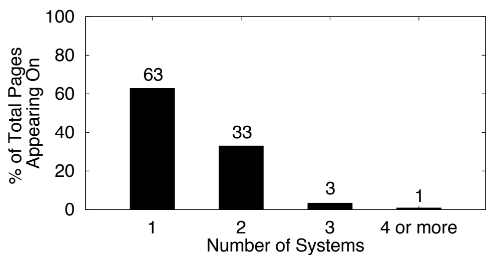


Figure 4.6. Percentage of memory pages duplication between VMs on a collection of 30 diverse laptops, desktops, and servers. 33% of pages were sharable with exactly one other machine, and 37% with one or more machines.

To ensure that inter-VM sharing is predominantly due to code pages, we randomize the data used by different instances of same application— the workloads and database contents are different for each VM instance to avoid sharing of data pages. For the multi-tier applications, we run all tiers within a single virtual machine. All Apache web servers are version 2.2.3 with PHP 4.4.4-9, MySQL databases are 5.0.41, Jigsaw is version 2.2.6, and the DB2 server was DB2 Express-C 9.1.2.

We extend our evaluation with a study of memory traces collected by our tracer tool. We have distributed the memory tracer application to volunteers within our department and gathered a total of over 130,000 memory fingerprints from more than 30 systems. [4] We use these traces both to analyze the sharing potential between actual systems and to allow emulation of larger data centers. Finally, we use these memory traces to analyze the accuracy and overhead of our different fingerprint comparison techniques.

4.7.2 Memory Trace Analysis

We have analyzed a subset of the traces gathered from machines within our department to provide a summary of the level of page sharing available across a diverse set of machines. Here we examine the fingerprints gathered on June 10th 2008 from 24 Linux and

[4]We have released a portion of these traces and the code to process them to researchers worldwide on the UMass Trace Repository website.

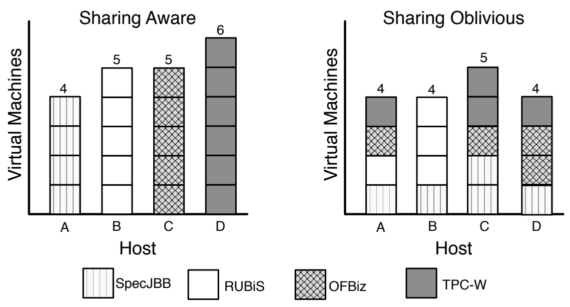


Figure 4.7. Sharing aware vs sharing oblivious placement. Sharing aware detects similar virtual machines and groups them on the same hosts.

Application	Measured Sharing	Pred. Sharing
TPC-W	38%	41%
OpenForBiz	18%	22%
RUBiS	16%	15%
SpecJBB	5%	5%

Table 4.1. Application types and their memory sharing levels. Measured sharing is obtained from the live statistics of the hypervisor, while predicted sharing is computed from the memory traces.

6 Mac OS X systems (our collection of Windows traces is currently too small to provide significant results).

Figure 4.6 shows the number of pages which appear on only one, two, or more systems. This indicates that, as expected, the majority of pages are unique, only appearing on a single host. However, a significant portion of the pages reside on two or more machines. This suggests that in an ideal case where all systems could be colocated onto a single host, the total memory requirements of running these machines could be reduced by about 37%, giving an upper bound on the amount of sharing which could ever occur. We also note that while many pages appear on two systems (33%), very few reside on three or more machines. This emphasizes that random placement of virtual machines is unlikely to realize the full benefits of memory sharing.

4.7.3 Case Study: Internet Data Center

Many web hosting services rent virtual machine servers to customers since they can be much cheaper than providing dedicated servers. These virtual servers are an excellent candidate for exploiting page sharing since the base servers often run similar operating systems and software, such as a LAMP stack or a J2EE environment. In this case study we first test Memory Buddies' ability to more effectively place different classes of applications typically found in an Internet data center. We utilize four different applications to vary the sharing rate between virtual machines, and a testbed with four hosts. Note that while the core application data is identical within an application class, the workloads and database contents are different for each VM instance. Table 4.1 lists the different application types and the level of sharing between pairs of virtual machines of the same type; actual sharing values vary within a few percent depending on paging activities. We present both the predicted sharing reported by our memory tracer and the actual level of sharing achieved by the hypervisor. For the first two applications, the predicted level of sharing is too high; this error occurs when the hypervisor does not choose to share some identical pages, typically because it expects them to change too quickly. For RUBiS, the tracer under predicts slightly, probably because our memory tracer is unable to access all memory regions. SpecJBB obtains the smallest amount of sharing because it is the most memory intensive application, quickly filling the VM's memory with randomly generated data as the benchmark runs.

We compare two placement algorithms: our *sharing aware* approach attempts to place each virtual machine on the host that will maximize its page sharing and the *sharing oblivious* scheme does not consider sharing opportunities when placing virtual machines, and instead places each virtual machine on the first host it finds with sufficient spare capacity. Although the sharing oblivious approach does not explicitly utilize sharing information to guide placement, page sharing will still occur if it happens to place virtual machines together with common pages. In addition, this means that self-sharing within each VM occurs in both scenarios, so the improvements we see are caused by intelligent colocation leading to better inter-vm sharing. For simplicity, we assume that memory is the bottleneck resource and do not consider CPU or network bandwidth as a constraint.

Initially, we create one virtual machine of each type and place it on its own physical host. Additional VMs of each type are then spawned on a fifth host and migrated to one of the four primary hosts. We compare the number of virtual machines which can be successfully hosted using both our sharing aware algorithm which migrates each new VM to the host with the greatest sharing potential and a sharing oblivious placement algorithm which migrates each VM to the first host it finds with sufficient memory, without regard to sharing. The experiment terminates when no new virtual machines can be placed.

Each virtual machine is configured with 384 MB of RAM, and the hosts have 1.5 GB of spare memory since VMware reserves 0.5 GB for itself. Thus we expect each host to be able to run about four VMs without sharing. Figure 4.7 displays the final placements reached by each algorithm. The three web applications, TPC-W, OFBiz, and RUBiS, demonstrate a benefit from utilizing sharing, allowing more VMs to be packed than the base four. The sharing oblivious algorithm places four VMs on each host, except for host C on which it fits an extra VM due to the sharing between TPC-W instances. The sharing aware approach is able to place a total of 20 virtual machines, while the Oblivious approach can only fit 17. For this scenario, exploiting sharing increased the data center's capacity by a modest 17%.

We next use the memory traces gathered from these applications to simulate a larger data center. We increase the total number of hosts to 100 and measure the number of virtual machines which can be placed depending on whether sharing information is used. Our trace driven simulator utilizes the same control plane and algorithms as described previously. On this larger scale testbed, the sharing-aware approach places a total of 469 VMs, while the sharing oblivious approach can only host 406, giving a benefit of about 16% when using sharing. This matches well with the results from our testbed; the slight

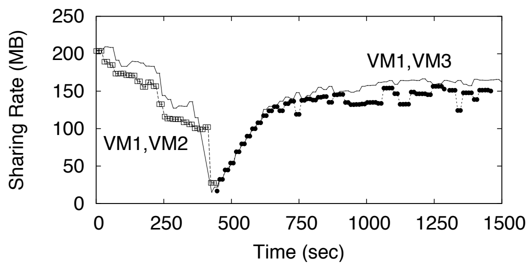


Figure 4.8. Hotspot mitigation: When a change in workload occurs, Memory Buddies migrates VMs to consolidate VMs with higher sharing potential on the same hosts.

change in performance is due to the sharing oblivious approach getting "lucky" and placing more VMs together which happen to share pages.

Result: By reducing the total memory requirements on each host, the effective capacity of a data center can be increased. Our testbed and trace driven simulations obtain benefits of 16-17% due to increased inter-vm sharing when Memory Buddies guides VM placement.

4.7.4 Hotspot Mitigation

Workload variations occur over time for most web applications and this can reduce the potential for memory sharing between colocated VMs in data centers. We have reproduced a typical data center scenario to demonstrate Memory Buddies' ability to detect and respond to a memory hotspot when application phase changes. The experiment employs two hosts, the first running two virtual machines (VM_1 and VM_2) and the second running only one (VM_3). All of the virtual machines are allocated 512MB of memory and serve static files generated following the SPECweb99 specification [90] with Apache web servers. Initially, we use httperf [88] to send an identical set of requests to each server resulting in a high potential for sharing between VMs.

Figure 4.8 shows the amount of memory shared by each VM with the other VMs residing on the same host as reported by VMWare ESX. Since VM_1 and VM_2 are colocated, they initially have a high level of sharing at about 400MB. After 60 seconds of load injection, we trigger a phase change for the requests being sent to VM_2. As a result, the sharing between VM_1 and VM_2 decreases significantly putting more memory pressure on the host. This triggers Memory Buddies hotspot mitigation mechanism at time 360 seconds. Since VM_1 and VM_3 continue to receive the same workload, there is a high potential for sharing between them. Therefore Memory Buddies determines that VM_1 should be migrated to Host 2. After the migration completes, the sharing rate between VM_1 and VM_3 gradually increases again as ESX Server CBPS identifies sharable pages.

Result: Memory Buddies' monitoring system is able to detect changes in sharing potential brought on by application phase transitions. This type of hotspot is automatically resolved by determining a different host with a higher sharing potential for one of the VMs.

4.7.5 Case Study: Desktop Virtualization

Desktop virtualization consists of moving traditional desktop environments to virtual machines colocated in a data center. The user can then access his desktop environment using a thin-client interface from various locations. System administrators that are planning to deploy desktop virtualization need to estimate the memory requirements to host all the desktop VMs on the servers. The potential effects of intra and inter-VM memory sharing are not known a priori which makes it very hard to plan adequate memory resources. In this case study, we show how we can use our tools to answer "what if" questions in order to predict the memory requirements of colocated desktop virtual machines.

We have deployed our memory tracer on a set of real workstations running Windows, Linux and MacOS X on PowerPC and Intel platforms. We have collected memory traces from each machine every 30 minutes over several weeks. This data has been consolidated in a database to allow for easier mining. Table 4.2 summarizes the various desktop configurations we have considered.

By combining the traces of the different machines in the database, we can quickly compute how much memory sharing can be achieved for a particular combination of VMs. The number of unique hashes found in the combined traces represent the number of physical

OS	CPU	RAM
Darwin 9.0.0	PowerBook 6, PowerPC	1152
Darwin 9.2.0	Macmini1, i386	1024
Darwin 9.4.0	MacBook2, i386	2048
Darwin 9.4.0	iMac7, i386	2048
Linux 2.6.9	Intel Family 15 Model 2	1010
Linux 2.6.18	Intel Family 6 Model 2	2018
Windows NT 5.1	x86 Family 6 Model 15	511

Table 4.2. Machine configurations (as reported by their operating system) considered in the desktop virtualization case study.

memory pages that will be needed by the hypervisor in case of perfect sharing. This upper-bound of sharing includes both inter and intra-VM sharing. We use the collected data and our tools to answer a number of "what if" questions with different combinations of OS colocation. Table 4.3 shows the results we obtained for 4 questions: what is the potential sharing (i) if 3 VMs have a different OS, (ii) if 2 VMs have the same OS but different versions and 1 VM has a different OS, (iii) if 3 VMs have the same OS but different versions and 1 VM has a different OS, (iv) all VMs have the same OS version but possibly different hardware platforms.

OS combination	Total memory	Shareable pages	Predicted server memory
Linux 2.6.9, Darwin 9.0.0, Win NT 5.1	4223 MB	13.2%	3666 MB
Darwin 9.4.0, Darwin 9.0.0, Win NT 5.1	5248 MB	35.3%	3397 MB
Darwin 9.*, Win NT 5.1	6272 MB	36.8%	3966 MB
Darwin 9.4.0 (3 MacBook2 + 1 iMac7)	8192 MB	40.0%	4917 MB

Table 4.3. Server memory usage prediction for various colocation configuration. Total memory represents the total memory required without sharing and predicted memory is the required memory on the server when all possible sharable pages are actually shared.

When heterogeneous OSes are combined (first line of the table), the sharing potential only comes from intra-VM sharing and remains at a modest 13%. When we replace Linux by another version of MacOS X, inter-VM sharing starts to play a significant role and memory sharing jumps to 35% overall. Adding different versions of the same operating system

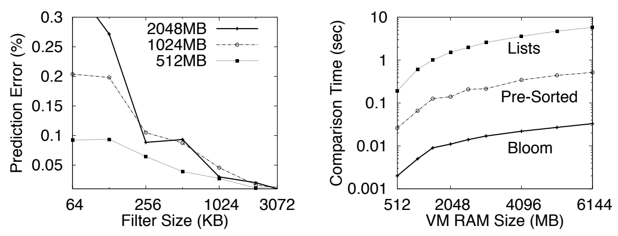
VM RAM	Hash List Size (KB)	Bloom Size (KB) w/0.2% Error
1GB	1024	92
4GB	4096	124
8GB	8192	368

Table 4.4. Per VM communication cost in KB for hash lists and Bloom filters with a 0.2% error rate.

(Darwin 9.0, 9.2 and 9.4) maintains a substantial inter-VM sharing for an overall memory sharing close to 37%. When homogeneous software configurations are used even on different hardware platforms, we observe memory sharing up to 40%. These numbers represent the optimal memory sharing case and actual sharing might be lower depending on the hypervisor implementation of page sharing. Note that the predicted server memory does not account for the hypervisor memory requirements that are usually fixed and implementation dependent.

If the machine workload varies greatly over time, it is possible to perform these computations with different traces taken at different points in time to evaluate the memory sharing evolution over time. We found in our experiments that the predicted memory sharing did not change significantly over time for desktop machines. Computing the memory server prediction for a given configuration usually only takes few seconds but this may vary depending on the database size and number of traces to analyze. It is then possible to use the technique to answer a broad range of “what if” questions like sharing potential over time or in the presence of workload variations.

Result: Memory Buddies can be used offline to compute memory sharing and answer “what if” questions when planning for desktop virtualization. We found that colocating different OSes only uses intra-VM sharing. However mixing different versions of the same OS leads to substantial inter-VM sharing. As expected, the maximum sharing is observed when similar versions of an OS are colocated.



(a) Bloom Filter Estimation Error (b) Comparison Efficiency

Figure 4.9. Bloom filter accuracy vs efficiency tradeoff. Smaller Bloom filter bit vectors reduce the accuracy of sharing estimates, but also significantly reduce the computation time required for comparison.

4.7.6 Fingerprint Efficiency and Accuracy

Memory Buddies allows a tradeoff between the accuracy, speed and space required for estimating sharing potential depending on whether hash lists or Bloom filters are used.

We first measure the accuracy of Bloom filter comparisons when varying the size of the Bloom filter's bit vector. We use pairs of traces gathered in our department study from systems with 512MB, 1GB, and 2GB of RAM. We report the average error in percent of total pages for four pairs of traces of each size. Figure 4.9(a) illustrates how the comparison error rapidly decreases as filter size rises, although larger memory sizes require bigger filters to prevent hash collisions. These results confirm the simulation data shown previously in Figure 4.3; a Bloom filter of only a few hundred KB is sufficient for an error rate of about 0.1%.

We next measure the time to compare two fingerprints to calculate the potential for sharing when using our exact and compact techniques. In both approaches, the computation time increases when using VMs with larger amounts of RAM, because either there are more hashes to be compared or a Bloom filter with a larger bit vector is required in order to meet a target accuracy level. Figure 4.9(b) demonstrates how the comparison time for a pair of VMs increases with memory size. The exact comparison technique using hash lists first sorts the two lists before comparing them. Since sorting can be the dominant cost, we

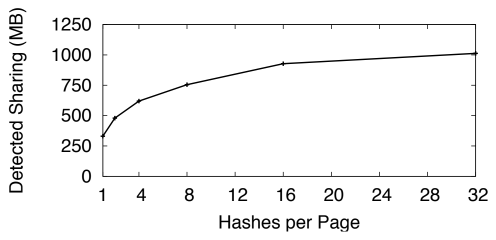


Figure 4.10. Splitting each page into multiple chunks allows Memory Buddies to detect sub-page level sharing between similar pages.

also present the time when lists are presorted by each VM prior to sending the hash lists to the control plane. Presorting the lists decreases the comparison time by about an order of magnitude, but incurs overhead on each host in the system. Switching to Bloom filters reduces the time further, but at the expense of reduced accuracy.

The total communication overhead of the system is dependent on the number of VMs running in the data center, the amount of RAM used by each VM, and the fingerprinting method used. Table 4.4 compares the cost of storing or transmitting Bloom filter based memory fingerprints or hash lists of various sizes. Fingerprints only need to be gathered once every few minutes, incurring minimal network cost if there is a small number of VMs. For very large data centers, the overhead of transmitting full hash lists can become prohibitive, while the Bloom filter approach remains manageable.

Result: Employing Bloom filters in large data centers can reduce sharing estimation time by an order of magnitude and can reduce network overheads by over 90%, while still maintaining a high degree of accuracy.

4.7.7 Sub-Page Sharing

While VMware ESX currently only supports memory sharing at the granularity of full pages, recent research has demonstrated that significant benefits can be obtained by sharing portions of similar, but not identical pages [47]. We have added preliminary support to Memory Buddies for detecting sub-page level sharing between systems by breaking each page into a series of n chunks, each of which is mapped to a 32bit hash. As a result, Memory Buddies produces a fingerprint n times as large for each system, but it can use its existing fingerprint comparison tools to detect similarity between different VMs.

To demonstrate the benefits of sub-page sharing, we have analyzed the amount of sharing achieved between two systems running 64bit Ubuntu Linux, each with 2GB of RAM, when the number of hashes per page is varied between one and thirty two. Figure 4.10 illustrates how subpage level sharing can triple the total amount of sharable memory. The number of hashes per page could be selected by the system operator to balance the added overhead of larger fingerprints against the increased accuracy in sub-page level sharing estimation.

Result: Although Memory Buddies does not currently use a hypervisor that supports sub-page level sharing, it can efficiently detect similar pages by generating multiple hashes per page. This can provide significant benefits in total sharing.

4.8 Related Work

Transparent page sharing in a virtual machine hypervisor was implemented in the Disco system [21]; however it required guest operating system modification, and detected identical pages based on factors such as origin from the same location on disk. Content-based page sharing was introduced in VMware ESX [133], and later in Xen [69]. These implementations use background hashing and page comparison in the hypervisor to transparently identify identical pages, regardless of their origin. Since our prototype lacks access to the memory hashes gathered by the hypervisor, we duplicate this functionality in the guest OS. In Memory Buddies, however, we extend the use of these page content hashes in order to detect the potential for memory sharing between distinct physical hosts, rather than within a single host.

The Difference Engine system was recently proposed as a means to enable even higher degrees of page sharing by allowing portions of similar pages to be shared [47]. Although

Memory Buddies has preliminary support for detecting sub-page sharing across machines by using multiple hashes per page, it currently relies on ESX's sharing functions which do not support sub-page level sharing. We believe that as the technologies to share memory become more effective and efficient, the benefits of using page sharing to guide placement will continue to rise.

Process migration was first investigated in the 80's [98, 121]. The re-emergence of virtualization led to techniques for virtual machine migration performed over long time scales in [107, 137, 70]. The means for "live" migration of virtual machines incurring downtimes of only tens of milliseconds have been implemented in both Xen [30] and VMware [91]. At the time of writing, however, only VMware ESX server supports both live migration and page sharing simultaneously.

Virtual machine migration was used for dynamic resource allocation over large time scales in [105, 117, 43]. Previous work [145] and the VMware Distributed Resource Scheduler [138] monitor CPU, network, and memory utilization in clusters of virtual machines and use migration for load balancing. The Memory Buddies system is designed to work in conjunction with these multi-resource load balancing systems by providing a means to use page sharing to help guide placement decisions. Moreover, offline planning of memory resources for desktop virtualization can be predicted accurately rather than relying on generic rules of thumb that are recommended by manufacturers.

Bloom filters were first proposed in [14] to provide a tradeoff between space and accuracy when storing hash coded information. Guo et al. provide a good overview of Bloom filters as well as an introduction to intersection techniques [44]. Bloom filters have also been used to rapidly compare search document sets in [59] by comparing the inner product of pairs of Bloom filters. The Bloom filter intersection technique we use provides a more accurate estimate based on the Bloom filter properties related to the probability of individual bits being set in the bit vector. This approach was used in [81] to detect similar workloads in peer to peer networks.

4.9 Memory Buddies Conclusions

Data center operators wish to deploy applications onto servers as efficiently as possible, but relying on system administrator knowledge is impractical due to the large scale of modern data centers. One of the key constraints in determining which applications can be placed on which servers is the memory limitations of each machine. In this chapter, we have presented Memory Buddies, a system that improves data center efficiency by automatically consolidating VMs based on their potential to share identical regions of memory. This reduces the memory footprint of each VM and allows more applications to be packed onto each server.

We have made three contributions: (i) a fingerprinting technique—based on hash lists or Bloom filters—to capture VM memory content and identify high page sharing potential, (ii) a smart VM colocation algorithm that can be used for both initial placement of virtual machines or to consolidate live environments and adapt to load variations using a hotspot mitigation algorithm, and (iii) a collection of memory traces of real-world systems that we are making available to other researchers to validate and explore further memory sharing experiments.

We have shown that Memory Buddies can increase the effective capacity of a data center by intelligently grouping VMs with similar memory contents, enabling a high degree of inter-vm page sharing. We have also demonstrated that Memory Buddies can effectively detect and resolve memory hotspots due to changes in sharing patterns. Thus the Memory Buddies system assists with both the initial deployment of applications into a virtualized data center and with live resource management, an area discussed further in the next chapter.

CHAPTER 5

DYNAMIC VIRTUAL MACHINE PROVISIONING

Once applications have been deployed to virtual machines within a data center, they still need to be carefully managed to ensure a high level of performance. The highly dynamic workload fluctuations seen by many data center applications means that the resource needs of an application can vary significantly over time. In this chapter we propose the use of dynamic VM resizing and live migration to balance load and prevent hotspots in data centers.

5.1 Background and Motivation

One of the key benefits of virtualization is the ability to flexibly and dynamically allocate resources to virtual machines. This is especially useful for data centers running Internet applications where customer websites may see highly variable and unpredictable load. Provisioning virtual machines for the maximum expected load can be wasteful if average load is low, but it can also be insufficient since "flash crowds" may cause huge unexpected traffic spikes. In order to maximize efficiency and still maintain high levels of application performance, an automated solution is needed to balance resources among many systems with differing workloads.

Two techniques for dynamically provisioning virtual machines are dynamic resizing and live migration. The abstraction layer provided by the virtualization platform makes it easy to dynamically adjust the amount of physical resources dedicated to each virtual machine. In addition, since VMs are not tied directly to physical resources, they can be migrated between physical servers in order to balance load across the data center.

In this chapter we present Sandpiper, a system which uses both VM resizing and migration to efficiently handle the dynamic workloads seen by data center applications. Sandpiper automates the procedure of detecting when virtual machines are becoming overloaded, calculating how many resources need to be assigned in order to meet application demands, and actualizing those resource requirements through a combination of live migrations and dynamic resource allocations.

5.2 System Overview

Historically, approaches to dynamic provisioning have either focused on dynamic *replication*, where the number of servers allocated to an application is varied, or dynamic *slicing*, where the fraction of a server allocated to an application is varied. With the re-emergence of server virtualization, application *migration* has become an option for dynamic provisioning. Since migration is transparent to applications executing within virtual machines, our work considers this third approach—resource provisioning via dynamic migrations in virtualized data centers. We present *Sandpiper*[1], a system for automated resource allocation and migration of virtual servers in a data center to meet application SLAs. Sandpiper assumes a large cluster of possibly heterogeneous servers. The hardware configuration of each server—its CPU, network interface, disk and memory characteristics—is assumed to be known to Sandpiper. Each physical server (also referred to as a physical machine or PM) runs a *virtual machine monitor* and one or more virtual machines. Each virtual server runs an application or an application component (the terms virtual servers and virtual machine are used interchangeably). Sandpiper currently uses Xen to implement such an architecture. Each virtual server is assumed to be allocated a certain slice of the physical server resources. In the case of CPU, this is achieved by assigning a subset of the host's CPUs to each virtual machine, along with a weight that the underlying Xen CPU

[1]A migratory bird.

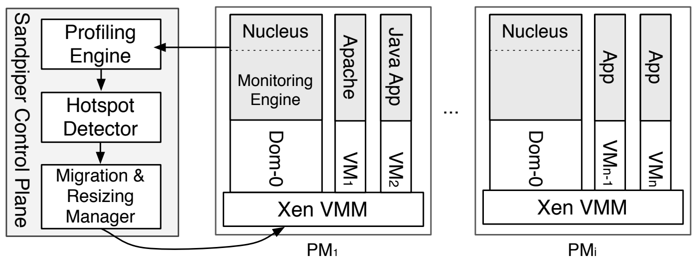


Figure 5.1. The Sandpiper architecture

scheduler uses to allocate CPU bandwidth. In case of the network interface, Xen is yet to implement a similar fair-share scheduler; a best-effort FIFO scheduler is currently used and Sandpiper is designed to work with this constraint. In case of memory, a slice is assigned by allocating a certain amount of RAM to each resident VM. All storage is assumed to be on a network file system or a storage area network, thereby eliminating the need to move disk state during VM migrations [30].

Sandpiper runs a component called the *nucleus* on each physical server; the nucleus runs inside a special virtual server (domain 0 in Xen) and is responsible for gathering resource usage statistics on that server (see Figure 5.1). It employs a *monitoring engine* that gathers processor, network interface and memory swap statistics for each virtual server. For gray-box approaches, it implements a daemon within each virtual server to gather OS-level statistics and perhaps application logs.

The nuclei periodically relay these statistics to the Sandpiper *control plane*. The control plane runs on a distinguished node and implements much of the intelligence in Sandpiper. It comprises three components: a *profiling engine*, a *hotspot detector* and a *migration & resizing manager* (see Figure 5.1). The profiling engine uses the statistics from the nuclei to construct resource usage profiles for each virtual server and aggregate profiles for each physical server. The hotspot detector continuously monitors these usage profiles to detect hotspots —informally, a hotspot is said to have occurred if the aggregate usage of any resource (processor, network or memory) exceeds a threshold or if SLA violations occur for a "sustained" period. Thus, the hotspot detection component determines *when* to signal the need for resource adjustments and invokes the resource manager upon hotspot detection, which attempts hotspot mitigation via resizing or dynamic migrations. It implements algorithms that determine *how much* of a resource to allocate the virtual servers (i.e., determine a new resource allocation to meet the target SLAs), *what* virtual servers to migrate from the overloaded servers, and *where* to move them. The resource manager assumes that the virtual machine monitor implements a migration mechanism that is transparent to applications and uses this mechanism to automate migration decisions; Sandpiper currently uses Xen's migration mechanisms that were presented in [30].

5.3 Monitoring and Profiling in Sandpiper

Sandpiper supports both black- and gray-box monitoring techniques that are combined with profile generation tools to detect hotspots and predict VM resource requirements.

5.3.1 Unobtrusive Black-box Monitoring

The monitoring engine is responsible for tracking the processor, network and memory usage of each virtual server. It also tracks the total resource usage on each physical server by aggregating the usages of resident VMs. The monitoring engine tracks the usage of each resource over a measurement interval $\mathcal{I}$ and reports these statistics to the control plane at the end of each interval.

In a pure black-box approach, all usages must be inferred solely from external observations and without relying on OS-level support inside the VM. Fortunately, much of the required information can be determined directly from the Xen hypervisor or by monitoring events within domain-0 of Xen. Domain-0 is a distinguished VM in Xen that is responsible for I/O processing; domain-0 can host device drivers and act as a "driver" domain that processes I/O requests from other domains [11, 45]. As a result, it is possible to track network and disk I/O activity of various VMs by observing the driver activity in domain-

0 [45]. Similarly, since CPU scheduling is implemented in the Xen hypervisor, the CPU usage of various VMs can be determined by tracking scheduling events in the hypervisor [46]. Thus, black-box monitoring can be implemented in the nucleus by tracking various domain-0 events and without modifying any virtual server. Next, we discuss CPU, network and memory monitoring using this approach.

CPU Monitoring: By instrumenting the Xen hypervisor, it is possible to provide domain-0 with access to CPU scheduling events which indicate when a VM is scheduled and when it relinquishes the CPU. These events are tracked to determine the duration for which each virtual machine is scheduled within each measurement interval $\mathcal{I}$. The Xen 3 distribution includes a monitoring application called *XenMon* [46] that tracks the CPU usages of the resident virtual machines using this approach; for simplicity, the monitoring engine employs a modified version of XenMon to gather CPU usages of resident VMs over a configurable measurement interval $\mathcal{I}$. On a multi-cpu system, a VM may only be granted access to a subset of the total CPUs. However, the number of CPUs allocated to a virtual machine can be adjusted dynamically.

It is important to realize that these statistics do not capture the CPU overhead incurred for processing disk and network I/O requests; since Xen uses domain-0 to process disk and network I/O requests on behalf of other virtual machines, this processing overhead gets charged to the CPU utilization of domain 0. To properly account for this request processing ovehead, analogous to proper accounting of interrupt processing overhead in OS kernels, we must apportion the CPU utilization of domain-0 to other virtual machines. We assume that the monitoring engine and the nucleus impose negligible overhead and that all of the CPU usage of domain-0 is primarily due to requests processed on behalf of other VMs. Since domain-0 can also track I/O request events based on the number of memory page exchanges between domains, we determine the number of disk and network I/O requests that are processed for each VM. Each VM is then charged a fraction of domain-0's usage based on the proportion of the total I/O requests made by that VM. A more precise approach requiring a modified scheduler was proposed in [45].

Network Monitoring: Domain-0 in Xen implements the network interface driver and all other domains access the driver via clean device abstractions. Xen uses a virtual firewall-router (VFR) interface; each domain attaches one or more virtual interfaces to the VFR [11]. Doing so enables Xen to multiplex all its virtual interfaces onto the underlying physical network interfaces.

Consequently, the monitoring engine can conveniently monitor each VM's network usage in Domain-0. Since each virtual interface looks like a modern NIC and Xen uses Linux drivers, the monitoring engines can use the Linux `/proc` interface (in particular `/proc/net/dev`) to monitor the number of bytes sent and received on each interface. These statistics are gathered over interval $\mathcal{I}$ and returned to the control plane.

Memory Monitoring: Black-box monitoring of memory is challenging since Xen allocates a user-specified amount of memory to each VM and requires the OS within the VM to manage that memory; as a result, the memory utilization is only known to the OS within each VM. It is possible to instrument Xen to observe memory accesses within each VM through the use of shadow page tables, which is used by Xen's migration mechanism to determine which pages are dirtied during migration. However, trapping each memory access results in a significant application slowdown and is only enabled during migrations[30]. Thus, memory usage statistics are not directly available and must be inferred.

The only behavior that is visible externally is *swap activity*. Since swap partitions reside on a network disk, I/O requests to swap partitions need to be processed by domain-0 and can be tracked. By tracking the reads and writes to each swap partition from domain-0, it is possible to detect memory pressure within each VM.

Our monitoring engine tracks the number of read and write requests to swap partitions within each measurement interval $\mathcal{I}$ and reports it to the control plane. Since substantial swapping activity is indicative of memory pressure, our current black-box approach is limited to reactive decision making and can not be proactive.

5.3.2 Gray-box Monitoring

Black-box monitoring is useful in scenarios where it is not feasible to “peek inside” a VM to gather usage statistics. Hosting environments, for instance, run third-party applications, and in some cases, third-party installed OS distributions. Amazon’s Elastic Computing Cloud (EC2) service, for instance, provides a “barebone” virtual server where customers can load their own OS images. While OS instrumentation is not feasible in such environments, there are environments such as corporate data centers where both the hardware infrastructure and the applications are owned by the same entity. In such scenarios, it is feasible to gather OS-level statistics as well as application logs, which can potentially enhance the quality of decision making in Sandpiper.

Sandpiper supports gray-box monitoring, when feasible, using a light-weight monitoring daemon that is installed inside each virtual server. In Linux, the monitoring daemon uses the `/proc` interface to gather OS-level statistics of CPU, network, and memory usage. The memory usage monitoring, in particular, enables proactive detection and mitigation of memory hotspots. The monitoring daemon also can process logs of applications such as web and database servers to derive statistics such as request rate, request drops and service times. Direct monitoring of such application-level statistics enables explicit detection of SLA violations, in contrast to the black-box approach that uses resource utilizations as a proxy metric for SLA monitoring.

5.3.3 Profile Generation

The profiling engine receives periodic reports of resource usage from each nucleus. It maintains a usage history for each server, which is then used to compute a profile for each virtual and physical server. A profile is a compact description of that server’s resouce usage over a sliding time window W. Three black-box profiles are maintained per virtual

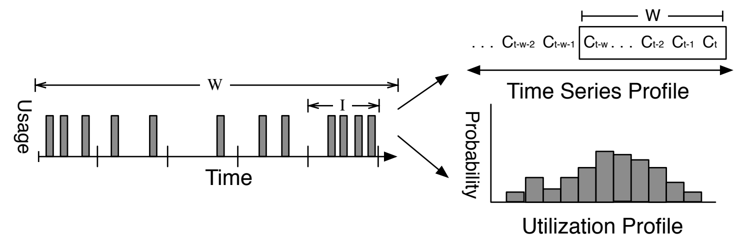


Figure 5.2. Profile generation in Sandpiper

server: CPU utilization, network bandwidth utilization, and swap rate (i.e., page fault rate). If gray-box monitoring is permitted, four additional profiles are maintained: memory utilization, service time, request drop rate and incoming request rate. Similar profiles are also maintained for each physical server, which indicate the aggregate usage of resident VMs.

Each profile contains a distribution and a time series. The distribution, also referred to as the distribution profile, represents the probability distribution of the resource usage over the window W. To compute a CPU distribution profile, for instance, a histogram of observed usages over all intervals $\mathcal{I}$ contained within the window W is computed; normalizing this histogram yields the desired probability distribution (see Figure 5.2).

While a distribution profile captures the variations in the resource usage, it does not capture temporal correlations. For instance, a distribution does not indicate whether the resource utilization increased or decreased within the window W. A time-series profile captures these temporal fluctuations and is simply a list of all reported observations within the window W. For instance, the CPU time-series profile is a list $(C_1, C_2, ..., C_k)$ of the k reported utilizations within the window W. Whereas time-series profiles are used by the hotspot detector to spot increasing utilization trends, distribution profiles are used by the migration manager to estimate peak resource requirements and provision accordingly.

5.4 Hotspot Detection

The hotspot detection algorithm is responsible for signaling a need for VM resizing whenever SLA violations are detected implicitly by the black-box approach or explicitly by the gray-box approach. Hotspot detection is performed on a per-physical server basis in the black-box approach—a hotspot is flagged if the aggregate CPU or network utilizations on the physical server exceed a threshold or if the total swap activity exceeds a threshold. In contrast, explicit SLA violations must be detected on a per-virtual server basis in the gray-box approach—a hotspot is flagged if the memory utilization of the VM exceeds a threshold or if the response time or the request drop rate exceed the SLA-specified values.

To ensure that a small transient spike does not trigger needless migrations, a hotspot is flagged only if thresholds or SLAs are exceeded for a sustained time. Given a time-series profile, a hotspot is flagged if at least k out the n most recent observations as well as the next predicted value exceed a threshold. With this constraint, we can filter out transient spikes and avoid needless migrations. The values of k and n can be chosen to make hotspot detection aggressive or conservative. For a given n, small values of k cause aggressive hotspot detection, while large values of k imply a need for more sustained threshold violations and thus a more conservative approach. Similarly, larger values of n incorporate a longer history, resulting in a more conservative approach. In the extreme, $n = k = 1$ is the most aggressive approach that flags a hostpot as soon as the threshold is exceeded. Finally, the threshold itself also determines how aggressively hotspots are flagged; lower thresholds imply more aggressive migrations at the expense of lower server utilizations, while higher thresholds imply higher utilizations with the risk of potentially higher SLA violations.

Sandpiper employs time-series prediction techniques to predict future values [16]. Specifically, Sandpiper relies on the auto-regressive family of predictors, where the n-th order predictor $AR(n)$ uses n prior observations in conjunction with other statistics of the time series to make a prediction. To illustrate the first-order $AR(1)$ predictor, consider a sequence of observations: u_1, u_2, ..., u_k. Given this time series, we wish to predict the demand in the $(k+1)$th interval. Then the first-order $AR(1)$ predictor makes a prediction using the previous value u_k, the mean of the the time series values μ, and the parameter ϕ which captures the variations in the time series [16]. The prediction $\hat{u}_{k+1}$ is given by:

$$\hat{u}_{k+1} = \mu + \phi(u_k - \mu) \tag{5.1}$$

As new observations arrive from the nuclei, the hot spot detector updates its predictions and performs the above checks to flag new hotspots in the system.

5.5 Resource Provisioning

A hotspot indicates a resource deficit on the underlying physical server to service the collective workloads of resident VMs. Before the hotspot can be resolved, Sandpiper must first estimate *how much* additional resources are needed by the overloaded VMs to fulfill their SLAs; these estimates are then used to determine if local resource allocation adjustments or migrations are required to resolve the hotspot.

5.5.1 Black-box Provisioning

The provisioning component needs to estimate the *peak* CPU, network and memory requirement of each overloaded VM; doing so ensures that the SLAs are not violated even in the presence of peak workloads.

Estimating peak CPU and network bandwidth needs: Distribution profiles are used to estimate the peak CPU and network bandwidth needs of each VM. The tail of the usage distribution represents the peak usage over the recent past and is used as an estimate of future peak needs. This is achieved by computing a high percentile (e.g., the 95^{th} percentile) of the CPU and network bandwidth distribution as an initial estimate of the peak needs.

Since both the CPU scheduler and the network packet scheduler in Xen are work-conserving, a VM can use more than its fair share, provided that other VMs are not using their full allocations. In case of the CPU, for instance, a VM can use a share that exceeds the share determined by its weight, so long as other VMs are using less than their weighted share. In such instances, the tail of the distribution will exceed the guaranteed share and provide insights into the actual peak needs of the application. Hence, a high percentile of the distribution is a good first approximation of the peak needs.

However, if *all* VMs are using their fair shares, then an overloaded VM will not be allocated a share that exceeds its guaranteed allocation, even though its peak needs are higher than the fair share. In such cases, the observed peak usage (i.e., the tail of the distribution) will equal its fair-share. In this case, the tail of the distribution will *under-estimate* the actual peak need. To correct for this under-estimate, the provisioning component must scale the observed peak to better estimate the actual peak. Thus, whenever the CPU or the network interface on the physical server are close to saturation, the provisioning component first computes a high-percentile of the observed distribution and then adds a constant Δ to scale up this estimate.

Example *Consider two virtual machines that are assigned CPU weights of 1:1 resulting in a fair share of 50% each. Assume that VM*$_1$ *is overloaded and requires 70% of the CPU to meet its peak needs. If VM*$_2$ *is underloaded and only using 20% of the CPU, then the work-conserving Xen scheduler will allocate 70% to VM*$_1$*. In this case, the tail of the observed distribution is a good inddicator of VM*$_1$*'s peak need. In contrast, if VM*$_2$ *is using its entire fair share of 50%, then VM*$_1$ *will be allocated exactly its fair share. In this case, the peak observed usage will be 50%, an underestimate of the actual peak need. Since Sandpiper can detect that the CPU is fully utilized, it will estimate the peak to be* $50 + \Delta$.

The above example illustrates a fundamental limitation of the black-box approach—it is not possible to estimate the true peak need when the underlying resource is fully utilized. The scale-up factor Δ is simply a guess and might end up over- or under-estimating the true peak.

Estimating peak memory needs: Xen allows an adjustable amount of physical memory to be assigned to each resident VM; this allocation represents a hard upper-bound that can not be exceeded regardless of memory demand and regardless of the memory usage in other VMs. Consequently, our techniques for estimating the peak CPU and network usage do not apply to memory. The provisioning component uses observed swap activity to determine if the current memory allocation of the VM should be increased. If swap activity exceeds the threshold indicating memory pressure, then the the current allocation is deemed insufficient and is increased by a constant amount Δ_m. Observe that techniques such as Geiger and hypervisor level caches that attempt to infer working set sizes by observing swap activity [62, 80] can be employed to obtain a better estimate of memory needs; however, our current prototype uses the simpler approach of increasing the allocation by a fixed amount Δ_m whenever memory pressure is observed.

5.5.2 Gray-box Provisioning

Since the gray-box approach has access to application-level logs, information contained in the logs can be utilized to estimate the peak resource needs of the application. Unlike the black-box approach, the peak needs can be estimated even when the resource is fully utilized.

To estimate peak needs, the peak request arrival rate is first estimated. Since the number of serviced requests as well as the the number of dropped requests are typically logged, the incoming request rate is the summation of these two quantities. Given the distribution profile of the arrival rate, the peak rate is simply a high percentile of the distribution. Let λ_{peak} denote the estimated peak arrival rate for the application.

Estimating peak CPU needs: An application model is necessary to estimate the peak CPU needs. Applications such as web and database servers can be modeled as G/G/1 queuing systems [125]. The behavior of such a G/G/1 queuing system can be captured using the following queuing theory result [68]:

$$\lambda_{cap} \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (d - s)} \right]^{-1} \tag{5.2}$$

where d is the mean response time of requests, s is the mean service time, and λ_{cap} is the request arrival rate. σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time, respectively. Note that response time includes the full queueing delay, while service time only reflects the time spent actively processing a request.

While the desired response time d is specified by the SLA, the service time s of requests as well as the variance of inter-arrival and service times σ_a^2 and σ_b^2 can be determined from the server logs. By substituting these values into Equation 5.2, a lower bound on request rate λ_{cap} that can be serviced by the virtual server is obtained. Thus, λ_{cap} represents the current capacity of the VM.

To service the estimated peak workload λ_{peak}, the current CPU capacity needs to be scaled by the factor $\frac{\lambda_{peak}}{\lambda_{cap}}$. Observe that this factor will be greater than 1 if the peak arrival rate exceeds the currently provisioned capacity. Thus, if the VM is currently assigned a CPU weight w, its allocated share needs to be scaled up by the factor $\frac{\lambda_{peak}}{\lambda_{cap}}$ to service the peak workload.

Estimating peak network needs: The peak network bandwidth usage is simply estimated as the product of the estimated peak arrival rate λ_{peak} and the mean requested file size b; this is the amount of data transferred over the network to service the peak workload. The mean request size can be computed from the server logs.

Estimating memory needs: Using OS level information about a virtual machine's memory utilization allows the gray box approach to more accurately estimate the amount of memory required by a virtual machine. The gray box approach can proactively adjust memory allocations when the OS reports that it is low on memory (but before swapping occurs). This data is also used to safely reduce the amount of memory allocated to VMs which are not using their full allotment, something which is impossible to do with only black box information about swapping.

5.6 Hotspot Mitigation

Once a hotspot has been detected, Sandpiper must determine if the hotspots can be resolved with local resource adjustments, or if migrations are required to balance load between hosts.

5.6.1 VM Resizing

While large changes in resource needs may require migration between servers, some hot spots can be handled by adjusting the resource allocation of the overloaded VM. Sandpiper first attempts to increase the resource allocation for an overloaded VM by either adding additional CPUs, network interfaces, or memory depending on which resource utilizations exceeded the warning thresholds.

If the profiling engine detects that a VM is experiencing an increasing usage of CPU, Sandpiper will attempt to allocate an additional virtual CPU to the VM. Xen and other virtualization platforms support dynamic changes in the number of CPUs a VM has access to by exploiting hot-swapping code that already exists in many operating system kernels. A similar approach can be used to add network interfaces to a VM, although this is not currently supported by Sandpiper.

In many cases, memory hotspots can also be resolved through local provisioning adjustments. When a VM has insufficient memory as detected by either swapping (black box) or OS statistics (gray box), Sandpiper will first attempt to increase the VM's memory allocation on its current host. Only if there is insufficient spare memory will the VM be migrated to a different host.

5.6.2 Load Balancing with Migration

If there are insufficient spare resources on a host, the migration manager invokes its hotspot mitigation algorithm to determine *which* virtual servers to migrate and *where* in order to dissipate the hotspot. Determining a new mapping of VMs to physical servers that avoids threshold violations is NP-hard—the multi-dimensional bin packing problem can be reduced to this problem, where each physical server is a bin with dimensions corresponding to its resource constraints and each VM is an object that needs to be packed with size equal to its resource requirements. Even the problem of determining if a valid packing exists is NP-hard.

Consequently, our hotspot mitigation algorithm resorts to a heuristic to determine which overloaded VMs to migrate and where *such that migration overhead is minimized.* Reducing the migration overhead (i.e., the amount of data transferred) is important, since Xen's live migration mechanism works by iteratively copying the memory image of the VM to the destination while keeping track of which pages are being dirtied and need to be resent. This requires Xen to intercept all memory accesses for the migrating domain, which significantly impacts the performance of the application inside the VM. By reducing the amount of data copied over the network, Sandpiper can minimize the total migration time, and thus, the performance impact on applications. Note that network bandwidth available for application use is also reduced due to the background copying during migrations; however, on a gigabit LAN, this impact is small.

Capturing Multi-dimensional Loads: Once the desired resource allocations have been determined by either our black-box or gray-box approach, the problem of finding servers with sufficient idle resource to house overloaded VMs is identical for both. The migration manager employs a greedy heuristic to determine which VMs need to be migrated. The basic idea is to move load from the most overloaded servers to the least-overloaded servers, while attempting to minimize data copying incurred during migration. Since a VM or a server can be overloaded along one or more of three dimensions–CPU, network and memory–we define a new metric that captures the combined CPU-network-memory load of a virtual and physical server. The *volume* of a physical or virtual server is defined as the product of its CPU, network and memory loads:

$$Vol = \frac{1}{1-cpu} * \frac{1}{1-net} * \frac{1}{1-mem} \tag{5.3}$$

where cpu, net and mem are the corresponding utilizations of that resource normalized by the number of CPUs and network interfaces allocated to the virtual or physical server.[2] The higher the utilization of a resource, the greater the volume; if multiple resources are heavily utilized, the above product results in a correspondingly higher volume. The volume captures the degree of (over)load along multiple dimensions in a unified fashion and can be used by the mitigation algorithms to handle all resource hotspots in an identical manner.

Migration Phase: To determine which VMs to migrate, the algorithm orders physical servers in decreasing order of their volumes. Within each server, VMs are considered in decreasing order of their *volume-to-size ratio (VSR)*; where VSR is defined as *Volume/Size*; size is the memory footprint of the VM. By considering VMs in VSR order, the algorithm attempts to migrate the maximum volume (i.e., load) per unit byte moved, which has been shown to minimize migration overhead [116].

The algorithm proceeds by considering the highest VSR virtual machine from the highest volume server and determines if it can be housed on the least volume (least loaded) physical server. The move is feasible only if that server has sufficient idle CPU, network and memory resources to meet the desired resource allocation of the candidate VM as determined by the provisioning component (Section 5.5). Since we use VSR to represent three resource quantities, the least loaded server may not necessarily "fit" best with a particular VM's needs. If sufficient resources are not available, then the algorithm examines the next least loaded server and so on, until a match is found for the candidate VM. If no physical server can house the highest VSR VM, then the algorithm moves on to the next highest VSR VM and attempts to move it in a similar fashion. The process repeats until the utilizations of all resources on the physical server fall below their thresholds.

[2]If a resource is fully utilized, its utilization is set to $1-\epsilon$, rather than one, to avoid infinite volume servers. Also, since the black-box approach is oblivious of the precise memory utilization, the value of mem is set to 0.5 in the absence of swapping and to $1-\epsilon$ if memory pressure is observed; the precise value of mem is used in the gray-box approach.

The algorithm then considers the next most loaded physical server that is experiencing a hotspot and repeats the process until there are no physcial servers left with a hotspot. The output of this algorithm is a list of overloaded VMs and a new destination server for each; the actual migrations are triggered only after all moves have been determined.

Swap Phase: In cases where there aren't sufficient idle resources on less loaded servers to dissipate a hotspot, the migration algorithm considers VM swaps as an alternative. A swap involves exchanging a high VSR virtual machine from a loaded server with one or more low VSR VMs from an underloaded server. Such a swap reduces the overall utilization of the overloaded server, albeit to a lesser extent than a one-way move of the VM. Our algorithm considers the highest VSR VM on the highest volume server with a hotspot; it then considers the lowest volume server and considers the k lowest VSR VMs such that these VMs collectively free up sufficient resources to house the overloaded VM. The swap is considered feasible if the two physical servers have sufficient resources to house the other server's candidate VM(s) without violating utilization thresholds. If a swap cannot be found, the next least loaded server is considered for a possible swap and so on. The process repeats until sufficient high VSR VMs have been swapped with less loaded VMs so that the hotspot is dissipated. Although multi-way swaps involving more than two servers can also be considered, our algorithm presently does not implement such complex swaps. The actual migrations to perform the swaps are triggered only after a list of all swaps is constructed. Note that a swap may require a third server with "scratch" RAM to temporarily house a VM before it moves to its final destination. An alternative is to (i) suspend one of the VMs on disk, (ii) use the freed up RAM to accommodate the other VM, and (iii) resume the first VM on the other server; doing so is not transparent to the temporarily suspended VM.

5.7 Implementation and Evaluation

The implementation of Sandpiper is based on Xen. The Sandpiper *control plane* is implemented as a daemon that runs on the control node. It listens for periodic usage reports from the various nuclei, which are used to generate profiles. The profiling engine currently uses a history of the past 200 measurements to generate virtual and physical server profiles. The hotspot detector uses these profiles to detect hotspots; currently a hotspot is triggered when 3 out of 5 past readings and the next predicted value exceed a threshold. The default threshold is set to 75%. The migration manager implements our provisioning and hotspot mitigation algorithms; it notifies the nuclei of any desired migrations, which then trigger them. In all, the control plane consists of less than 750 lines of Python code.

The Sandpiper *nucleus* is a Python application that extends the XenMon CPU monitor to also acquire network and memory statistics for each VM. The monitoring engine in the nucleus collects and reports measurements once every 10 seconds—the default measurement interval. The nucleus uses Xen's Python management API to trigger migrations and adjust resource allocations as directed by the control plane. While black-box monitoring only requires access to domain-0 events, gray-box monitoring employs two additional components: a Linux OS daemon and an Apache module.

The gray-box linux daemon runs on each VM that permits gray-box monitoring. It currently gathers memory statistics via the `/proc` interface—the memory utilization, the number of free pages and swap usage are reported to the monitoring engine in each interval. The gray-box apache module comprises of a real-time log analyzer and a dispatcher. The log-analyzer processes log-entries as they are written to compute statistics such as the service time, request rate, request drop rate, inter-arrival times, and request/file sizes. The dispatcher is implemented as a kernel module based on Linux IP Virtual server (IPVS) ver 1.2.1; the goal of the kernel module is to accurately estimate the request arrival rate during overload periods, when a high fraction of requests may be dropped. Since requests can be dropped at the TCP layer as well as at the HTTP layer during overloads, the use of a

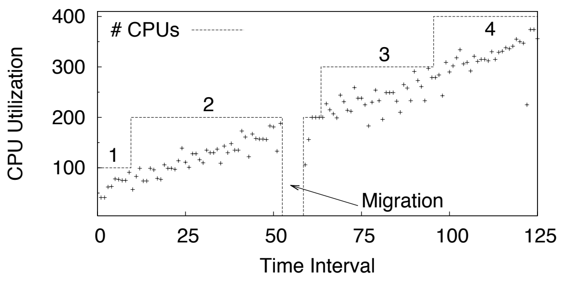


Figure 5.3. Sandpiper increases the number of virtual CPU cores allocated to a VM. A migration is required to move from a 2 to 4 core PM.

transport-level dispatcher such as IPVS is necessary for accurately estimating the drop (and hence arrival) rates. Ordinarily, the kernel dispatcher simply forwards incoming requests to Apache for processing. In all, the nucleus comprises 650 lines of Python code.

Our evaluation of Sandpiper is based on a prototype data center consisting of twenty 2.4Ghz Pentium-4 servers connected over a gigabit Ethernet. These servers run Linux 2.6 and Xen 3.0.2-3 and are equipped with at least 1GB of RAM. Experiments involving multi-core systems run on Intel Quad-Core servers with 4GB of RAM and Xen 3.1. A cluster of Pentium-3 Linux servers is used to generate workloads for our experiments. One node in the cluster is designated to run the Sandpiper control plane, while the rest host one or more VMs, all of which run the Sandpiper nucleus in domain- 0. In the following experiments, our VMs run Apache 2.0.54, PHP 4.3.10, and MySQL 4.0.24.

5.7.1 VM Resizing

While migrations are necessary for large changes in resource allocations, it is less expensive if resources can be adjusted locally without the overhead of migration. This experiment demonstrates Sandpiper's ability to detect increasing CPU requirements and respond by allocating additional CPU cores to the virtual machine.

Initially, a VM running a CPU intensive web application is allocated a single CPU core. During the experiment, the number of clients accessing the web server increases. Sandpiper responds by increasing the number of virtual CPUs allocated to the VM. The VM starts on

VM	Peak 1	Peak 2	Peak 3	RAM (MB)	Start PM
1	200	130	130	256	1
2	90	90	90	256	1
3	60	200	60	256	2
4	60	90	90	256	2
5	10	10	130	128	3

Table 5.1. Workload in requests/second, memory allocations, and initial placement.

a dual core host; as the load continues to rise, a migration is required to move the VM to a host with four available CPUs as shown in Figure 5.3.

Result: Resizing a VM's resource allocation incurs little overhead. When additional resources are not available locally, migrations are required.

5.7.2 Migration Effectiveness

Our next experiment exercises Sandpiper's hotspot detection and migration algorithms; we subject a set of black-box servers to a series of workloads that repeatedly place the system in overload. Our experiment uses three physical servers and five VMs with memory allocations as shown in Table 5.1. All VMs run Apache serving dynamic PHP web pages. The PHP scripts are designed to be CPU intensive so that a low client request rate places a large CPU load on a server without significant network or memory utilization. We use *httperf* to inject a workload that goes through three phases, each of which causes a hotspot on a different physical machine. The peak request rates for each phase are shown in Table 5.1.

Figure 5.4 presents a time series of the load placed on each VM along with the triggered migrations. In the first phase, a large load is placed on VM_1, causing the CPU utilization on PM_1 to exceed the CPU threshold. The system detects a hotspot at t=166 seconds. The migration manager examines candidates for migration in VSR order. VM_1 has the highest VSR, so it is selected as a candidate. Since PM_3 has sufficient spare capacity to house VM_1, it is migrated there, thereby eliminating the hotspot less than 20 seconds after

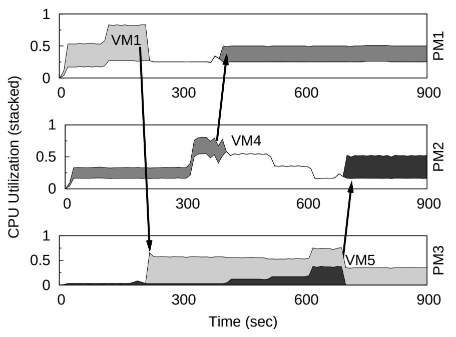


Figure 5.4. A series of migrations resolve hotspots. Different shades are used for each migrating VM.

detection. This represents the ideal case for our algorithm: if possible, we try to migrate the most loaded VM from an overloaded PM to one with sufficient spare capacity.

In the second phase, PM_2 becomes overloaded due to increasing load on VM_3. However, the migration manager is unable to migrate this VM because there is insufficient capacity on the other PMs. As a result, at t=362 seconds, the VM on PM_2 with the second highest VSR VM_4, is migrated to PM_1 that now has spare capacity. This demonstrates a more typical case where none of the underloaded PMs have sufficient spare capacity to run the overloaded PM's highest VSR VM, so instead we migrate less overloaded VMs that can fit elsewhere.

In the final phase, PM_3 becomes overloaded when both its VMs receive *identical* large loads. Unlike the previous two cases where candidate VMs had equal memory footprints, VM_5 has half as much RAM as VM_1, so it is chosen for migration. By selecting the VM with a lower footprint, Sandpiper maximizes the reduction in load per megabyte of data transfered.

Result: To eliminate hotspots while minimzing the overhead of migration, our placement algorithm tries to move the highest VSR VM to the least loaded PM. This maximizes the amount of load displaced from the hotspot per megabyte of data transferred.

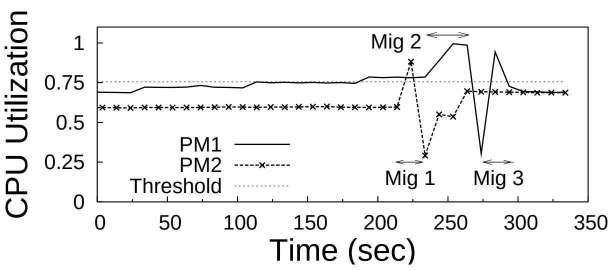


Figure 5.5. After a hotspot is detected on PM_1, a swap occurs via the scratch PM. Initially, VM_1 and VM_2 are on PM_1. VM_2 is swapped with VM_4 from PM_2 to resolve overload.

5.7.3 Virtual Machine Swaps

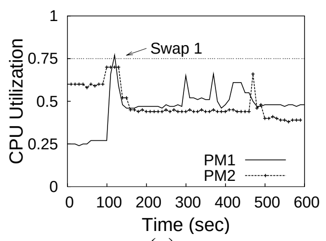
Next we demonstrate how VM swaps can mitigate hotspots. The first two VMs in our setup are allocated 384 MB of RAM on PM_1; VM_3 and VM_4 are assigned 256 MB each on PM_2. The load on VM_1 steadily increases during the experiment, while the others are constant. As before, clients use httperf to request dynamic PHP pages.

Figure 5.5 shows that a hotspot is detected on PM_1 due to the increasing load on VM_1. However, there is insufficient spare capacity on PM_2 to support a migrated VM. The only viable solution is to swap VM_2 with VM_4. To facilitate such swaps, Sandpiper uses spare RAM on the control node as scratch space.

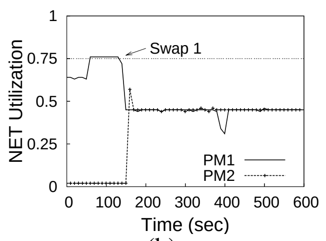
By utilizing this scratch space, Sandpiper never requires either physical server to simultaneously run both VMs. It does require us to perform three migrations instead of two; however, Sandpiper reduces the migration cost by always migrating the smaller footprint VM via the scratch server. As shown in Figure 5.5, the load on PM_2 drops at t=219 due to the migration of VM_4 to scratch. VM_2 is then migrated directly from PM_1 to PM_2 at t=233, followed by a migration of VM_4 from scratch to PM_1, which takes an additional 10s.

The figure also depicts the CPU overhead of a migration—as indicated by a rise in CPU utilization of the initiating server whenever a migration begins. This suggests using lower CPU hotspot thresholds to safely absorb the additional overheads caused by a migration.

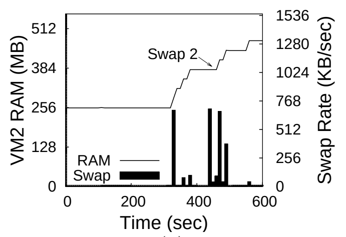
Result: Swaps incur more overhead, but increase the chances of mitigating hotspots in clusters with high average utilization.



(a)



(b)



(c)

VM	% CPU Util.	% Net Util.
1	9	39
2	10	38
3	28	1
4	31	1

(d)

Figure 5.6. Swaps and migrations to handle network- and memory-intensive loads. Initially, VM_1 and VM_2 are on PM_1, the rest on PM_2. After two swaps, PM_1 hosts VM_1 and VM_4.

5.7.4 Mixed Resource Workloads

Sandpiper can consolidate applications that stress different resources to improve the overall multiplexing of server resources. Our setup comprises two servers with two VMs each. Both VMs on the first server are network-intensive, involving large file transfers, while those on the second server are CPU-intensive running Apache with dynamic PHP scripts. All VMs are initially allocated 256 MB of memory. VM_2 additionally runs a main-memory database that stores its tables in memory, causing its memory usage to grow over time.

Figures 5.6(a) and (b) show the resource utilization of each PM over time. Since PM_1 has a network hotspot and PM_2 has a CPU hotspot, Sandpiper swaps a network-intensive VM for a CPU-intensive VM at t=130. This results in a lower CPU and network utilization on both servers. Figure 5.6(d) shows the initial utilizations of each VM; after the swap, the aggregate CPU and network utilizations on both servers falls below 50%.

In the latter half, memory pressure increases on VM_2 due to its main-memory database application. As shown in 5.6(c), Sandpiper responds by increasing the RAM allocation in

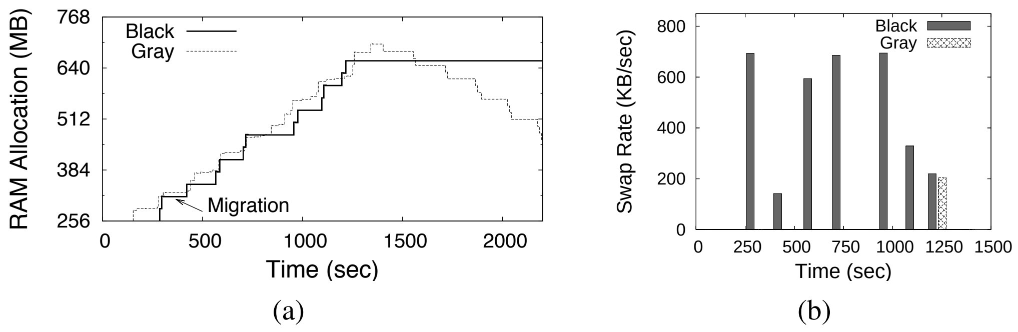


Figure 5.7. The black-box system lags behind the gray-box system in allocating memory. The gray-box approach proactively increases memory and safely reduces the VM's memory allocation when demand falls.

steps of 32MB every time swapping is observed; when no additional RAM is available, the VM is swapped to the second physical server at t=430. This is feasible because two cpu-intensive jobs are swapped, leaving CPU and network utilization balanced, and the second physical server has more RAM than the first. Memory allocations are reactive since only black-box stats are available. Next we demonstrate how a gray-box approach can proactively respond to memory pressure.

Result: Sandpiper can respond to network, CPU, or memory hotspots and can collocate VMs that stress different resources to improve overall system utilization.

5.7.5 Gray v. Black: Memory Allocation

We compare the effectiveness of the black- and gray-box approaches in mitigating memory hotspots using the SPECjbb 2005 benchmark. SPECjbb emulates a three-tier web application based on J2EE servers. We use SPECjbb to apply an increasingly intense workload to a single VM. The workload increases every two minutes, causing a significant increase in memory usage. After twenty minutes, the application reaches its peak intensity, after which the workload decreases at a similar rate.

The VM is initially assigned 256MB of RAM, and resides on a physical machine with 384 MB total RAM. We also run a second, idle physical server which has 1GB RAM. We run the experiment with two separate pairs of servers, Black and Gray, that correspond to the black- and gray-box approaches, respectively. The Gray system is configured to signal a hotspot whenever the amount of free RAM in the virtual machine falls below 32MB.

Fig. 5.7(a) plots the memory allocation of the VM over time. Both systems gradually increase the VM's memory until all unused RAM is exhausted. Since Black can only respond to swapping, it lags in responsiveness. At t=380 seconds, Gray determines that there is insufficient RAM for the VM and migrates it to the second PM with 1GB RAM; Black initiates the same migration shortly afterward. Both continue to increase the VM's memory as the load rises. Throughout the experiment, Black writes a total of 32MB to swap, while Gray only writes 2MB. Note that a lower memory hotspot threshold in Gray can prevent swapping altogether, while Black can not eliminate swapping due to its reactive nature.

During the second phase of the trial, Gray is able to detect the decreasing memory requirements and is able to safely reduce the VM's memory allocation. Since the black-box system can only detect swapping, it cannot reduce the memory allocation without fear of causing swapping and worse performance.

Result: A key weakness of the black-box approach is its inability to infer memory usage. Using this information, the gray-box system can reduce or eliminate swapping and can safely decrease a VM's memory allocation.

5.7.6 Gray v. Black: Apache Performance

Recall from Section 5.5 that when resources are fully utilized, they hamper the black-box approach from accurately determining the needs of overloaded VMs. This experiment demonstrates how a black-box approach may incur extra migrations to mitigate a hotspot, whereas a gray-box approach can use application-level knowledge for faster hotspot mitigation.

Our experiment employs three physical servers and four VMs. Initially, VM_1 through VM_3 reside on PM_1, VM_4 resides on PM_2, and PM_3 is idle. We use httperf to generate

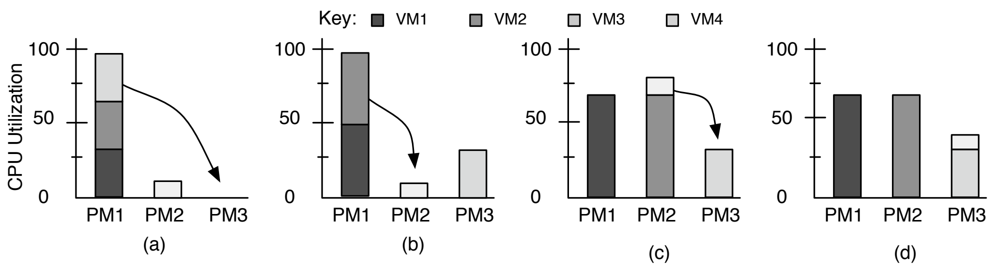


Figure 5.8. The black-box system incorrectly guesses resource requirements since CPU usage is saturated, resulting in an increased resolution time. The gray-box system infers usage requirements and transitions directly from a) to d).

requests for CPU intensive PHP scripts on all VMs. At t=80s, we rapidly increase the request rates on VM_1 and VM_2 so that actual CPU requirement for *each* virtual machine reaches 70%, creating an extreme hotspot on PM_1. The request rates for VM_3 and VM_4 remain constant, requiring 33% and 7% CPU respectively. We use an aggressive 6 second measurement interval so that Sandpiper can respond quickly to the increase in workload.

Without accurate estimates of each virtual machine's resource requirements, the black-box system falters in its decision making as indicated in Figure 5.8. Since the CPU on PM_1 is saturated, each virtual machine receives an equal portion of processing time and appears equivalent to Sandpiper. Sandpiper must select a VM at random, and in the worst case, tries to eliminate the hotspot by migrating VM_3 to PM_3. Since VM_1 and VM_2 continue to reside on PM_1, the hotspot persists even after the first migration. Next, the black-box approach assumes that VM_2 requires only 50% of the CPU and migrates it to PM_2. Unfortunately, this results in PM_2 becoming overloaded, so a final migration must be performed to move VM_4 to PM_3.

We repeat this scenario with the Apache gray-box module running inside of each virtual machine. Since the gray-box monitor can precisely measure the incoming request rates, Sandpiper can accurately estimate the CPU needs of VM_1 and VM_2. By using this information, Sandpiper is able to efficiently respond to the hotspot by immediately migrating VM_3 to PM_2 and VM_2 to PM_3. Figure 5.9 depicts the improved performance of the gray-box

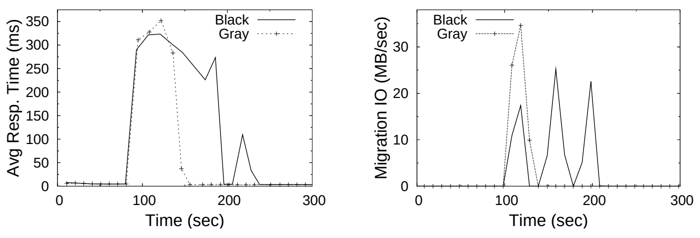


Figure 5.9. The gray-box system balances the system more quickly due to more informed decision making. The black-box system must perform migrations sequentially and incurs an additional migration.

approach. Note that since Sandpiper requires the hotspot to persist for k out of n intervals before it acts, it is not until $t = 98$s that either system considers itself overloaded. Once a hotspot is flagged, the gray-box approach can mitigate it within 40 seconds with just two migrations, while the black-box approach requires 110 seconds and three migrations to do so. Although response time increases equally under both systems, the gray-box approach is able to reduce response times to an acceptable level 61% faster than the black-box system, producing a corresponding reduction in SLA violations.

Result: Application-level statistics enable the gray-box approach to better infer resource needs and improves the quality of migration decisions, especially in scenarios where resource demands exceed server capacity.

5.7.7 Prototype Data Center Evaluation

Next we conduct an experiment to demonstrate how Sandpiper performs under realistic data center conditions. We deployed a prototype data center on a cluster of 16 servers that run a total of 35 VMs. An additional node runs the control plane and one node is reserved as a scratch node for swaps. The virtual machines run a mix of data center applications ranging from Apache and streaming servers to LAMP servers running Apache, PHP, and MySQL within a single VM. We run RUBiS on our LAMP servers—RUBiS is an open-source

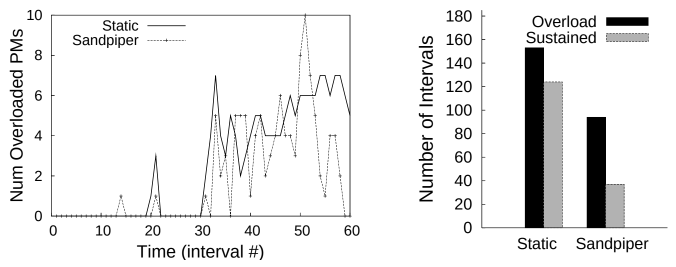


Figure 5.10. Sandpiper eliminates all hotspots and reduces the number of intervals experiencing sustained overload by 61% .

multi-tier web application that implements an eBay-like auction web site and includes a workload generator that emulates users browsing and bidding on items.

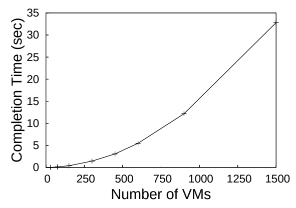
Of the 35 deployed VMs, 5 run the RUBiS application, 5 run streaming servers, 5 run Apache serving CPU-intensive PHP scripts, 2 run main memory database applications, and the remaining 15 serve a mix of PHP scripts and large HTML files. We use the provided workload generators for the RUBiS applications and use httperf to generate requests for the other servers.

To demonstrate Sandpiper's ability to handle complex hotspot scenarios, we orchestrate a workload that causes multiple network and CPU hotspots on several servers. Our workloads causes six physical servers running a total of 14 VMs to be overloaded—four servers see a CPU hotspot and two see a network hotspot. Of the remaining PMs, 4 are moderately loaded (greater than 45% utilization for at least one resource) and 6 have lighter loads of between 25 and 40% utilization. We compare Sandpiper to a statically allocated system with no migrations.

Figure 5.10 demonstrates that Sandpiper eliminates hotspots on all six servers by interval 60. These hotspots persist in the static system until the workload changes or a system administrator triggers manual migrations. Due to Xen's migration overhead, there are brief periods where Sandpiper causes more physical servers to be overloaded than in the static case. Despite this artifact, even during periods where migrations are in progress, Sandpiper reduces the number of intervals spent in sustained overload by 61%. In all, Sandpiper

Log Type	Bytes
Black-box	178
Gray: Mem	60
Gray: Apache	50
Total	288

(a)



(b)

Figure 5.11. Sandpiper overhead and scalability

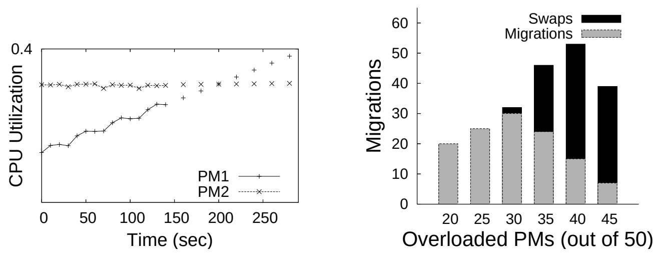
performs seven migrations and two swaps to eliminate all hotspots over a period of 237 seconds after hotspot detection.

Result: Sandpiper is capable of detecting and eliminating simultaneous hotspots along multiple resource dimensions. Despite Xen's migration overhead, the number of servers experiencing overload is decreased even while migrations are in progress.

5.7.8 System Overhead and Scalability

Sandpiper's CPU and network overhead is dependent on the number of PMs and VMs in the data center. With only black-box VMs, the type of application running in the VM has no effect on Sandpiper's overhead. If gray-box modules are in use, the overhead may vary depending on the size of application-level statistics gathered.

Nucleus Overheads: Sandpiper's nucleus sends reports to the Control Plane every measurement interval (10 seconds by default). The table in Figure 5.11(a) gives a breakdown of overhead for each report type. Since each report uses only 288 bytes per VM, the resulting overhead on a gigabit LAN is negligible. To evaluate the CPU overhead, we compare the performance of a CPU benchmark with and without our resource monitors running. Even on a single physical server running 24 concurrent VMs, our monitoring overheads only reduce the CPU benchmark performance by approximately one percent. This is comparable to the overheads reported by XenMon, which much of our code is based on [46].



(a) (b)

Figure 5.12. (a) Using time series predictions (the dotted lines) allows Sandpiper to better select migration destinations, improving stability. (b) Higher levels of overload requires more migrations until there is no feasible solution.

Control Plane Scalability: The main source of computational complexity in the control plane is the computation of a new mapping of virtual machines to physical servers after detecting a hotspot. Although the problem is NP-hard, we only require an approximate solution, and our heuristics make the problem tractable for reasonable system sizes. For very large data centers with tens or hundreds of thousands of virtual machines, the servers can be broken up into pools, each controlled independently by its own control plane. For clusters of up to 500 virtual servers, the algorithm completes in less than five seconds as shown in Figure 5.11(b).

5.7.9 Stability During Overloads

This section demonstrates how Sandpiper ensures stable system behavior by avoiding "thrashing" migrations. First, Sandpiper avoids migrations to physical machines with rising loads, since this can trigger additional migrations if the load rises beyond the threshold; time-series predictions are used to determine future load trends when selecting a physical server. Thus, Figure 5.12(a) shows that when a migration decision is required at t=140 sec, Sandpiper will prefer PM_2 over PM_1 as a target. Even though PM_2 has a *higher* current load, the 120 second prediction window indicates a rising load on PM_1.

Next, we demonstrate Sandpiper's behavior in the presence of increasing number of hotspots. We simulate a data center with fifty physical servers, each with three virtual servers. We increase the number of simultaneous hotspots from 20 to 45; the mean utilizations are set to 85% and 45% for servers with and without hotspots. Figure 5.12(b) depicts the mean number of migrations performed to resolve these hotspots over multiple runs. If fewer than half of the servers are overloaded, then all hotspots can typically be resolved with one migration per overloaded server. After this threshold, swaps are required and it is increasingly difficult to fully resolve overload until it becomes infeasible. With 35 overloaded servers, Sandpiper was able to eliminate all hotspots 73% of the time (over multiple runs); with 40 overloaded servers, a complete solution was found only 3% of the time. In the extreme case, Sandpiper is still able to resolve 22 of the 45 hotspots before giving up. In all cases, Sandpiper first finds a solution before initiating migrations or swaps; when no feasible solutions are found, Sandpiper either implements a partial solution or gives up entirely rather than attempting wasteful migrations. This bounds the number of migrations which will ever be performed and explains the decrease in migrations beyond 40 overloaded servers, where there is no feasible solution.

5.7.10 Tuning Sandpiper

Sandpiper has several parameters which the system administrator can tune to make hotspot detection and mitigation more or less aggressive. Our experiments suggest the following rules of thumb:

Setting Thresholds: If overload thresholds are set too high, then the additional overhead during migration can cause additional SLA violations. Our experiments show that the average throughput of a CPU-intensive Apache server can drop by more than 50% during a migration. We suggest a CPU threshold of 75% to absorb the CPU overhead of migration while maximizing server utilization. We also suggest a 75% threshold for network utilization based on experiments in [30] which indicate that the network throughput of a highly loaded server can drop by about 20% during portions of a migration (due to network copying overheads).

Sustained Overload Requirement: Our experiments (not reported here) reveal that Sandpiper is not sensitive to a particular choice of the measurement interval $\mathcal{I}$ so long as it is between a few seconds and a few tens of seconds. For a measurement interval of 10s, we suggest $k = 3$ and $n = 5$ for the "k out of n" check; this corresponds to requiring the time period of about 3 migrations to exceed the resource threshold before we initiate a migration. The Δ paramter is used in the black-box system to increase resource allocations when utilization is saturated. This should be set equal to the maximum increase in resource requirements that a service is likely to see during a measurement interval and may vary based on workload; we use 10% in our experiments. Using more advanced time series forecasting techniques would allow Sandpiper to dynamically determine Δ.

5.8 Related Work

Our work draws upon recent advances in virtual machines and dynamic provisioning in data centers to address a question of increasing research and commercial interest: can virtual machine migration enable robust and highly responsive provisioning in data centers? The Xen migration work [30] alludes to this motivation. What is missing is a convincing validation and algorithms to effect migration, which is the focus of this chapter.

The idea of process migration was first investigated in the 80's [121]. Support for migrating groups of processes across OSes was presented in [93], but applications had to be suspended and it did not address the problem of maintaining open network connections. Virtualization support for commodity operating systems in [42] led towards techniques for virtual machine migration over long time spans, suitable for WAN migration [107]. More recently, Xen [30] and VMWare [91] have implemented "live" migration of VMs that involve extremely short downtimes ranging from tens of milliseconds to a second. VM migration has been used for dynamic resource allocation in Grid environments [105, 117, 43]. A system employing automated VM migrations for scientific nano-technology workloads on federated grid environments was investigated in [105]. The Shirako system provides infrastructure for leasing resources within a federated cluster environment and was extended to use virtual machines for more flexible resource allocation in [43]. Shirako uses migrations to enable dynamic placement decisions in response to resource broker and cluster provider policies. In contrast, we focus on data center environments with stringent SLA requirements that necessitate highly responsive migration algorithms for online load balancing. VMware's Distributed Resource Scheduler [138] uses migration to perform automated load balancing in response to CPU and memory pressure. DRS uses a userspace application to monitor memory usage similar to Sandpiper's gray box monitor, but unlike Sandpiper, it cannot utilize application logs to respond directly to potential SLA violations or to improve placement decisions.

Dedicated hosting is a category of dynamic provisioning in which each physical machine runs at most one application and workload increases are handled by spawning a new replica of the application on idle servers. Physical server granularity provisioning has been investigated in [6, 101]. Techniques for modeling and provisioning multi-tier Web services by allocating physical machines to each tier are presented in [125]. Although dedicated hosting provides complete isolation, the cost is reduced responsiveness - without virtualization, moving from one physical machine to another takes on the order of several minutes [125] making it unsuitable for handling flash crowds. Our current implementation does not replicate virtual machines, implicitly assuming that PMs are sufficiently provisioned.

Shared hosting is the second variety of dynamic provisioning, and allows a single physical machine to be shared across multiple services. Various economic and resource models to allocate shared resources have been presented in [25]. Mechanisms to partition and share resources across services include [8, 25]. A dynamic provisioning algorithm to allocate CPU shares to VMs on a single physical machine (as opposed to a cluster) was presented and evaluated through simulations in [85]. In comparison to the above systems, our work assumes a shared hosting platform and uses VMs to partition CPU, memory, and network resources, but additionally leverages VM migration to meet SLA objectives.

Estimating the resources needed to meet an appliction's SLA requires a model that inspects the request arrival rates for the application and infers its CPU, memory, and network bandwidth needs. Developing such models is not the focus of this work and has been addressed by several previous efforts such as [64, 6].

5.9 Sandpiper Conclusions

This chapter argued that virtualization provides new and more effective tools for online resource management in data centers. The speed of VM migrations and the ability to dynamically adjust resource shares allows for more agile responses to varying workload demands. We designed Sandpiper to automate the task of monitoring and detecting hotspots, determining a new mapping of physical to virtual resources, and resizing or migrating VMs to eliminate server overload. Sandpiper supports both a black-box strategy that is fully OS- and application-agnostic as well as a gray-box approach that can exploit OS- and application-level statistics.

Our evaluation of Sandpiper demonstrates the effectiveness of VM migration as a technique for rapid hotspot elimination. Using solely black-box methods, Sandpiper is capable of eliminating simultaneous hotspots involving multiple resources. We found that utilizing gray-box information can improve the responsiveness of our system, particularly by allowing for proactive memory allocations and better inferences about resource requirements. Sandpiper improves the agility of data centers by automating the response to unpredictable workload changes.

CHAPTER 6

SEAMLESS CONNECTIVITY AND OPTIMIZED WAN MIGRATION

As more data centers are deployed across the Internet, businesses desire to run applications using pools of resources from several data centers. Unfortunately, current data centers and cloud platforms provide limited mechanisms for seamlessly connecting and managing the resources of multiple data center sites. The limitations on connectivity between data center sites means that the VM migration techniques that were so useful in the previous two chapters cannot be used to move applications *between* geographically separated data centers. In this chapter we expand our view from looking at a single data center to considering multiple data centers owned by a business or the set of data centers that make up a cloud computing platform and its customers. We propose a virtual network infrastructure which can be used to bridge multiple data center sites in a seamless and secure way, and an optimized form of VM migration that minimizes the performance and bandwidth cost of moving live applications across the Internet.

6.1 Background and Motivation

From an IT perspective, it would be ideal if both in-house data centers and private and public clouds could be considered as a flexible pool of computing and storage resources that are seamlessly connected to overcome their geographical separation. The management of such a pool of resources requires the ability to flexibly map applications to different sites as well as the ability to move applications and their data across and within pools. The agility with which such decisions can be made and implemented determines the responsiveness with which enterprise IT can meet changing business needs.

As shown in the previous chapters, virtualization is a key technology that has enabled such agility *within* a data center. However, this same flexibility is also desirable *across* geographically distributed data centers. Such cross data center management requires efficient migration of both memory and disk state between such data centers, overcoming constraints imposed by the WAN connectivity between them.

In this chapter we propose CloudNet, the combination of a virtual network infrastructure and optimized VM migration tools that allow applications to be seamlessly moved across the Internet. CloudNet's optimized migration technique reduces the impact of migration on application performance, accounts for the limited bandwidth available on Internet links, and automates network redirection so that active network connections are not disrupted.

6.2 WAN Migration Use Cases

Virtual machine migration has become an important tool within data centers. We believe that WAN migration of VMs would likewise enable new resource management techniques, simplify deployment into cloud data centers, and promote new application architectures. Consider the following use cases that illustrate the potential benefits of efficient WAN migration:

Cloud bursting: Cloud bursting is a technique where an enterprise normally employs local servers to run applications and dynamically harnesses cloud servers to enhance capacity during periods of workload stress. The stress on local IT servers can be mitigated by temporarily migrating a few overloaded applications to the cloud or by instantiating new application replicas in the cloud to absorb some of the workload increase. These cloud resources are deallocated once the workload peak has ebbed. Cloud bursting eliminates the need to pre-provision for the peak workload locally, since cloud resources can be provisioned dynamically when needed, yielding cost savings due to the cloud's pay-as-you go model. Current cloud bursting approaches adopt the strategy of spawning new replicas of the application. This limits the range of enterprise applications that may use cloud bursting to stateless applications or those that include elaborate consistency mechanisms. Live migration permits *any* application to exploit cloud bursting while experiencing minimal downtime.

Enterprise IT Consolidation: Many enterprises with multiple data centers have attempted to deal with data center "sprawl" and cut costs by consolidating multiple smaller sites into a few large data centers. Such consolidation requires applications and data to be moved from one site to another over a WAN; a subset of these applications may also be moved to cloud platforms if doing so is more cost-effective than hosting locally. Typically such transformation projects have incurred application down-times, often spread over multiple days. Hence, the ability to implement these moves with minimal or no down-time is attractive due to the corresponding reduction in the disruption seen by a business.

Follow the sun: "Follow the sun" is a new IT strategy that is designed for project teams that span multiple continents. The scenario assumes multiple groups spanning different geographies that are collaborating on a common project and that each group requires low-latency access to the project applications and data during normal business hours. One approach is to replicate content at each site—e.g., a data center on each continent—and keep the replicas consistent. While this approach may suffice for content repositories or replicable applications, it is often unsuitable for applications that are not amenable to replication. In such a scenario, it may be simpler to migrate one or more VM containers with applications and project data from one site to another every evening; the migration overhead can be reduced by transferring only incremental state and applying it to the snapshot from the previous day to recreate the current state of the application.

These scenarios represent the spectrum from pre-planned to reactive migrations across data centers. Although the abstraction of treating resources that span data centers and cloud providers as a single unified pool of resources is attractive, the reality of these resources being distributed across significant geographic distances and interconnected via static wide area network links (WANs) conspire to make the realization of this vision difficult.

6.3 Cloudnet Overview

In this section, we present an overview of the key abstractions and design building blocks in CloudNet.

6.3.1 Resource Pooling: Virtual Cloud Pools

At the heart of CloudNet is a Virtual Cloud Pool (VCP) abstraction that enables server resources across data centers and cloud providers to be logically grouped into a single server pool as shown in Figure 6.1. The notion of a Virtual Cloud Pool is similar to that of a Virtual Private Cloud, which is used by Amazon's EC2 platform and was also proposed in our previous research [139]. Despite the similarity, the design motivations are different. In our case, we are concerned with grouping server pools across data centers, while Amazon's product seeks to provide the abstraction of a private cloud that is hosted on a public cloud. Both abstractions use virtual private networks (VPNs) as their underlying interconnection technology—we employ Layer 2 VPNs to implement a form of network virtualization/transparency, while Amazon's VPC uses layer 3 VPNs to provide control over the network addressing of VM services.

The VCPs provided by CloudNet allow cloud resources to be connected to as securely and seamlessly as if they were contained within the enterprise itself. Further, the cloud to enterprise mappings can be adjusted dynamically, allowing cross data center resource pools to grow and change depending on an enterprise's needs. In the following sections we discuss the benefits of these abstractions for enterprise applications and discuss how this dynamic infrastructure facilitates VM migration between data centers.

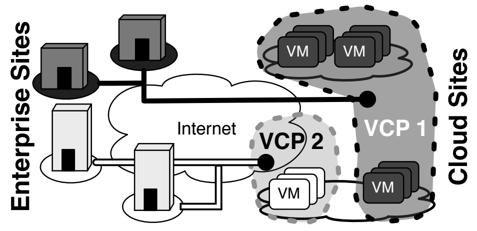


Figure 6.1. Two VCPs isolate resources within the cloud sites and securely link them to the enterprise networks.

6.3.2 Dynamic, Seamless Cloud Connections

CloudNet uses Multi-Protocol Label Switching (MPLS) based VPNs to create the abstraction of a private network and address space shared by multiple data centers. Since addresses are specific to a VPN, the cloud operator can allow customers to use any IP address ranges that they prefer without concern for conflicts between cloud customers. CloudNet makes the level of abstraction even greater by using *Virtual Private LAN Services (VPLS)* that bridge multiple MPLS endpoints onto a single LAN segment. This allows cloud resources to appear *indistinguishable* from existing IT infrastructure already on the enterprise's own LAN. VPLS provides transparent, secure, and resource guaranteed layer-2 connectivity without requiring sophisticated network configuration by end users. This simplifies the network reconfiguration that must be performed when migrating VMs between data centers.

VPNs are already used by many large enterprises, and cloud sites can be easily added as new secure endpoints within these existing networks. VCPs use VPNs to provide secure communication channels via the creation of "virtually dedicated" paths in the provider network. The VPNs protects traffic between the edge routers at each enterprise and cloud site. Within a cloud site, the traffic for a given enterprise is restricted to a particular VLAN. This provides a secure end-to-end path from the enterprise to the cloud and eliminates the need to configure complex firewall rules between the cloud and the enterprise, as all sites can be connected via a private network inaccessible from the public Internet.

As enterprises deploy and move resources between cloud data centers, it is necessary to adjust the topology of the client's VCP. In typical networks, connecting a new data center to a VPN endpoint can take hours or days, but these delays are administrative rather than fundamental to the network operations required. CloudNet utilizes a VPN Controller to automate the process of VPN reconfiguration, allowing resources at a new cloud data center to be connected to a VPN within seconds.

6.3.3 Efficient WAN Migration

Currently, moving an application to the cloud or another data center can require substantial downtime while application state is copied and networks are reconfigured before the application can resume operation. Alternatively, some applications can be easily replicated into the cloud while the original continues running; however, this only applies to a small class of applications (e.g. stateless web servers or MapReduce style data processing jobs). These approaches are insufficient for the majority of enterprise applications which have not been designed for ease of replication. Further, many legacy applications can require significant reconfiguration to deal with the changed network configuration required by current approaches. In contrast, the live VM migration supported by CloudNet provides a much more attractive mechanism for moving applications between data centers because it is completely application independent and can be done with only minimal downtime.

Most recent virtualization platforms support efficient migration of VMs within a local network [30, 91]. By virtue of presenting WAN resources as LAN resources, CloudNet's VCP abstraction allows these live migration mechanisms to function unmodified across data centers separated by a WAN. However, the lower bandwidth and higher latencies over WAN links result in poor migration performance. In fact, VMWare's preliminary support for WAN VM migration requires at least 622 Mbps of bandwidth dedicated to the transfer, and is designed for links with less than 5 msec latency [128]. Despite being interconnected using "fat" gigabit pipes, data centers will typically be unable to dedicate such high

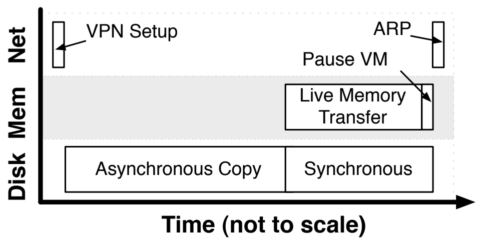


Figure 6.2. The phases of a migration for non-shared disk, memory, and the network in CloudNet .

bandwidth for a *single* application transfer and enterprises will want the ability to migrate a group of related VMs concurrently. CloudNet uses a set of optimizations to conserve bandwidth and reduce WAN migration's impact on application performance.

Current LAN-based VM migration techniques assume the presence of a shared file system which enables them to migrate only memory data and avoid moving disk state. A shared file system may not always be available across a WAN or the performance of the application may suffer if it has to perform I/O over a WAN. Therefore, CloudNet coordinates the hypervisor's memory migration with a disk replication system so that the entire VM state can be transferred if needed.

Current LAN-based live migration techniques must be optimized for WAN environments, and cloud computing network infrastructure must be enhanced to support dynamic relocation of resources between cloud and enterprise sites; these challenges are the primary focus of this chapter.

6.4 WAN VM Migration

Consider an organization which desires to move one or more applications (and possibly their data) between two data centers. Each application is assumed to be run in a VM, and we wish to live migrate those virtual machines across the WAN. CloudNet uses these steps to live migrate each VM:

Step 1: Establish virtual connectivity between VCP endpoints.

Step 2: If storage is not shared, transfer all disk state.

Step 3: Transfer the memory state of the VM to a server in the destination data center as it continues running without interruption.

Step 4: Once the disk and memory state have been transferred, briefly pause the VM for the final transition of memory and processor state to the destination host. This process must not disrupt any active network connections between the application and its clients.

While these steps, illustrated in Figure 6.2, are well understood in LAN environments, migration over the WAN poses new challenges. The constraints on bandwidth and the high latency found in WAN links makes steps 2 and 3 more difficult since they involve large data transfers. The IP address space in step 4 would typically be different when the VM moves between routers at different sites. Potentially, application, system, router and firewall configurations would need to be updated to reflect this change, making it difficult or impossible to seamlessly transfer active network connections. CloudNet avoids this problem by virtualizing the network connectivity so that the VM appears to be on the same virtual LAN. We achieve this using VPLS VPN technology in step 1, and CloudNet utilizes a set of migration optimizations to improve performance in the other steps.

6.4.1 Migrating Networks, Disk, and Memory

Here we discuss the techniques used in CloudNet to transfer disk and memory, and to maintain network connectivity throughout the migration. We discuss further optimizations to these approaches in Section 6.4.2.

6.4.1.1 Dynamic VPN Connectivity to the Cloud

A straightforward implementation of VM migration between IP networks results in significant network management and configuration complexity [48]. As a result, *virtualizing network connectivity is key in CloudNet for achieving the task of WAN migration seamlessly relative to applications*. However, reconfiguring the VPNs that CloudNet can take advantage of to provide this abstraction has typically taken a long time because of manual (or

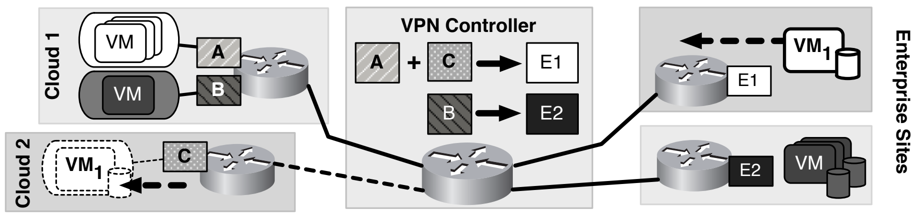


Figure 6.3. The VPN Controller remaps the route targets (A,B,C) advertised by each cloud data center to match the proper enterprise VPN (E1 or E2). To migrate VM_1 to Cloud Site 2, the VPN controller redefines E1's VPN to include route target A and C, then performs the disk and memory migration.

nearly manual) provisioning and configuration. CloudNet explicitly recognizes the need to set up new VPN endpoints quickly, and exploits the capability of BGP route servers [127] to achieve this.

In many cases, the destination data center will already be a part of the customer's virtual cloud pool because other VMs owned by the enterprise are already running there. However, if this is the first VM being moved to the site, then a new VPLS endpoint must be created to extend the VCP into the new data center.

Creating a new VPLS endpoint involves configuration changes on the data center routers, a process that can be readily automated on modern routers [63, 29]. A traditional, but naive, approach would require modifying the router configurations at each site in the VCP so they all advertise and accept the proper *route targets*. A route target is an ID used to determine which endpoints share connectivity. An alternative to adjusting the router configurations directly, is to dynamically adjust the routes advertised by each site within the network itself. CloudNet takes this approach by having data center routers announce their routes to a centralized VPN Controller. The VPN Controller acts as an intelligent route server and is connected via BGP sessions to each of the cloud and enterprise data centers. The controller maintains a ruleset indicating which endpoints should have connectivity; as all route control messages pass through this VPN Controller, it is able to rewrite the route targets in these messages, which in turn control how the tunnels forming each VPLS are created.

Figure 6.3 illustrates an example where VM_1 is to be migrated from enterprise site E1 to Cloud Site 2. The VPN Controller must extend E1's VPLS to include route targets A and C, while Enterprise 2's VPLS only includes route target B. Once the change is made by the VPN Controller, it is propagated to the other endpoints via BGP. This ensures that each customer's resources are isolated within their own private network, providing CloudNet's virtual cloud pool abstraction. Likewise, the VPN Controller is able to set and distribute fine grained access control rules via BGP using technologies such as Flowspec (RFC 5575).

Our approach allows for fast VCP reconfiguration since changes only need to be made at a central location and then propagated via BGP to all other sites. This provides simpler connectivity management compared to making changes individually at each site, and allows a centralized management scheme that can set connectivity and access control rules for all sites.

In our vision for the service, the VPN Controller is operated by the network service provider. As the VPLS network in CloudNet spans both the enterprise sites and cloud data centers, the cloud platform must have a means of communicating with the enterprise's network operator. The cloud platform needs to expose an interface that would inform the network service provider of the ID for the VLAN used within the cloud data center so that it can be connected to the appropriate VPN endpoint. Before the VPN Controller enables the new endpoint, it must authenticate with the cloud provider to ensure that the enterprise customer has authorized the new resources to be added to its VPN. These security details are orthogonal to our main work, and in CloudNet we assume that there is a trusted relationship between the enterprise, the network provider, and the cloud platform.

6.4.1.2 Disk State Migration

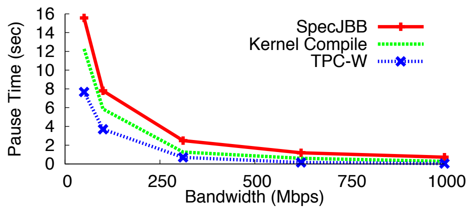
LAN based live migration assumes a shared file system for VM disks, eliminating the need to migrate disk state between hosts. As this may not be true in a WAN environment, CloudNet supports either shared disk state or a replicated system that allows storage to be migrated with the VM.

If we have a "shared nothing" architecture where VM storage must be migrated along with the VM memory state, CloudNet uses the DRBD disk replication system to migrate storage to the destination data center [35]. In Figure 6.3, once connectivity is established to Cloud 2, the replication system must copy VM_1's disk to the remote host, and must continue to synchronize the remote disk with any subsequent writes made at the primary. In order to reduce the performance impact of this synchronization, CloudNet uses DRBD's *asynchronous* replication mode during this stage. Once the remote disk has been brought to a consistent state, CloudNet switches to a *synchronous* replication scheme and the live migration of the VM's memory state is initiated. During the VM migration, disk updates are synchronously propagated to the remote disk to ensure consistency when the memory transfer completes. When the migration completes, the new host's disk becomes the primary, and the origin's disk is disabled.

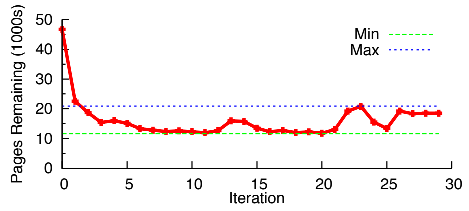
Migrating disk state typically represents the largest component of the overall migration time as the disk state may be in the tens or hundreds of gigabytes. Fortunately, disk replication can be enabled well in advance of a planned migration. Since the disk state for many applications changes only over very long time scales, this can allow the majority of the disk to be transferred with relatively little wasted resources (e.g., network bandwidth). For unplanned migrations such as a cloud burst in response to a flash crowd, storage may need to be migrated on demand. CloudNet's use of asynchronous replication during bulk disk transfer minimizes the impact on application performance.

6.4.1.3 Transferring Memory State

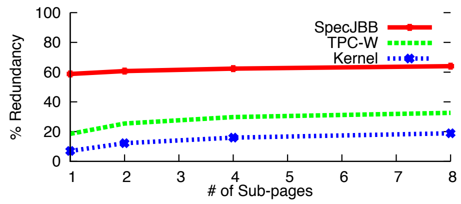
Most VM migration techniques use a "pre-copy" mechanism to iteratively copy the memory contents of a live VM to the destination machine, with only the modified pages being sent during each iteration [30, 91]. At a certain point, the VM is paused to copy the



(a)



(b)



(c)

Figure 6.4. (a) Low bandwidth links can significantly increase the downtime experienced during migration. (b) The number of pages to be sent quickly levels off. Intelligently deciding when to stop a migration eliminates wasteful transfers and can lower pause time. (c) Each application has a different level of redundancy. Using finer granularity finds more redundancy, but has diminishing returns.

final memory state. WAN migration can be accomplished by similar means, but the limited bandwidth and higher latencies can lead to decreased performance–particularly much higher VM downtimes–since the final iteration where the VM is paused can last much longer. CloudNet augments the existing migration code from the Xen virtualization platform with a set of optimizations that improve performance, as described in Section 6.4.2.

The amount of time required to transfer a VM's memory depends on its RAM allocation, working set size and write rate, and available bandwidth. These factors impact both the total time of the migration, and the application-experienced downtime caused by pausing the VM during the final iteration. With WAN migration, it is desirable to minimize both these times as well as the bandwidth costs for transferring data. While pause time may have the most direct impact on application performance, the use of synchronous disk replication throughout the memory migration means that it is also important to minimize the total time to migrate memory state, particularly in high latency environments.

As bandwidth reduces, the total time and pause time incurred by a migration can rise dramatically. Figure 6.4(a) shows the pause time of VMs running several different applications, as the available bandwidth is varied (assumes shared storage and a constant 10 msec round trip latency). Note that performance decreases non-linearly; migrating a VM running the SPECjbb benchmark on a gigabit link incurs a pause time of 0.04 seconds, but rises to 7.7 seconds on a 100 Mbps connection. This nearly 200X increase is unacceptable for most applications, and happens because a migration across a slower link causes each iteration to last longer, increasing the chance that additional pages will be modified and thus need to be resent. This is particularly the case in the final iteration. This result illustrates the importance of optimizing VM migration algorithms to better handle low bandwidth connections.

Migrations over the WAN may also have a greater chance of being disrupted due to network failures between the source and destination hosts. Because the switch-over to the second site is performed only after the migration is complete, CloudNet will suffer no ill effects from this type of failure because the application will continue running on the origin site, unaffected. In contrast, some pull or ”post-copy” based migration approaches start running the application at the destination prior to receiving all data, which could lead to the VM crashing if there is a network disconnection.

6.4.1.4 Maintaining Network Connections

Once disk and memory state have been migrated, CloudNet must ensure that VM_1's active network connections are redirected to Cloud 2. In LAN migration, this is achieved by having the destination host transmit an unsolicited ARP message that advertises the VM's MAC and IP address. This causes the local Ethernet switch to adjust the mapping for the VM's MAC address to its new switch port [30]. Over a WAN, this is not normally a feasible solution because the source and destination are not connected to the same switch. Fortunately, CloudNet's use of VPLS bridges the VLANs at Cloud 2 and E1, causing the

ARP message to be forwarded over the Internet to update the Ethernet switch mappings at both sites. This allows open network connections to be seamlessly redirected to the VM's new location.

In the Xen virtualization platform, this ARP is triggered by the VM itself after the migration has completed. In CloudNet, we optimize this procedure by having the destination host preemptively send the ARP message immediately after the VM is paused for the final iteration of the memory transfer. This can reduce the downtime experienced by the VM by allowing the ARP to propagate through the network in parallel with the data sent during the final iteration. However, on our evaluation platform this does not appear to influence the downtime, although it could be useful with other router hardware since some implementations can cache MAC mappings rather than immediately updating them when an ARP arrives.

6.4.2 Optimizing WAN VM Migration

In this section we propose a set of optimizations to improve the performance of VM migration over the WAN. The changes are made within the virtualization hypervisor; while we use the Xen virtualization platform in our work [30], they would be equally useful for other platforms such as VMWare which uses a similar migration mechanism [91].

6.4.2.1 Smart Stop and Copy

The Xen migration algorithm typically iterates until either a very small number of pages remain to be sent or a limit of 30 iterations is reached. At that point, the VM is paused, and all remaining pages are sent. However, our results indicate that this tends to cause the migration algorithm to run through many unnecessary iterations, increasing both the total time for the migration and the amount of data transferred.

Figure 6.4(b) shows the number of pages remaining to be sent at the end of each iteration during a migration of a VM running a kernel compilation over a link with 622 Mbps bandwidth and 5 msec latency. After the fourth iteration there is no significant drop in the number of pages remaining to be sent. This indicates that (i) a large number of iterations only extends the total migration time and increases the data transferred, and (ii) the migration algorithm could intelligently pick when to stop iterating in order to decrease both total and pause time. For the migration shown, picking the optimal point to stop the migration would reduce pause time by 40% compared to the worst stopping point.

CloudNet uses a *Smart Stop and Copy* optimization to reduce the number of unnecessary iterations and to pick a stopping point that minimizes pause time. Unfortunately, these two goals are potentially conflicting. Stopping after only a few iterations would reduce *total time*, but running for an extra few rounds may result in a lower *pause time*, which can potentially have a larger impact on application performance. The Smart Stop algorithm is designed to balance this trade-off by minimizing pause time without significantly increasing total time.

We note that in most cases (e.g. Figure 6.4(b)), after about five iterations the migration reaches a point of diminishing returns, where in a given iteration, approximately the same amount of data is dirtied as is sent. To detect this point, the first stage of Smart Stop monitors the number of pages sent and dirtied until they become equal. Prior to this point there was a clear gain from going through another iteration because more data was sent than dirtied, lowering the potential pause time.

While it is possible to stop the migration immediately at the point where as many pages are dirtied as sent, we have found that often the random fluctuations in how pages are written to can mean that waiting a few more iterations can result in a lower pause time with only a marginal increase in total time. Based on this observation, Smart Stop switches mode once it detects this crossover, and begins to search for a local minimum in the number of pages remaining to be sent. If at the start of an iteration, the number of pages to be sent is less than any previous iteration in a sliding window, Smart Stop pauses the VM to prevent any more memory writes and sends the final iteration of memory data.

6.4.2.2 Content Based Redundancy

Content based redundancy (CBR) elimination techniques have been used to save bandwidth between network routers [4], and we use a similar approach to eliminate the redundant data while transferring VM memory and disk state.[1] Disks can have large amounts of redundant data caused by either empty blocks or similar files. Likewise, a single virtual machine can often have redundant pages in memory from similar applications or duplicated libraries.

There are a variety of mechanisms that can be used to eliminate redundancy in a network transfer, and a good comparison of techniques is found in [1]. CloudNet can support any type of redundancy elimination algorithm; for efficiency, we use a block based approach that detects identical, fixed size regions in either a memory page or disk block. We have also tested a Rabin Fingerprint based redundancy elimination algorithm, but found it to be slower without substantially improving the redundancy detection rate.

CloudNet's block based CBR approach splits each memory page or disk block into fixed sized blocks and generates hashes based on their content using the Super Fast Hash Algorithm [53]. If a hash matches an entry in fixed size, FIFO caches maintained at the source and destination hosts, then a block with the same contents was sent previously. After verifying the pages match (in case of hash collisions), the migration algorithm can simply send a 32bit index to the cache entry instead of the full block (e.g. 4KB for a full memory page).

Dividing a memory page into smaller blocks allows redundant data to be found with finer granularity. Figure 6.4(c) shows the amount of memory redundancy found in several applications during migrations over a 100 Mbps link as the number of blocks per page was varied. Increasing the number of sub-pages raises the level of redundancy that is found, but

[1]Commercial products such as those from RiverBed Technologies can also perform CBR using a transparent network appliance. Such products may not be suitable in our case since memory and/or disk migration data is likely to use encryption to avoid interception of application state. In such cases, end-host based redundancy elimination has been proposed as an alternative [1]—an approach we use here also.

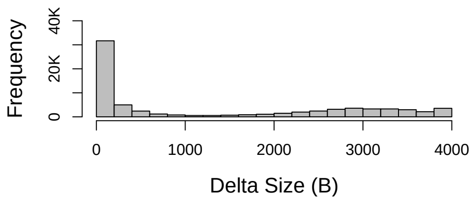
it can incur greater overhead since each block requires a hash table lookup. In CloudNet we divide each page into four sub-pages since this provides a good tradeoff of detection rate versus overhead.

Disk transfers can also contain large amounts of redundant data. Our redundancy elimination code is not yet fully integrated with DRBD, however, we are able to evaluate the potential benefit of this optimization by analyzing disk images with an offline CBR elimination tool.

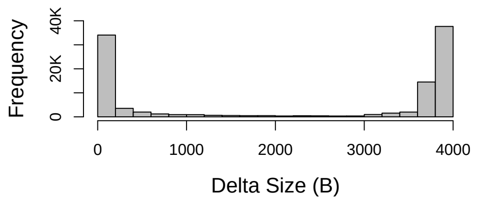
We currently only detect redundancy within a single VM's memory or disk. Previous work has demonstrated that different virtual machines often have some identical pages in memory, e.g. for common system libraries [47, 146]. Likewise, different virtual machines often have large amounts of identical data on disk due to overlap in the operating system and installed applications. Some of this redundancy could be found by using a network based appliance to detect redundancy across the migration traffic of multiple virtual machines. However, a network based approach can only find a redundant disk or memory block if it matches a packet sent during a previous migration. In order to find redundancy in the disks or memories of VMs which are not being moved, such an approach could be complemented with a distributed, content addressable cache run across the hosts at each site [103]. Fortunately, the single VM redundancy detection technique used in CloudNet is still able to save a significant amount of bandwidth without this added complexity.

6.4.2.3 Using Page Deltas

After the first iteration, most of the pages transferred are ones which were sent previously, but have since been modified. Since an application may be modifying only portions of pages, another approach to reduce bandwidth consumption is to keep a cache of previously transmitted pages, and then only send the difference between the cached and current page if it is retransmitted. This technique has been demonstrated in the Remus high availability system to reduce the bandwidth required for VM synchronization [32] in a LAN.



(a) Kernel Compile



(b) TPC-W

Figure 6.5. During a kernel compile, most pages only experience very small modifications. TPC-W has some pages with small modifications, but other pages are almost completely changed.

We enhance this type of communicating deltas in a unique manner by complementing it with our CBR optimization. This combination helps overcome the performance limitations that would otherwise constrain the adoption of WAN migration

We have modified the Xen migration code so that if a page, or sub page block, does not match an entry in the cache using the CBR technique described previously, then the page address is used as a secondary index into the cache. If the page was sent previously, then only the difference between the current version and the stored version of the page is sent. This delta is calculated by XOR'ing the current and cached pages, and run length encoding the result.

Figure 6.5 shows histograms of delta sizes calculated during migrations of two applications. A smaller delta means less data needs to be sent; both applications have a large number of pages with only small modifications, but TPC-W also has a collection of pages that have been completely modified. This result suggests that page deltas can reduce the amount of data to be transferred by sending only the small updates, but that care must be taken to avoid sending deltas of pages which have been heavily modified.

While it is possible to perform some WAN optimizations such as redundancy elimination in network middleboxes [4], the Page Delta optimization relies on memory page address information that can only be obtained from the hypervisor. As a result, we make all of our modifications within the virtualization and storage layers. This requires no extra support from the network infrastructure and allows a single cache to be used for both redun-

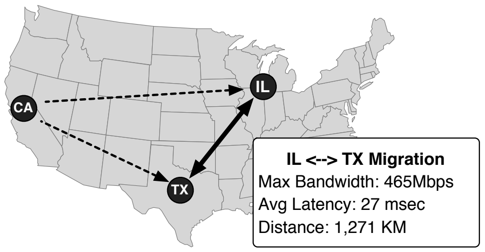


Figure 6.6. Our CloudNet testbed is deployed across three data centers. Migrations are performed between the data centers in IL and TX, with application clients running in CA.

dancy elimination and deltas. Further, VM migrations are typically encrypted to prevent eavesdroppers from learning the memory contents of the VM being migrated, and network level CBR generally does not work over encrypted streams [1]. Finally, we believe our optimization code will be a valuable contribution back to the Xen community.

6.5 Evaluation

This section evaluates the benefits of each of our optimizations and studies the performance of several different application types during migrations between data center sites under a variety of network conditions. We also study migration under the three use case scenarios described in the introduction: Section 6.5.4 illustrates a cloud burst, Section 6.5.8 studies multiple simultaneous migrations as part of a data center consolidation effort, and Section 6.5.9 looks at the cost of disk synchronization in a follow-the-sun scenario.

6.5.1 Testbed Setup

We have evaluated our techniques between three data center sites spread across the United States, and interconnected via an operational network, as well as on a laboratory testbed that uses a network emulator to mimic a WAN environment.

Data Center Prototype: We have deployed CloudNet across three data centers in Illinois, Texas, and California as shown in Figure 6.6. Our prototype is run on top of the ShadowNet infrastructure which is used by CloudNet to configure a set of logical routers located at each site [26]. At each site we have Sun servers with dual quad-core Xeon CPUs and 32GB of RAM. We use Juniper M7i routers to create VPLS connectivity between all sites. We use the California site to run application clients, and migrate VMs between Texas and Illinois. Network characteristics between sites are variable since the data centers are connected via the Internet; we measured an average round trip latency of 27 msec and a max throughput of 465 Mbps between the sites used for migrations.

Lab Testbed: Our lab testbed consists of multiple server/router pairs linked by a VPLS connection. The routers are connected through gigabit ethernet to a PacketSphere Network Emulator capable of adjusting the bandwidth, latency, and packet loss experienced on the link. We use this testbed to evaluate WAN migrations under a variety of controlled network conditions.

6.5.2 Applications and Workloads

Our evaluation studies three types of business applications. We run each application within a Xen VM and allow it to warm up for at least twenty minutes prior to migration.

SPECjbb 2005 is a Java server benchmark that emulates a client/server business application [111]. The majority of the computation performed is for the business logic performed at the application's middle tier. SPECjbb maintains all application data in memory and only minimal disk activity is performed during the benchmark.

Kernel Compile represents a development workload. We compile the Linux 2.6.31 kernel along with all modules. This workload involves moderate disk reads and writes, and memory is mainly used by the page cache. In our simultaneous migration experiment we run a compilation cluster using *distcc* to distribute compilation activities across several VMs that are all migrated together.

TPC-W is a web benchmark that emulates an Amazon.com like retail site [122]. We run TPC-W in a two tier setup using Tomcat 5.5 and MySQL 5.0.45. Both tiers are run within a single VM. Additional servers are used to run the client workload generators,

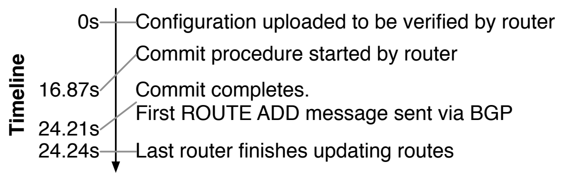

Timeline
0s
Configuration uploaded to be verified by router
Commit procedure started by router
16.87s
Commit completes.
First ROUTE ADD message sent via BGP
24.21s
24.24s
Last router finishes updating routes

Figure 6.7. Timeline of operations to add a new endpoint

emulating 600 simultaneous users accessing the site using the "shopping" workload that performs a mix of read and write operations. The TPC-W benchmark allows us to analyze the client perceived application performance during the migration, as well as verify that active TCP sessions do not reset during the migration.

6.5.3 VPN Endpoint Manipulation

Before a migration can begin, the destination site may need to be added to the customer's VPN. This experiment measures the time required for CloudNet's VPN Controller to add the third data center site to our Internet-based prototype by manipulating route targets. Figure 6.7 shows a timeline of the steps performed by the VPN Controller to reconfigure its intelligent route server. The controller sends a series of configuration commands followed by a commit operation to the router, taking a total of 24.21s to be processed on our Juniper M7i routers; these steps are manufacturer dependent and may vary depending on the hardware. As the intelligent route server does not function as a general purpose router, it would be possible to further optimize this process if reduction in VPN reconfiguration time is required.

Once the new configuration has been applied to the router maintained by the VPN controller, the updated information must be propagated to the other routers in the network. The information is sent in parallel via BGP. On our network where three sites need to have their routes updated, the process completes in only 30 milliseconds, which is just over one round trip time. While propagating routes may take longer in larger networks, the initial intelligent route server configuration steps will still dominate the total cost of the operation.

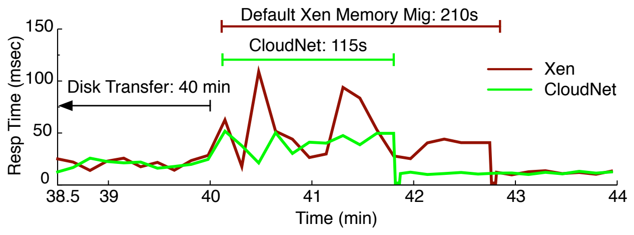


Figure 6.8. Response times rise to an average of 52 msec during the memory migration, but CloudNet shortens this period of reduced performance by 45%. Response time drops to 10msec once the VM reaches its destination and can be granted additional resources.

6.5.4 Cloud Burst: Application Performance

Cloud Bursting allows an enterprise to offload computational jobs from its own data centers into the cloud. Current cloud bursting techniques require applications to be shut down in the local site and then restarted in the cloud; the live WAN migration supported by CloudNet allows applications to be seamlessly moved from an enterprise data center into the cloud.

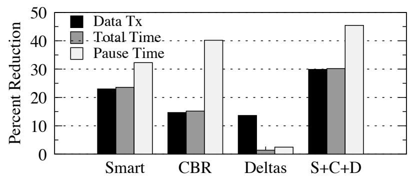
We consider a cloud bursting scenario where a live TPC-W web application must be moved from an overloaded data center in Illinois to one in Texas without disrupting its active clients; we migrate the VM to a more powerful server and increase its processor allocation from one to four cores once it arrives at the new data center location. In a real deployment a single VM migration would not have access to the full capacity of the link between the data centers, so we limit the bandwidth available for the migration to 85Mbps; the VM is allocated 1.7GB of RAM and has a 10GB disk. We assume that CloudNet has already configured the VPN endpoint in Texas as described in the previous section. After this completes, the DRBD subsystem begins the initial bulk transfer of the virtual machine disk using asynchronous replication; we discuss the disk migration performance details in Section 6.5.5 and focus on the application performance during the memory migration here.

The full disk transfer period takes forty minutes and is then followed by the memory migration. Figure 6.8 shows how the response time of the TPC-W web site is affected during the final 1.5 minutes of the storage transfer and during the subsequent memory migration when using both default Xen and CloudNet with all optimizations enabled. During the disk transfer period, the asynchronous replication imposes negligible overhead; average response time is 22 msec compared to 20 msec prior to the transfer. During the VM migration itself, response times become highly variable, and the average rises 2.5X to 52 msec in the default Xen case. This overhead is primarily caused by the switch to synchronous disk replication—any web request which involves a write to the database will see its response time increased by at least the round trip latency (27 msec) incurred during the synchronous write. As a result, it is very important to minimize the length of time for the memory migration in order to reduce this period of lower performance. After the migration completes, the response time drops to an average of 10 msec in both cases due to the increased capacity available for the VM.

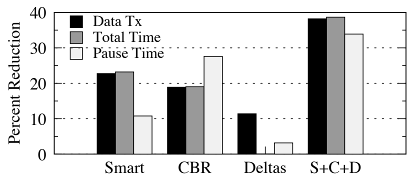
While both default Xen and CloudNet migrations do suffer a performance penalty during the migration, CloudNet's optimizations reduce the memory migration time from 210 to 115 seconds, a 45% reduction. CloudNet also lowers the downtime by half, from 2.2 to 1 second. Throughout the migration, CloudNet's memory and disk optimizations conserve bandwidth. Using a 100MB cache, CloudNet reduces the memory state transfer from 2.2GB to 1.5GB. Further, the seamless network connectivity provided by the CloudNet infrastructure prevents the need for any complicated network reconfiguration, and allows the application to continue communicating with all connected clients throughout the migration. This is a significant improvement compared to current cloud bursting techniques which typically cause lengthy downtime as applications are shutdown, replicated to the second site, and then rebooted in their new location.

6.5.5 Disk Synchronization

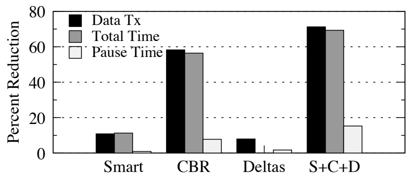
Storage migration can be the dominant cost during a migration in terms of both time and bandwidth consumption. The DRBD system used by CloudNet transfers disk blocks to the migration destination by reading through the source disk at a constant rate (4MB/s) and



(a) Kernel Compile



(b) TPC-W



(c) SPECjbb

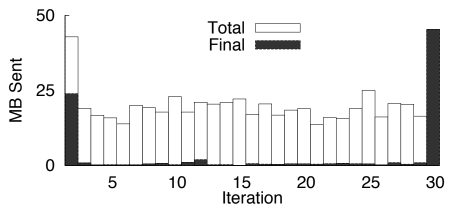
Figure 6.9. CloudNet's optimizations affect different classes of application differently depending on the nature of their memory accesses. Combining all optimizations greatly reduces bandwidth consumption and time for all applications.

	Data Tx (GB)	Total Time (s)	Pause Time (s)
TPC-W	$1.5 \rightarrow 0.9$	$135 \rightarrow 78$	$3.7 \rightarrow 2.3$
Kernel	$1.5 \rightarrow 1.1$	$133 \rightarrow 101$	$5.9 \rightarrow 3.5$
SPECjbb	$1.2 \rightarrow 0.4$	$112 \rightarrow 35$	$7.8 \rightarrow 6.5$

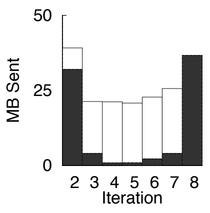
Table 6.1. CloudNet reduces bandwidth, total time, and pause time during migrations over a 100Mbps link with shared disk.

transmitting the non-empty blocks. This means that while the TPC-W application in the previous experiment was allocated a 10GB disk, only 6.6GB of data is transferred during the migration.

The amount of storage data sent during a migration can be further reduced by employing redundancy elimination on the disk blocks being transferred. Using a small 100MB redundancy elimination cache can reduce the transfer to 4.9GB, and a larger 1GB cache can lower the bandwidth consumption to only 3.6GB. Since the transfer rate is limited by the disk read speed, disk migration takes the same amount of time with and without CloudNet's optimizations; however, the use of content based redundancy significantly reduces bandwidth costs during the transfer.



(a) Default Xen



(b) Smart Stop

Figure 6.10. Smart Stop reduces the iterations in a migration, significantly lowering the number of "useless" page transfers that end up needing to be retransmitted in the default case.

6.5.6 Memory Transfer

Here we discuss the benefits provided by each of our optimizations for transferring memory state. To understand each optimization's contribution, we analyze migration performance using VMs allocated 1GB of RAM running each of our three test applications; we create the VMs on a shared storage device and perform the migrations over a 100 Mbps link with 20 msec RTT in our local testbed.

Figure 6.9 shows each of CloudNet's optimizations enabled individually and in combination. We report the average improvement in total time, pause time, and data transferred over four repeated migrations for each optimization. Overall, the combination of all optimizations provides a 30 to 70 percent reduction in the amount of data transferred and total migration time, plus up to a 50% reduction in pause time. Table 6.1 lists the absolute performance of migrations with the default Xen code and with CloudNet's optimizations.

Smart Stop: The Smart Stop optimization can reduce the data transferred and total time by over 20% (Figure 6.9). Using Smart Stop lowers the number of iterations from 30 to an average of 9, 7, and 10 iterations for Kernel Compile, TPC-W, and SPECjbb respectively. By eliminating the unnecessary iterations, Smart Stop saves bandwidth and time.

Smart Stop is most effective for applications which have a large working set in memory. In TPC-W, memory writes are spread across a database, and thus it sees a large benefit from the optimization. In contrast, SPECjbb repeatedly updates a smaller region of memory, and

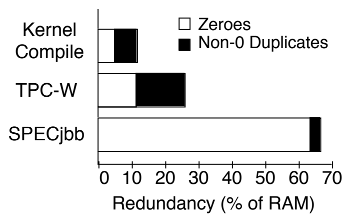


Figure 6.11. Different applications have different levels of redundancy, in some cases mostly from empty zero pages.

these updates occur fast enough that the migration algorithm defers those pages until the final iteration. As a result, only a small number of pages would have been sent during the intermediate iterations that Smart Stop eliminates.

Figure 6.10 shows the total number of pages sent in each iteration, as well as how much of the data is *final*–meaning it does not need to be retransmitted in a later iteration–during a TPC-W migration. After the second iteration, TPC-W sends over 20MB per iteration, but only a small fraction of the total data sent is final–the rest is resent in later iterations when pages are modified again. Smart Stop eliminates these long and unnecessary iterations to reduce the total data sent and migration time.

Smart Stop is also able to reduce the pause time of the kernel compile by over 30% (Figure 6.9(a)). This is because the compilation exhibits a variance in the rate at which memory is modified (Figure 6.4(b)). The algorithm is thus able to pick a more intelligent iteration to conclude the migration, minimizing the amount of data that needs to be sent in the final iteration.

Redundancy Elimination: Figure 6.11 shows the amount of memory redundancy found in each applications during migrations over a 100 Mbps link when each memory page is split into four blocks. SPECjbb exhibits the largest level of redundancy; however, the majority of the redundant data is from empty “zero” pages. In contrast, a kernel compilation has about 13% redundancy, of which less than half is zero pages. The CBR optimization eliminates this redundancy, providing substantial reductions in the total data transferred and migration time (Figure 6.9). Since CBR can eliminate redundancy in por-

	Data Transfer (MB)		Page Delta
	Iter 1	Iters 2-30	Savings (MB)
TPC-W	954	315	172
Kernel	877	394	187
SPECjbb	932	163	127

Table 6.2. The Page Delta optimization cannot be used during the first iteration, but it provides substantial savings during the remaining rounds.

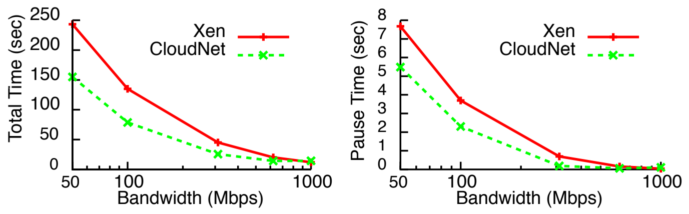
tions of a page, it also can significantly lower the pause time since pages sent in the final iteration often have only small modifications, allowing the remainder of the page to match the CBR cache. This particularly helps the kernel compile and TPC-W migrations which see a 40 and 26 percent reduction in pause time respectively. SPECjbb does not see a large pause time reduction because most of the redundancy in its memory is in unused zero pages which are almost all transferred during the migration's first iteration.

Page Deltas: The first iteration of a migration makes up a large portion of the total data sent since during this iteration the majority of a VM's memory–containing less frequently touched pages–is transferred. Since the Page Delta optimization relies on detecting memory addresses that have already been sent, it can only be used from the second iteration onward, and thus provides a smaller overall benefit, as seen in Figure 6.9.

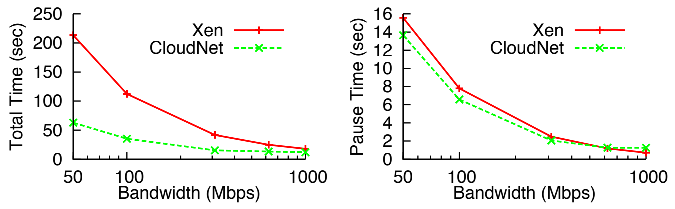
Table 6.2 shows the amount of memory data transferred during the first and remaining iterations during migrations of each application. While the majority of data is sent in the first round, during iterations 2 to 30 the Page Delta optimization still significantly reduces the amount of data that needs to be sent. For example, TPC-W sees a reduction from 487MB to 315MB, a 36 percent improvement.

Currently, the Page Delta optimization does not reduce migration time as much as it reduces data transferred due to inefficiencies in the code. With further optimization, the Page Delta technique could save both bandwidth and time.

Results Summary: The combination of all optimizations improves the migration performance more than any single technique. While the Page Delta technique only comes into



(a) TPC-W



(b) SpecJBB

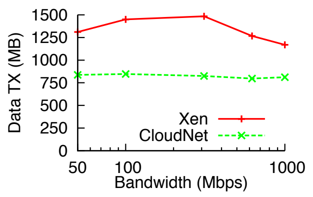
Figure 6.12. Decreased bandwidth has a large impact on migration time, but CloudNet's optimizations reduce the effects in low bandwidth scenarios.

effect after the first iteration, it can provide significant reductions in the amount of data sent during the remainder of the migration. The CBR based approach, however, can substantially reduce the time of the first iteration during which many empty or mostly empty pages are transferred. Finally, Smart Stop eliminates many unnecessary iterations and combines with both the CBR and Page Delta techniques to minimize the pause time during the final iteration.

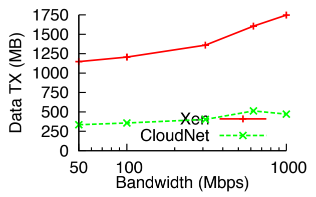
6.5.7 Impact of Network Conditions

We next use the network emulator testbed to evaluate the impact of latency and bandwidth on migration performance.

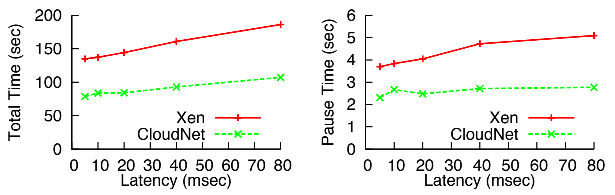
Bandwidth: Many data centers are now connected by gigabit links. However, this is shared by thousands of servers, so the bandwidth that can be dedicated to the migration of a single application is much lower. In this experiment we evaluate the impact of bandwidth on migrations when using a shared storage system. We vary the link bandwidth from 50 to 1000 Mbps, and maintain a constant 10 msec round trip delay between sites.



(a) TPC-W Bandwidth Usage



(b) SPECjbb Bandwidth Usage



(c) TPC-W Latency Impact

Figure 6.13. (a-b) CloudNet's optimizations significantly reduce bandwidth consumption. (c) Increased latency has only a minor impact on the migration process, but may impact application performance due to synchronous disk replication.

Figure 6.12 compares the performance of default Xen to CloudNet's optimized migration system. We present data for TPC-W and SPECjbb; the kernel compile performs similar to TPC-W. Decreased bandwidth increases migration time for both applications, but our optimizations provide significant benefits, particularly in low bandwidth scenarios. CloudNet also substantially reduces the amount of data that needs to be transferred during the migration because of redundancy elimination, page delta optimization and the lower number of iterations, as seen in Figure 6.13(a-b).

CloudNet's code presently does not operate at linespeed when the transfer rate is very high (e.g. about 1Gbps or higher *per VM transfer*). Thus in high bandwidth scenarios, CloudNet reduces the data transferred, but does not significantly affect the total migration or pause time compared to default Xen. We expect that further optimizing the CloudNet code will improve performance in these areas, allowing the optimizations to benefit even LAN migrations.

Latency: Latency between distant data centers is inevitable due to speed of light delays. This experiment tests how latency impacts migration performance as we adjust the delay introduced by the network emulator over a 100Mbps link. Even with TCP settings optimized for WAN environments, slow start causes performance to decrease some as latency rises. CloudNet's optimizations still provide a consistent improvement regardless of link latency as shown in Figure 6.13(c). While latency has only a minor impact on total migration and pause time, it can degrade application performance due to the synchronous disk replication required during the VM migration. Fortunately, CloudNet's optimizations can significantly reduce this period of lowered performance.

Results Summary: CloudNet's optimized migrations perform well even in low bandwidth (50 to 100Mbps) and high latency scenarios, requiring substantially less data to be transferred and reducing migration times compared to default Xen. In contrast to commercial products that require 622 Mbps per VM transfer, CloudNet enables efficient VM migrations in much lower bandwidth and higher latency scenarios.

6.5.8 Consolidation: Simultaneous Migrations

We next mimic an enterprise consolidation where four VMs running a distributed development environment must be transitioned from the data center in Texas to the data center in Illinois. Each of the VMs has a 10GB disk (of which 6GB is in use) and is allocated 1.7GB of RAM and one CPU, similar to a "small" VM instance on Amazon EC2[2]. The load on the cluster is created by repeatedly running a distributed kernel compilation across the four VMs. The maximum bandwidth available between the two sites was measured as 465Mbps with a 27 msec round trip latency; note that bandwidth must be *shared* by the four simultaneous migrations.

We first run a baseline experiment using the default DRBD and Xen systems. During the disk synchronization period a total of 24.1 GB of data is sent after skipping the empty disk blocks. The disk transfers take a total of 36 minutes. We then run the VM memory

[2]Small EC2 instances have a single CPU, 1.7GB RAM, a 10GB root disk, plus an additional 150GB disk. Transferring this larger disk would increase the storage migration time, but could typically be scheduled well in advance.

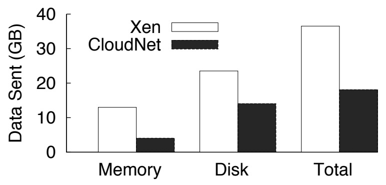


Figure 6.14. CloudNet saves nearly 20GB of bandwidth when simultaneously migrating four VMs.

migrations using the default Xen code, incurring an additional 245 second delay as the four VMs are transferred.

Next, we repeat this experiment using CloudNet's optimized migration code and a 1GB CBR cache for the disk transfer. Our optimizations reduce the memory migration time to only 87 seconds, and halves the average pause time from 6.1 to 3.1 seconds. Figure 6.14 compares the bandwidth consumption of each approach. CloudNet reduces the data sent during the disk transfers by 10GB and lowers the memory migrations from 13GB to 4GB. In total, the data transferred to move the memory and storage for all four VMs falls from 37.4GB in the default Xen case to 18.5GB when using CloudNet's optimizations.

Results Summary: CloudNet's optimizations reduce pause time by a factor of 2, and lower memory migration time–when application performance is impacted most–by nearly 3X. The combination of eliminating redundant memory state and disk blocks can reduce the total data transferred during the migration by over 50%, saving nearly 20GB worth of network transfers.

6.5.9 Follow-the-Sun: Disk Synchronization

In a follow-the-sun scenario, one or more applications are moved between geographic locations in order to be co-located with the workforce currently using the application. In this experiment we consider moving an application with a large amount of state back and forth between two locations. We focus on the disk migration cost and demonstrate the benefits of using incremental state updates when moving back to a location which already has a snapshot from the previous day.

We use the TPC-W web application, but configure it with a much larger 45GB database. The initial migration of this disk takes 3.6 hours and transfers 51GB of data to move the database and root operating system partitions. We then run a TCP-W workload which lasts for 12 hours to represent a full workday at the site. After the workload finishes, we migrate the application back to its original site. In this case, only 723MB of storage data needs to be transferred since the snapshot from the previous day is used as a base image. This reduces the migration time to under five minutes, and the disk and memory migrations can be performed transparently while workers from either site are accessing the application. This illustrates that many applications with large state sizes typically only modify relatively small portions of their data over the course of a day. Using live migration and incremental snapshots allows applications to be seamlessly moved from site to site for relatively little cost and only minimal downtime.

6.6 Related Work

Cloud Computing: Armbrust et al provide a thorough overview of the challenges and opportunities in cloud computing [7]. There are several types of cloud platforms, but we focus on Infrastructure as a Service (IaaS) platforms which rent virtual machine and storage resources to customers. InterCloud explores the potential for federated cloud platforms to provide highly scalable services [22]; CloudNet seeks to build a similar environment and uses WAN migration to move resources between clouds and businesses.

Private Clouds & Virtual Networks: The VIOLIN and Virtuoso projects use overlay networks to create private groups of VMs across multiple grid computing sites [105, 118]. VIOLIN also supports WAN migrations over well provisioned links, but does not have a mechanism for migrating disk state. Overlay network approaches require additional software to be run on each host to create network tunnels. CloudNet places this responsibility on the routers at each site, reducing the configuration required on end hosts.

Our vision for Virtual Private Clouds was initially proposed in [139]. Subsequently, Amazon EC2 launched a new service also called "Virtual Private Clouds" which similarly uses VPNs to securely link enterprise and cloud resources. However, Amazon uses IPSec based VPNs that operate at layer-3 by creating software tunnels between end hosts or IPSec routers. In contrast, CloudNet focuses on VPNs provided by a network operator. Network based VPNs are typically realized and enabled by multiprotocol label switching (MPLS) provider networks, following the "hose model" [36] and are commonly used by enterprises. Provider based VPNs can be either layer-3 VPNs following RFC 2547, or layer-2 virtual private LAN Service (VPLS) VPNs according to RFC 4761. CloudNet relies on network based VPLS as it simplifies WAN migration, has lower overheads, and can provide additional functionality from the network provider, such as resource reservation.

LAN Migration: Live migration is essentially transparent to any applications running inside the VM, and is supported by most major virtualization platforms [91, 30, 55]. Work has been done to optimize migration within the LAN by exploiting fast interconnects that support remote memory access technology [54]. Jin et al. have proposed using memory compression algorithms to optimize migrations [61]. Breitgand et al. have developed a model based approach to determine when to stop iterating during a memory migration [18], similar to Smart Stop. Their approach can allow them to more precisely predict the best time to stop, but it requires knowledge of the VM's memory behavior, and it is not clear how the model would perform if this behavior changes over time. CloudNet's CBR and Page Delta optimizations are simple forms of compression, and more advanced compression techniques could provide further benefits in low bandwidth WAN scenarios, although at the expense of increased CPU overhead. The Remus project uses a constantly running version of Xen's live migration code to build an asynchronous high availability system [32]. Remus obtains a large benefit from an optimization similar to CloudNet's Page Delta technique because it runs a form of continuous migration where pages see only small updates between iterations.

WAN Migration: VMware has announced limited support for WAN migration, but only under very constrained conditions: 622 MBps link bandwidth and less than 5 msec network delay [128]. CloudNet seeks to lower these requirements so that WAN migration can become an efficient tool for dynamic provisioning of resources across data centers. Past research investigating migration of VMs over the WAN has focused on either storage or network concerns. Bradford et al. describe a WAN migration system focusing on efficiently synchronizing disk state during the migration; they modify the Xen block driver to support storage migration, and can throttle VM disk accesses if writes are occurring faster than what the network supports [17]. Shrinker uses content based addressing to detect redundancy across *multiple* hosts at the destination site during VM migrations [103]. This could allow it to reduce bandwidth costs compared to CloudNet, but exposes it to security concerns due to hash collisions, although the likelihood of this can be bounded. The VM Turntable Demonstrator showed a VM migration over intercontinental distances with latencies of nearly 200 msec; they utilize gigabit lightpath links, and like us, find that the increased latency has less impact on performance than bandwidth [123]. Harney et al. propose the use of Mobile IPv6 to reroute packets to the VM after it is moved to a new destination [49]; this provides the benefit of supporting layer-3 connections between the VM and clients, but the authors report a minimum downtime of several seconds due to the Mobile IP switchover, and the downtime increases further with network latency. In this work, we leverage existing mechanisms to simplify storage migration and network reconfiguration, and propose a set of optimizations to reduce the cost of migrations in low bandwidth and high latency environments.

6.7 CloudNet Conclusions

The scale of cloud computing is growing as business applications are increasingly being deployed across multiple global data centers. This chapter presented CloudNet, a prototype cloud computing platform that coordinates with the underlying network provider to create seamless connectivity between enterprise and data center sites, as well as supporting live WAN migration of virtual machines. CloudNet supports a holistic view of WAN migration that handles persistent storage, network connections, and memory state with minimal downtime even in low bandwidth, high latency settings.

While existing migration techniques can wastefully send empty or redundant memory pages and disk blocks, CloudNet is optimized to minimize the amount of data transferred and lowers both total migration time and application-experienced downtime. Reducing this downtime is critical for preventing application disruptions during WAN migrations. CloudNet's use of both asynchronous and synchronous disk replication further minimizes the impact of WAN latency on application performance during migrations. We have demonstrated CloudNet's performance on both a prototype deployed across three data centers separated by over 1,200km and a local testbed. During simultaneous migrations of four VMs between operational data centers, CloudNet's optimizations reduced memory transfer time by 65%, and saved 20GB in bandwidth for storage and memory migration. CloudNet simplifies the initial deployment of applications by enabling seamless migration between data centers, and we believe that WAN migration may become an important resource management technique for handling resources spread across multiple data centers.

CHAPTER 7
HIGH PERFORMANCE, NO DATA LOSS DISASTER RECOVERY

One of the potential uses of the WAN VM migration techniques proposed in the previous chapter is for moving applications between data centers in anticipation of planned data center downtime. However, there is not always sufficient advance notice to move applications between sites, particularly in the event of unexpected power outages or natural disasters. In these cases, businesses must rely on disaster recovery services which continuously replicate application data to a secondary location. In this chapter we discuss how virtualized cloud data centers can be an ideal platform for disaster recovery services form an economic perspective, but that the high latency between cloud data centers can significantly reduce performance if synchronous replication is required. We propose a new replication technique that uses speculative execution to provide synchronous consistency guarantees but performance on par with asynchronous approaches.

7.1 Background and Motivation

Businesses and government enterprises utilize Disaster Recovery (DR) systems to minimize data loss as well as the downtime incurred by catastrophic system failures. Current DR mechanisms range from periodic tape backups that are trucked offsite, to continuous synchronous replication of data between geographically separated sites. Typical DR solutions incur high infrastructure costs since they require a secondary data center for each primary site as well as high bandwidth links to secondary sites. Further, the "larger" the potential impact of a disaster, the greater is the need for geographic separation between the primary and the secondary replica site. The resulting wide area latency can, however, place a large burden on performance.

The recent emergence of commercial cloud computing has made cloud data centers an attractive option for implementing cost-effective DR due to their "resource-on-demand" model and high degree of automation [142]. During normal operation, the cost of providing DR in the cloud can be minimized, and additional resources only need to be brought online—and paid for—when a disaster actually occurs. In addition, the cloud platform's ability to rapidly activate resources on-demand helps minimize the recovery cost after a disaster. The automation that is designed into accessing cloud services enables the DR service to support *Business Continuity*, with substantially lower recovery times.

Despite these attractive economics, a major barrier to using cloud data centers for DR is their large geographic separation from primary sites—the increased latency in communicating with distant cloud sites can become a major performance bottleneck. This is amplified by the limited control cloud users have over the actual placement of their cloud resources. Consequently, a synchronous replication scheme will expose every data write to the performance impact of this wide-area latency, forcing system administrators to seek alternative solutions. Often, such alternatives trade off loss of data for performance by using asynchronous replication, in which a consistent "snapshot" is replicated to the backup site. Asynchronous replication improves performance since the primary site can proceed without waiting for the replication to complete. However, disk writes at the primary site subsequent to the last replicated snapshot will be lost in case of a disaster. Consequently, to implement cloud-based DR for mission-critical business applications, we must design a mechanism that combines the performance benefits of *asynchronous replication* with the no-data-loss consistency guarantee of *synchronous replication*.

In this chapter we propose *Pipelined Synchronous Replication* as an approach to provide high performance disaster recovery services over WAN links connecting enterprises and cloud platforms. Pipelined synchrony targets client-server style applications, and exploits the fundamental observation that an external client is only concerned with a guarantee that data writes related to its requests are committed to storage (both at the primary and the secondary) before a response is received from the system. Because it can potentially take a large amount of time for the secondary site to receive the write, commit and acknowledge, there is a substantial opportunity to overlap processing during this time interval which synchronous approaches ignore. The opportunity is even more compelling when we observe that multi-tier applications can take advantage of pipelined synchronous replication by overlapping remote replication with complex processing across the multiple tiers that are typical in such environments. The key challenge in designing pipelined synchrony is to efficiently track all writes triggered by processing of a request as it trickles through the (multi-tier) system, and to inform the external entity (i.e., a client) only when all these writes have been made "durable". We achieve this by holding up network packets destined for the client until all disk writes that occurred concurrently with request processing have been acknowledged by the backup. This approach imposes causality (i.e., via Lamport's happened-before relation) across externally-bound network packets and disk writes, providing the same consistency guarantee as if the disk writes had been performed synchronously. Since we seek to implement pipelined synchrony in an Infrastructure-as-a-Service (IaaS) cloud environment that relies on Virtual Machines (VMs) to encapsulate user applications, an additional challenge is to employ black-box techniques when providing cloud-based DR.

Our pipelined synchronous replication-based disaster recovery system, *PipeCloud*, effectively exploits cloud resources for a cost-effective disaster recovery service. PipeCloud makes the following contributions: (i) a replication system that offers clients synchronous consistency guarantees at much lower performance cost by pipelining request processing and write propagation; (ii) a communication based synchronization scheme that allows the state of distributed or multi-tier applications to be replicated in a consistent manner; and

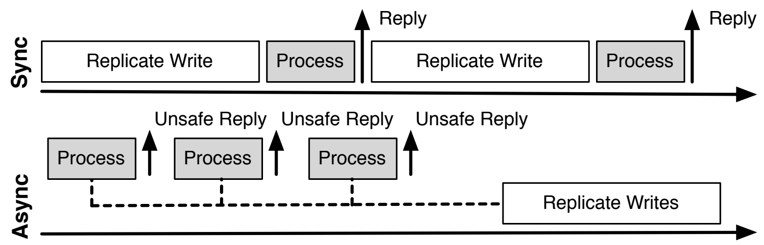
(iii) a black-box implementation that efficiently protects the disks of virtual machines without any modifications to the running applications or operating system.

Our results illustrate the significant performance benefits of using pipelined synchrony for disaster recovery. PipeCloud substantially lowers response time and increases the throughput of a database by twelve times compared to a synchronous approach. When protecting the TPC-W E-commerce web application with a secondary replica 50ms away, PipeCloud reduces the percentage of requests violating a one second SLA from 30% to 3%, and provides throughput equivalent to an asynchronous approach. We demonstrate that PipeCloud offers the same consistency to clients as synchronous replication when disasters strike, and evaluate the potential of using cloud services such as EC2 as a backup site.

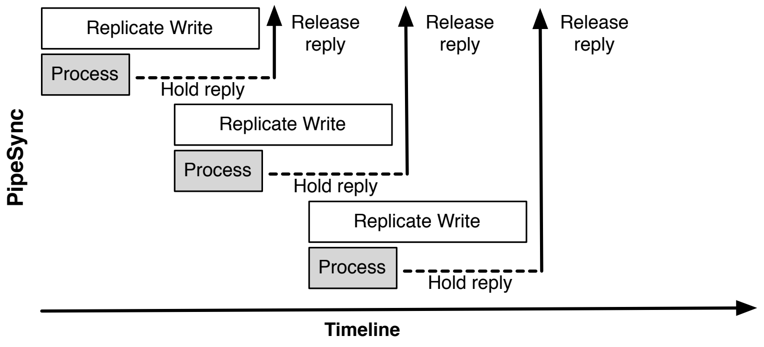
7.2 Disaster Recovery Challenges

The two key metrics that determine the capabilities of a Disaster Recovery (DR) system are *Recovery Point Objective* (RPO) and *Recovery Time Objective* (RTO). The former refers to the acceptable amount of application data that can be lost due to a disaster: a zero RPO means no data can be lost. RTO refers to the amount of downtime that is permissible before the system recovers. A zero RTO means that failover must be instantaneous and transparent and is typically implemented using hot standby replicas.

In scenarios where a small downtime is tolerable (i.e., RTO>0), the cost of DR can be reduced substantially by eliminating hot standbys. The limiting factors in optimizing the RTO in this case depend on engineering considerations: how swiftly can we provision hardware resources at the backup site to recreate the application environment? Once resources have been provisioned, what is the bootstrapping latency for the application software environment? Since disasters happen mid-execution, is a recovery procedure such as a file system check necessary to ensure that the preserved application data is in a usable state? As indicated earlier, leveraging cloud automation can significantly improve the RTO metric;



(a) Existing replication strategies



(b) Pipelined Synchronous Replication

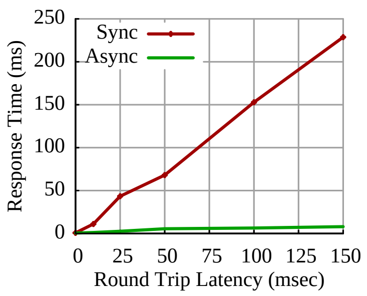
Figure 7.1. Replication strategies. (a) Existing approaches: Synchronous and Asynchronous replication. (b) Pipelined Synchronous Replication.

we have also shown that the cloud's economics driven by on-demand resource utilization are a natural fit for substantially lowering the cost of DR deployments [142].

Next, in Section 7.2.1, we discuss the impact of latency between the primary and secondary sites on the RPO that can be achieved with different replication strategies. In Section 7.2.2 we introduce a broader RPO definition which takes into consideration the client's view of a DR system. Finally, in Section 7.2.3 we describe the specific DR operational assumptions and system model considered in our work.

7.2.1 Replication Strategies and Latency

When preserving stored data to a secondary location, the network round trip time (RTT) between the primary and secondary locations significantly impacts the choice of replication algorithm, and thus, the ability to provide an RPO as close as possible to zero (i.e., no data loss). Latency considerations lead to a choice between two primary modes of data



(a)

Location	RTT (ms)	Distance (km)
Virginia	16	567
California	106	4,250
Ireland	92	4,917
Singapore	281	15,146

(b)

Figure 7.2. (a) Latency significantly reduces performance for synchronous replication. Table (b) lists round trip latency and approximate distance from UMass to Amazon data centers.

replication for disaster survivability: synchronous and asynchronous replication [60]. With synchronous (sync) replication, no data write is reported as complete until it has succeeded in both the primary and secondary sites. With asynchronous (async) replication, writes only need to succeed locally for the application to make progress, and they will be trickled back opportunistically to the secondary replica. The timelines in Figure 7.1 (a) illustrate the behavior of sync and async replication.

With sync replication, applications obtain an RPO of zero by construction: no application progress is permitted until data has been persisted remotely. However, a higher latency results in corresponding increase in the response time and lower throughput for client-server type applications. Figure 7.2 (a) shows the performance impact of increasing the latency between the primary and backup sites. For this experiment, we used DRBD [35], a standard block device replication tool which supports both sync and async modes, to protect a MySQL database. The response time of performing inserts into the database increases linearly with the RTT. Even for relatively short distances, e.g., from Massachusetts to the EC2 data center in Virginia (16msec, Figure 7.2 (b)), the response time degrades noticeably. For these reasons, while mission-critical applications need to resort to synchronous replication, they incur a high performance cost. To mitigate this overhead, the secondary site is often chosen to be geographically close (within tens of km) to the primary. However, replication to nearby facilities is unlikely to withstand many kinds of disasters that have struck infrastructure in recent memory: hurricanes, earthquakes and tsunamis, and regional energy blackouts. Another way to mitigate application overhead while maintaining a zero RPO is to commit significant financial resources to have a dedicated infrastructure. This may allow the use of chained topologies, that progressively separate replicated data from the primary site [38]. We emphasize that these deployments are far removed from the capabilities of most users.

Async replication sacrifices RPO guarantees, as illustrated with the "unsafe replies" of the second timeline in Figure 7.1 (a). However, asynchrony also decouples application performance from data preservation. The area of asynchronous replication has thus been a fertile ground for optimization research [66, 60] that explores the tradeoffs between replication frequency, application RPO demands, financial outlay by application owners, and possibly even multi-site replication as outlined above. Fundamentally, however, async replication exposes applications to a risk of inconsistency: clients may be notified of a request having completed even though it has not yet been preserved at the backup and may be lost in the case of a disaster. Further, the number of these "unsafe replies" increases as latency rises since the backup lags farther behind the primary.

7.2.2 Client RPO Guarantees

To better illustrate the impact of replication algorithm on application consistency, we formalize here our notion of *Client* RPO guarantees. We define three views of the application state: the primary site view (PV), the secondary site view (SV), and the external clients view (CV), and we illustrate their relationship in Figure 7.3.

In both synchronous and asynchronous replication, the SV is a subset of the PV: the SV lags in time behind the PV, and reflects a past state of the application data. For asynchronous replication the delta between SV and PV can be arbitrarily large. For synchronous replication, the delta is at most one write (or one logical write, if we consider a set of scatter-

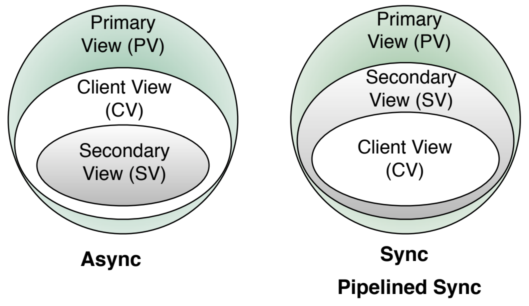


Figure 7.3. Relationship between application views under different replication algorithms. In particular, SV and CV relations define the RPO guarantees upon disaster.

gather DMA writes issued concurrently as one single logical packet): an application cannot make further progress until that write is made durable at the secondary. In all cases, the CV is also a subset of the PV, since the primary performs local processing before updating clients.

The key difference resides in the delta between CV and SV. In asynchronous replication, clients are acknowledged before writes are made durable, and thus the SV is a subset of the CV, reflecting the non-zero RPO. In synchronous replication, clients are acknowledged only after writes have been persisted remotely, and thus the CV is a subset of the SV. As shown in Figure 7.3, Pipelined Synchrony, which we present in Section 7.3, maintains the same client RPO guarantees as synchronous replication.

7.2.3 System Model

Our work assumes an enterprise primary site that is a modern virtualized data center and a secondary site that is a cloud data center; the cloud site is assumed to support Infrastructure-as-a-Service (IaaS) deployments through the use of system virtualization. The primary site is assumed to run multiple applications, each inside virtual machines. Applications may be distributed across multiple VMs, and one or more of these VMs may write data to disks that require DR protection. Data on any disk requiring DR protection is assumed to be replicated to the secondary site while ensuring the same client RPO guarantees as sync replication. We assume that a small, non-zero RTO can be tolerated by the

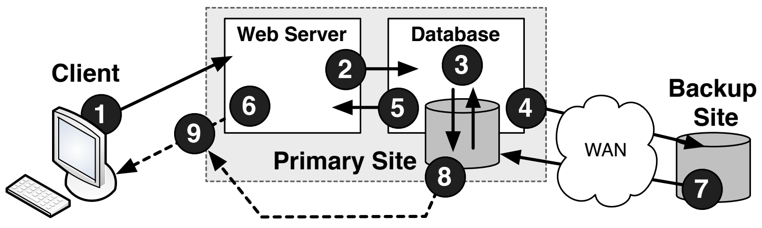


Figure 7.4. Replication in a simple multi-tiered application. Number sequence corresponds to Pipelined Synchronous Replication.

application, allowing us to leverage cloud automation services to dynamically start application VMs after a disaster. We further assume that application VMs are black-boxes and that we are unable to require specific software or source code changes for the purposes of disaster recovery. While this makes our techniques broadly applicable and application-agnostic, the price of this choice is the limited "black-box" visibility into the application that can be afforded at the VM level. Our mechanisms are, however, general, and could be implemented at the application level.

Despite adhering to a black-box methodology, we require applications to be well-behaved with respect to durability. For example, we cannot support a MySQL database configured with the MyISAM backend, which does not support transactions (and durability), and replies to clients while writes may still be cached in memory. More broadly, applications should first issue a write to storage, ensure such write has been flushed from memory to the disk controller, and only then reply to a client the result of an operation. Synchronous replication cannot guarantee adequate data protection without this assumption, and neither can our approach.

7.3 Pipelined Synchrony

Given the WAN latencies between the primary and a secondary cloud site, we seek to design a replication mechanism, as part of an overall DR solution, that combines the performance benefits of async replication with the consistency guarantees of sync replication.

To do so, we must somehow leverage the overlapping of replication and processing that is typical of asynchrony, while retaining the inherent safety of synchronous approaches.

We make two primary observations. First, from the perspective of an external client it does not matter if the transmission of the write and the processing overlap, as long as the client is guaranteed that the data writes are durably committed to the backup before it receives a reply. Second, the potential for performance improvements compared to synchronous replication is substantial when there is a large delay to overlap, as is the case of DR systems with high WAN latencies. The case becomes stronger for multi-tiered applications and, more generally, clustered applications or distributed systems interfacing with external clients via some form of frontend. These applications often require complex processing across multiple tiers or components. We apply these observations to realize a technique called *pipelined synchronous replication*.

7.3.1 Pipelined Synchronous Replication

Pipelined synchronous replication is defined as blocking on an externally visible event until all writes resulting from the (distributed) computation that generated the external event have been committed to disk at the primary and the secondary. When processing a request, pipelined synchronous replication allows overlapping of computation and remote writes—i.e., writes to the secondary are asynchronous and pipelined with the remote writes, allowing subsequent processing to proceed. Upon generating an externally visible event (such as a network packet or a reply), however, the event must be blocked, and not released to the client until all pending writes have finished. In essence, our approach mitigates the performance penalties associated with speed-of-light delays by overlapping or pipelining computation and remote writes, like in async replication, while ensuring the same relation between client view and secondary view as synchronous replication. Figure 7.1 (b) depicts an illustrated timeline of the pipelined synchronous approach.

To contrast pipelined synchronous replication with existing replication strategies, consider the illustrative example in Figure 7.4. Here a client, Alice, goes through the process of buying a ticket from a travel website, by submitting her credit card information in step 1. As is common, Alice interacts with a frontend web server which may perform some processing before forwarding the request on to a backend database (step 2) to record her purchase. In step 3, the DB writes the transaction to disk. Since this is critical state of the application, in step 4 the disk write is also replicated across a WAN link to the backup site, to be preserved. Here the system behavior depends on the type of replication used. With sync replication, the system would wait for the replication to complete (i.e., for the acknowledgement from the remote site in step 7), before continuing processing (step 5) and responding to the client (step 6). With async replication the system would immediately continue with steps 5 and 6 after the DB write has succeeded locally, deferring the replication in step 4 for later. In contrast, with pipelined synchronous replication, the transfer in step 4 is performed immediately yet asynchronously, allowing the database to return its reply to the front tier server in step 5 concurrently. The front tier continues processing the request, for example combining the ticket information with a mashup of maps and hotel availability. Eventually, in step 6, the web tier produces a reply to return to Alice. In the pipelined synchrony case, this reply cannot be returned to the client until the database write it was based on has been persisted to the remote site. Only after step 7 completes and the remote server has acknowledged the write as complete can the reply to the client be released (step 8) and returned to Alice's web browser to show her the purchase confirmation (step 9).

The use of pipelined synchrony means that steps 5 and 6, which may include significant computation cost, can be performed in parallel with the propagation of the disk write to the backup site. This can provide a substantial performance gain compared to synchronous replication which must delay this processing for the length of a network round trip. Since

Pipelined Synchrony defers replying to Alice until after the write is acknowledged in step 7, she is guaranteed that the data her reply is based on has been durably committed.

Thus the key challenge is to track which *durable* write requests, i.e., those that need to be persisted to the secondary site, are causally related (dependent) on which *externally-visible* network packets. In other words, Pipelined Synchrony replication must guarantee a causal ordering between externally-visible and durable events as defined by Lamport's *happened-before* $\rightarrow$ relation [74]: if any write request $\rightarrow$ a network packet, then the write must complete before the packet is released.

To intuitively understand how such dependencies can be tracked, first assume a global clock in the system. Further assume that every write is timestamped using this global clock. In this case, a total ordering of all events is obtained. Hence, if a network packet is generated at time t, then it is sufficient to hold this packet until *all* disk writes that have a timestamp $\leq t$ have finished at the secondary site. Observe that not all of these writes are causally related to the network packet; however, by waiting for all previously issued writes to complete at the secondary, we ensure that all causally related writes will also finish, thereby ensuring safety.

In practice, a multi-tier application does not have the luxury of a global clock and techniques such as Lamport's logical clocks only yield a partial, rather than a total, ordering of events. Thus we must devise a scheme to perform this timestamping, using logical clocks, so as to identify and track causally dependent writes in multi-tier distributed applications. The problem is further complicated by the fact that our approach provides *black box* protection of VMs.

7.3.2 PipeCloud

In this section we present the design of PipeCloud, our pipelined-synchronous disaster recovery engine, while in Section 7.4 we elaborate on the implementation details. As explained above, in a single VM application, the local machine clock may be used as a

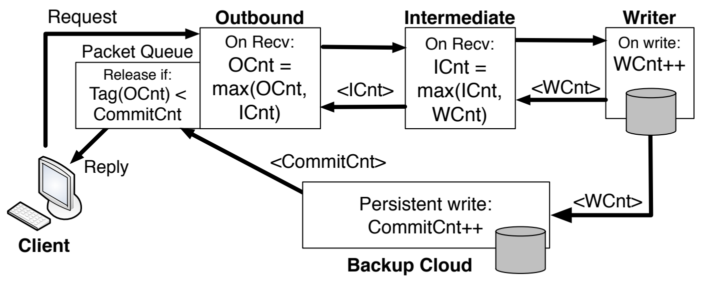


Figure 7.5. Multi-Tier pipelined replication. The reply to the client is buffered until the dependent write has been committed.

global clock, allowing a simple approach for tracking writes that are causally dependent on a network packet. This can be easily extended to support multi-tier applications with only a single protected writer, but becomes more complex for multi-tier multi-writer applications. We first explain the case of providing DR protection to a multi-tier application with a single writer in Section 7.3.2.1, and then generalize to the multi-tier, multi-writer case in Section 7.3.2.2.

7.3.2.1 Single Writer Protection

Our primary target is the classical multi-tier web service, e.g., an Apache, application, and database server setup (a.k.a. "LAMP" stack). Most services structured this way use web servers as frontends to serve static content, application servers to manipulate session-specific dynamic content, and a DB as a data backend. In this case, the only tier that is necessary to protect in our model is the DB. This benefits our approach because further work performed by the upper tiers can be overlapped with the replication of DB writes.

To protect application state while allowing computation to overlap, we must track which outbound network packets (externally-visible events) depend on specific storage writes that are made durable at a backup site. We assume that PipeCloud is running in the VMM of each physical server and is able to monitor all of the disk writes and network packets being produced by the VMs. PipeCloud must (i) replicate all disk writes to a backup server, (ii) track the order of disk writes at the primary site and the dependencies of any network interaction on such writes and (iii) prevent outbound network packets from being released until the local writes that preceded them have been committed to the backup.

The procedure described above relies on being able to propagate information about disk writes between tiers along with all communication, and is depicted in Figure 7.5. We divide the tiers along three roles, namely *writer*, *intermediate* and *outbound*; to simplify exposition, Figure 7.5 shows one intermediate and outbound tiers, but there could be multiple tiers assuming those roles. Each tier maintains its own logical *pending write counter*, i.e., $WCnt$, $ICnt$ and $OCnt$. Note that $WCnt$ represents the true count of writes performed to the disk, while $OCnt$ and $ICnt$ represent the (possibly outdated) view of the disk state by the other tiers. In addition, the backup maintains a committed write count, $CommitCnt$. These counters are essentially monotonically increasing logical clocks, and we use these terms interchangeably.

Without loss of generality, assume all counters are zero when the system receives the first client request, which propagates through the tiers, ultimately resulting in a DB update at the writer tier. The writer tier increases the value of its pending counter after it has performed a local write, and before that write is issued to the backup site. Each tier appends the current value of its pending count to all communication with other elements in the system. Thus the writer tier propagates its pending counter through the intermediate nodes so they can update their local view: the pending counter at each non-writer tier is updated to the maximum of its own pending clock, and any pending clocks it receives from other tiers.

On receiving the write request from the writer tier at the primary site, the DR system at the backup site will commit the write to disk and then increase its committed write counter. The current value of the committed counter is then communicated back to the outbound node(s) at the primary site.

The outbound tier implements a packet buffering mechanism, tagging packets destined to external clients with its own version of the pending counter, $OCnt$, as this represents

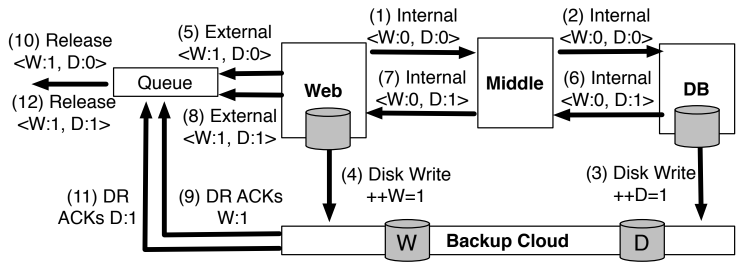


Figure 7.6. Multi-writer case with count vectors

the number of system writes which could have causally preceded the packet's creation. Packets can be released from this queue only when their pending clock tag is less than or equal to the committed clock received from the backup site. This guarantees that clients only receive a reply once the data it is dependent on has been saved. Finally, note that protection of the single VM case is covered by this scheme: the single VM becomes both a writer and outbound node.

7.3.2.2 Multiple Writer Protection

The most challenging case is when the application is in effect a distributed system: multiple nodes cooperate, and more than one of the nodes issue writes that need to be persisted. Examples include a LAMP stack with a master-master DB replication scheme or a NoSQL-style replicated key-value store.

We cater to this case by extending the notion of a logical counter to a count vector maintained by each node[1]. The pending write count for node i thus becomes the vector $P_i =< p_1, ..., p_n >$, with an entry for each of the n writers in the system. When issuing disk writes, node i increments its local counter in $P_i[i]$. All packets are tagged with the count vector of pending writes, and local knowledge is updated with arriving packets by merging the local and arriving vectors: each entry becomes the maximum of the existing and arriving entry. By exchanging information about writes in this way, the entry $P_i[j]$

[1]Our count vector is similar to a vector clock [83], but vector clocks are traditionally updated on every message send or receive, while ours only count disk writes events, similar to [73].

indicates the number of writes started by node j that i is aware of. Thus at any given time, the pending count vector represents the *write frontier* for node i—the set of writes at other nodes and itself that any computation or network packet might be causally dependent on. Note that non-writer nodes never increment write counts, only merge on packet reception, and that the single-writer case equates to a single-entry count vector.

In this manner, causality spreads across the system in a gossiping or anti-entropy manner [73, 120], all the way to outbound packets. As before, an outbound packet is tagged with the pending count vector at the moment of its creation. Outbound nodes maintain similar count vectors of committed writes $C_i =< c_1, ..., c_n >$, which represent the set of writes known to have been safely persisted. A client-bound packet can only be released once every entry in C_i is greater than or equal to the that in the packet's tag. This approach allows for a partial ordering of unrelated writes and network packets, but it guarantees that no packet is released until any write it is causally related to has been committed.

Figure 7.6 illustrates a multi-writer system. The DB and Web tiers issue writes that are persisted to the secondary in steps 3 and 4, respectively. The web tier generates an outbound packet with no knowledge of the DB write (step 5), that is buffered. A causal chain of communication emanates all the way from the DB to another outbound buffered packet, in steps 6 to 8. Acknowledgement of write commits arrive out of order, with the web tier write arriving first (step 9) and thus allowing the outbound packet dependent on the web write (but not on the DB write) to leave the system in step 10. Finally, when the DB write is acknowledged in step 11, the packet buffered in step 8 leaves the system in step 12.

7.3.3 Other DR Considerations

We end the design section by enumerating some aspects of a full disaster recovery solution that lie outside the scope of this work.

Detecting Failure: The usual approach to deciding that disaster has struck involves a keep-alive mechanism, in which the primary has to periodically respond to a ping message [78]. We note our system is no different from other DR solutions in this aspect.

Failure in the secondary: Failure of the secondary site, or a network partition, will impede propagation of data writes. Without a suitable remedial measure, externally-visible application response will be stalled waiting for replies. PipeCloud is no different from sync replication in this aspect: a reverse keep-alive is usually employed to trigger a fallback to async replication.

Memory Protection: We only protect the application state recorded to disk at the primary site. We assume that applications can be fully restored from disk in the case of disruption, as in standard database behavior. We do not attempt to protect the memory state of applications as this entails a significant overhead in WANs. Remus [32] is able to provide memory and disk protection within a LAN, but requires significant bandwidth and minimal latency. Our experiments with Remus in an emulated WAN environment with 100ms of RTT from primary to backup, show that average TPC-W response times exceeded ten seconds and replication of both disk and memory consumed over 680 Mbps of bandwidth. Providing Remus-like black-box memory protection over WAN remains an open problem due to these performance issues.

Transparent handoff: Our focus is on ensuring that storage is mirrored to the backup site and minimizing RPO. Techniques such as the virtual cloud pool infrastructure proposed in the previous chapter must be used to enable seamless network redirection after failure.

7.4 Implementation

Our PipeCloud prototype is split between a replication-aware virtualization system that is run at the primary site and a backup server at the cloud backup location. PipeCloud requires modifications to the virtualization platform at the primary, but only needs to run

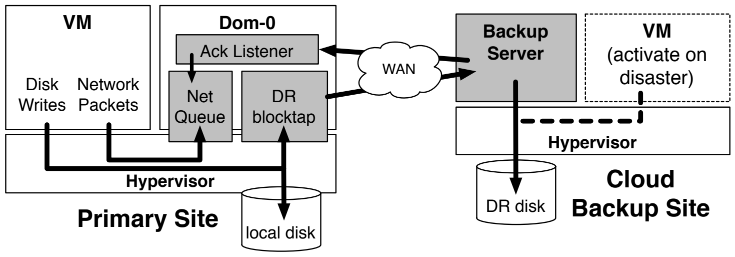


Figure 7.7. PipeCloud's implementation components

a simple user space application at the backup. This allows PipeCloud to be used today on commercial clouds which don't give users control over the low level platform.

At the primary site, PipeCloud is based on the Xen Virtual Machine Monitor (VMM) version 4.0.0. VMs in Xen perform IO using a split driver architecture with components in the guest operating system (OS) and *dom0*, a privileged domain that runs hardware drivers and a control stack. Frontend drivers in the VM's OS issue requests through shared memory and virtual interrupt lines to backend drivers. The backend drivers in dom0 unpack the requests and re-issue them against the hardware drivers. As depicted in Figure 7.7, our implementation only requires hooks at the level of virtual backend drivers; *NetQueue* and *DR Blocktap* in the figure. While we chose Xen due to familiarity, we do not foresee any problems porting our approach to VMMs like kvm or VMware ESX.

In keeping with the goal of a black-box approach, we do not mandate source code changes within protected VMs. However, we benefit from information regarding application deployment; we term this *configuration gray box.* We need a specification of the topology of the multi-tier application being persisted: which VMs make up the system, identified by their IP or MAC addresses; which VMs (or virtual disks) need to be persisted, e.g., the data disk of the DB VMs; and which nodes are allowed to perform outbound communications, e.g., the load-balancer gateway, or the Apache pool. We expect that VMs for which storage needs to be persisted will be backed by two sets of virtual disks: one storing critical application data, such as a DB bit store; and the other backing temporary files, and other miscellaneous non-critical data. Given our black box nature, this setup alleviates replication overhead (and noise) by differentiating critical disk writes which must be preserved to the backup site. All the information we require is deployment-specific, and provided by the sys-admin who configured the installation (as opposed to a developer modifying the code).

We structure this section following the nomenclature described in Section 7.3.2.1. We first describe the implementation details for *writer* nodes, then *intermediate* and finish with *outbound* nodes. We close with a discussion of the secondary site backup component.

7.4.1 Replicating Disks – VM Side

To track disk writes and replicate them to the backup server, PipeCloud uses a custom virtual disk driver backend which we dub *DR Blocktap*. This is a user-space dom0 daemon utilizing the blocktap Xen infrastructure. As a VM performs reads or writes to its protected disk, the requests are passed to our disk driver. Read requests are processed as usual.

Writes, however, are demultiplexed: they are issued both to local storage and sent through a socket to the remote site. After issuing each write, the local logical clock of pending writes, maintained as a kernel data structure in dom0, is increased. Local writes are then performed with caching disabled, so requests do not return until DMA by the local hardware driver has succeeded. At this point, we indicate to the VM that the write has completed, regardless of the status of the write traveling to the secondary site.

We note that typical OS behavior (Windows or UNIX) consists of issuing multiple writes simultaneously to leverage scatter-gather DMA capabilities. There are no expectations about ordering of writes in hardware, and a successful response is propagated upstream only after all writes in the batch have succeeded. In the absence of disk synchronization barriers (a hardware primitive that is not yet supported in Xen virtual drivers), the OS achieves `sync()`-like behavior by waiting for the batch to finish. We tag each individual write in a batch with its own value, and thus need to respect write ordering, when processing backup acknowledgements, to maintain the expected `sync()` semantics.

7.4.2 Propagating Causality

In order to track causality as it spreads throughout a multi-tier system, we need to propagate information about disk writes between tiers along with all communication. Our implementation does this by injecting the value of the local logical clock into packet headers of inter-VM communication, specifically through the addition of an IP header option in IP packets.

Virtual networking in Xen is achieved by creating a network interface in dom0. This interface injects in the dom0 network stack replicas of the Ethernet frames emanating from the VM. It replicates a frame by first copying to the dom0 address space the Ethernet, IP and TCP headers. By copying these bytes, dom0 can now modify headers (e.g., to realize NAT or similar functionality). The remainder of the packet is constructed by mapping the relevant VM memory pages read-only.

For our purposes, we split the copying of the header bytes right at the point of insertion of an IP header option. We construct our option, relying on an unused IP header option ID number (`0xb`). The IP header option payload simply contains the logical clock. We then copy the remaining header bytes. We do not introduce any extra copying, mapping, or reallocation overhead when expanding packets in this way. We modify length fields in the IP header, and recalculate checksums. Typically, the VM OS offloads checksum calculations to "hardware", which in reality is the backend virtual driver. Therefore, we do not incur additional overhead in the data path by computing checksums at this point.

Not all packets are tagged with the logical clock. Non-IP packets are ignored; in practice, non-IP packets (ARP, STP, etc) do not represent application-visible messages. Packets that already contain an IP header option are not tagged. This was done to diminish implementation complexity, but is not a hard constraint – in practice, we do not frequently see packets containing an IP header option. Optionally, empty TCP segments may not be tagged. These typically refer to empty TCP ACKS, which do not affect causality because they do not represent application-visible events. Nagle's algorithm and large bandwidth-delay products mean that success of a `write()` syscall on a TCP socket only guarantees having copied the data into an OS buffer. Additionally, most application protocols include their own application-level ACKs (e.g. HTTP 200). Empty TCP segments with the SYN, FIN, or RST flags, which do result in application visible events, are tagged.

We also considered using a "tracer" packet to communicate logical clock updates to other tiers. We ruled this out primarily due to the need to ensure correct and in-order delivery of the tracer before any other packets are allowed to proceed. Our approach, instead, does not introduce new packets, is resilient to re-ordering since logical clocks are only allowed to increase, and ensures that as long as existing packets are delivered, logical clock updates will propagate between tiers. We could not observe measurable overhead in latency ping tests, or throughput netperf tests.

Our current space overhead is 20 bytes per packet, with 16 bytes dedicated to the logical clock in the IP header option. In the case of vector clocks for multi-writer systems, this limits the size of the vector to four 32 bit entries. This is not a hard limit, although accommodating hundreds of entries would result in little useful payload per packet, and a noticeable bandwidth overhead.

7.4.3 Buffering Network Packets

Outbound nodes maintain two local logical clocks. The clock of pending writes is updated by either (or both of) the issuing disk writes or propagation of causality through internal network packets. The second clock of *committed* writes is updated by acknowledgments from the DR backup site. Comparing the two clocks allows us to determine if a network packet produced by the VM can be released or if it must be temporarily buffered.

We achieve this by appending a queueing discipline to the network backend driver (*NetQueue*), which tags outbound packets with the current pending write clock. Because logical clocks increase monotonically, packets can be added at the back of the queue and taken from the front without the need for sorting. Packets are dequeued as updates to the committed logical clock are received.

We make no assumptions on the behavior of disk IO at the secondary site, and thus need to consider that write completion may be acknowledged out of order – this is particularly relevant given our previous discussion on `sync()`-like behavior. Out-of-sequence acknowledgements are thus not acted upon until all intermediate acks arrive.

For vector clocks with two or more entries, we use a set of cascading queues, one for each entry in the vector. As different entries in the committed vector clock are updated, the affected packets at the front of the corresponding queue are dequeued and inserted in the next queue in which they have to block. Insertion uses binary search to maintain queue ordering. Once popped from the last queue, packets leave the system.

7.4.4 Storage in the Cloud Backup Site

At the secondary site, a Backup Server collects all incoming disk writes, commits them to a storage volume which can be used as the disk of a recovery VM if there is a disaster, and acknowledges write commits to the primary site. The Backup Server is a user level process, and thus does not require any special privileges on the backup site; our evaluation demonstrates how we have deployed PipeCloud using the Amazon Elastic Compute Cloud and Elastic Block Store services.

When there is only a single disk to be protected, the Backup Server performs writes directly to the backup storage volume. These writes should be performed in a durable manner which flushes them past the OS level cache. For the multi-writer case, a single Backup Server receives write streams from multiple protected VMs. Unlike the single disk case, this means that writes from multiple disks must be preserved respecting total ordering, if possible. Without special handling, the WAN link, the kernel and the disk controller at the Backup Server may all reorder writes. If a failure occurred, this could result in a write being preserved without causally-precedent writes having survived. We emphasize that *no* *existing DR service addresses these concerns*, as they preserve the contents of disks for each server in a distributed application independently.

To avoid this problem, we use the vector clocks maintained at each primary server as a guide for how writes to the backup disks should be ordered. When a primary sends a disk write to the backup, it includes its current vector clock. Ordering of the vector clocks indicates causality precedence and allows the Backup Server to enforce the same ordering in its writes. Without having application-level hints, we allow any ordering for writes which are considered to have concurrent vector clocks.

In the ideal case, the Backup Server maintains dedicated hardware (an SSD or a separate log-structured rotating disk) to initially record the writes it receives, with metadata about each write prepending the data. Writes are thus persisted, and acknowledgements returned to the primary, with minimal latency. A separate consumer process then opportunistically transfers the writes to the actual VM disk volumes. The vector clock-based ordering of writes among tiers is performed at this later stage, outside of the critical acknowledgement path. While we have not implemented this, we can preserve a causal history of each disk by retaining all writes with their logical clocks. All storage volumes are backed by RAID or hardware with similarly strong durability guarantees.

Unfortunately, the reality of cloud storage is removed from these requirements. Services such as Amazon's EBS, do not offer many necessary guarantees. There is no user control to ensure disk writes are uncached, committed in-order, and have actually moved from memory to the disk. In the event of infrastructure outages, there are loose availability guarantees [102]—simultaneous use of multiple providers has been proposed to mitigate this [12, 15]. Virtualization hides the details of hardware features which are relevant to the performance of a backup server, such as availability of SSDs, or physical layout for log-structured partitions.

Recent events indicate a shift in cloud providers toward providing SLAs in their storage services [102]. This could be offered as a differential service, and would represent the ideal substrate for cloud-based DR. We believe DR makes for an extremely compelling case to move forward on storage durability and availability guarantees. SLAs already in place by many providers [9], point toward widely realizing these basic primitives in the short term.

7.5 Black-Box Causality and RPO

Guaranteeing that clients will experience the same recovery semantics upon disaster as with synchronous replication is closely tied to our ability to introspect causality relations on a black-box VM. We start analyzing the guarantees we provide, and our limitations, with this basic building block.

As argued earlier, in a single-VM system, we can use the local machine clock as a "global" clock to timestamp writes and network packets and derive a total ordering of events; in this case, holding a packet until all writes with timestamps lower or equal to the packet is sufficient to ensure all causally dependent writes have finished. Next consider the multi-VM single-writer scenario.

Lemma 1 *In a multi-VM single writer system, it is sufficient to hold a network packet until the commit count at the secondary becomes greater than or equal to the local counter value at the outbound node.*

Proof Sketch: At the writer node, tagging internal messages with the local write counter captures *all* writes that were issued prior to sending out this internal message (and thus, all causally dependent writes as well). As shown in Figure 7.5, each node computes the max of its local counter and the one on the arriving message; since the counter at other nodes lag the writer node, doing so propagates the counter value from the writer node. At the outbound node, holding the network packet until writes committed by the secondary exceed this counter ensures that all causally dependent writes issued by the writer node have been completed at the secondary. ∎

Our PipeCloud implementation uses three additional mechanisms to ensure that this property holds in practice even in multi-core environments: (i) the Xen backend driver

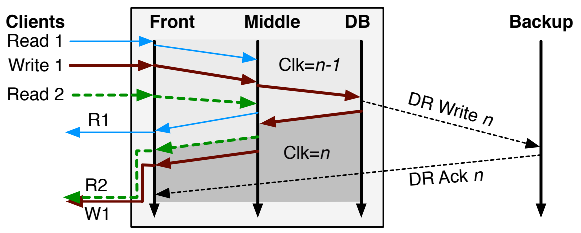


Figure 7.8. Pipelined Synchrony determines dependency information from communication between tiers. The replies R2 and W1 must be buffered because they are preceded by the database write, but reply R1 can be sent immediately.

executes in a serialized manner for a given virtual device (ii) we limit VMs to a single network interface and a single protected block device (iii) the clock is updated with atomic barrier instructions.

Finally consider the general multi-VM multi-writer case. Here we resort to using count vectors that track a "write frontier" of causally dependent writes at each writer. Like in the single writer case, the dependencies are propagated by piggybacking count vectors on internal messages. Upon message receipt, a node computes the max. of the local and piggybacked count vector thereby capturing the union of all causally dependent writes that need to be tracked. Thus, the following applies:

Lemma 2 *In a multi-writer system, it is sufficient for PipeCloud to release a network packet once all writes with a count vector less than or equal that of the network packet have finished.*

Proof Sketch: The count vector, m, that an outgoing message is tagged with, represents the events at each writer node that happened-before the packet was created. If $m[i] = k$, then all writes up to k at writer node i must have causally preceded packet m. Likewise, any write with value greater than k at node i is considered to have happened concurrently (or after) packet m. Hence, holding the packet until all writes in m have finished on all machines ensures that all causally related writes complete before network output becomes visible. ∎

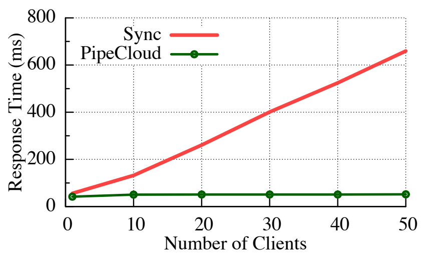
Armed with these conclusions, we revisit client RPO guarantees from Section 7.2.2. First, the Secondary view (SV) remains a subset of the Primary (PV): writes are propagated to the secondary after issuance in the primary. Second, and most importantly, the Client View (CV) remains a subset of the SV: the CV is only updated when client-bound packets are released. Thus, PipeCloud is no worse than sync replication, and therefore yields a client RPO of zero. Per lemma 2, clients will not receive updates that are not contained in the secondary site, and therefore no inconsistencies will arise after disaster recovery. Clients thus perceive synchronous replication and PipeCloud as indistinguishable.

One limitation of our black-box causality tracking is that our approach conservatively marks all writes issued before an outgoing message as dependent; while this set of writes contains all causally dependent writes, it may include other independent writes as well. Since our black-box system has no application visibility, we are unable to discern between dependent and independent writes, requiring us to conservatively mark all prior writes as dependent for safety. To illustrate, consider two separate application threads processing requests on different data items. PipeCloud cannot differentiate between these threads and, as a result, may conservatively mark the network packets from one thread as being dependent on the writes from the other thread which happened to precede them, regardless of actual application level dependence. This is illustrated in Figure 7.8: while both reads are independent from "Write 1", "Read 2" is affected by communication between the middle and DB tier after "Write 1" is propagated to the secondary. As a result, the reply to "Read 2" is unnecessarily delayed.

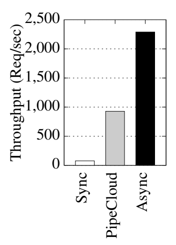
7.6 Evaluation

We have evaluated the performance of PipeCloud under normal operating conditions, as well as its failure properties, using both a local testbed and resources from Amazon EC2. On our local testbed, we use a set of Dell servers with quad core Intel Xeon 2.12GHz CPUs and 4GiB of RAM. The servers are connected by a LAN, but we use the Linux *tc* tool to emulate network latency between the hosts; we have found that this provides a reliable WAN network emulation of 50 or 100 ms delays. Our EC2 experiments use “Large” virtual machine instances (having two 64-bit cores and 7.5GiB of memory) in the US East region. Virtual machines in both the local testbed and EC2 use CentOS 5.1, Tomcat 5.5, MySQL 5.0.45, and Apache 2.23. We compare three replication tools: DRBD 8.3.8 in synchronous mode, our pipelined synchrony implementation, and an asynchronous replication tool based on our pipelined synchrony but without any network buffering.

Our evaluation focus is on client-server applications and we consider three test applications. 1) We use the **MySQL** database either by itself or as part of a larger web application. We use the InnoDB storage engine to ensure that database writes are committed to disk before replying to clients, and we store all of the InnoDB data and log files on a protected disk partition. In single VM experiments, we communicate directly with the MySQL database via a Java application running on an external client machine. 2) **TPC-W** is an e-commerce web benchmark that emulates an online bookstore. TPC-W is composed of two tiers, a Tomcat application server and a MySQL database, that each run in separate virtual machines; we only protect the disk used for the database files. TPC-W includes a client workload generator which we run on a server which is considered external to the protected system. 3) We have also written a PHP based web application called **CompDB** which allows us to more precisely control the amount of computation and database queries performed when processing requests. The application can be deployed across one, two, or three tiers, each of which runs an Apache server and a MySQL database. Requests are generated by the *httperf* tool and access a PHP script on the front-end server which performs a configurable amount of computation and insert queries to the local database before being propagated to the next tier which repeats the process. Together, these applications allow us to emulate realistic application workloads and perform controlled analyses of different workload factors.



(a) MySQL Response Time



(b) Throughput

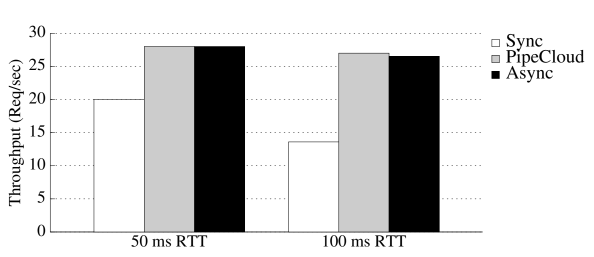
Figure 7.9. Clients must wait for multiple WAN delays with Sync, but PipeCloud has a consistent response time just barely over the RTT of 50 ms. By reducing the time DB tables must be locked for each transaction, PipeCloud is able to provide a much higher throughput relative to Sync.

7.6.1 Single Writer Database Performance

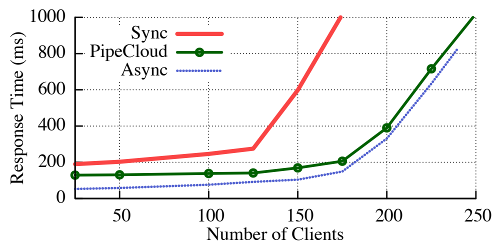
We first measure the performance of PipeCloud when protecting the disk of a MySQL database. We imposed a 50ms RTT from primary to backup in our local testbed; this limits the minimum possible response time to 50ms for the non-asynchronous approaches.

Figure 7.9(a) shows how the response time changes as the client load on the database increases. Each client connects to the database server and repeatedly inserts small 8 byte records into a table. Since the protected application is a database that must ensure consistency, the table must be locked for each individual transaction so they can be performed serially. With Sync, the table must be locked for at least one RTT because a transaction cannot complete until the remote disk acknowledges the write being finished. This means that when a client request arrives, it must wait for a round trip delay for each pending request that arrived before it. This causes the response time, when using Sync, to increase linearly with a slope based on the round trip delay to the backup.

In PipeCloud, the database table only needs to be locked until the local disk write completes, allowing for much faster request processing. This results in a much lower response time and higher throughput because many requests can be processed during each round trip delay to the backup. Figure 7.9(b) shows that PipeCloud achieves a maximum throughput over twelve times higher than Sync. While using an asynchronous approach may allow



(a) TPC-W Maximum Throughput



(b) TPC-W Response Time (50 ms RTT)

Figure 7.10. PipeCloud has higher TPC-W throughput than synchronous, and performs almost equivalently to an asynchronous approach when there is a 50 ms round trip delay.

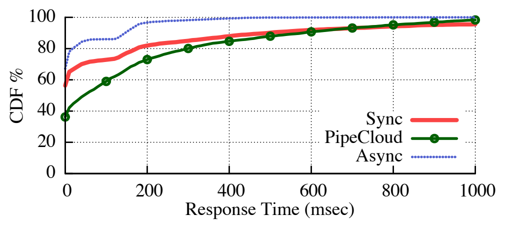
for an even higher throughput, PipeCloud provides what asynchronous approaches cannot: *zero data loss*.

7.6.2 Multi-Tier TPC-W Performance

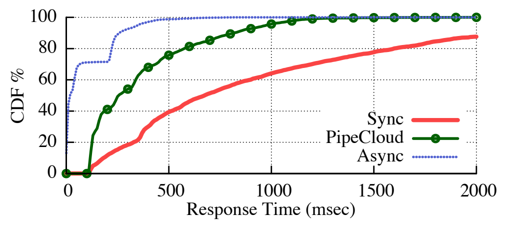
We use PipeCloud to protect a set of virtual machines running the TPC-W online store web benchmark to see the performance of a realistic application with a mixed read/write workload.

We first measure the overall performance of TPC-W as we vary the latency between the primary and backup site when using different replication mechanisms. Figure 7.10(a) shows the maximum throughput achieved by the different replication schemes. PipeCloud's maximum throughput is nearly identical to that of an asynchronous scheme—the ability to pipeline request processing and state replication effectively masks the overhead of disaster recovery. When the round trip delay increases to 100 ms, the throughput of synchronous drops even further, but PipeCloud's performance is effectively unaffected. PipeCloud is able to maintains a throughput two times better than the synchronous approach.

In Figure 7.10(b) we see that PipeCloud's pipelining also reduces response times compared to a synchronous approach, even for relatively low client loads where the throughput of each approach is similar. The load-response time curve for PipeCloud closely follows the asynchronous approach, offering a substantial performance benefit compared to synchronous and the same level of consistency guarantees.



(a) Read Requests



(b) Write Requests

Figure 7.11. With 100 ms RTT, the black-box network buffering in PipeCloud causes some read-only requests to have higher response times, but it provides a significant performance improvement for write requests compared to synchronous.

We next categorize the request types of the TPC-W application into those which involve writes to the database and those which are read-only. The workload contains a mix of 60% reads and 40% writes, and we measure the response times for each category. Figure 7.11(a) shows a CDF of the response times for read-only requests when there are 50 active clients and there is a 100 ms roundtrip time to the backup. PipeCloud has a slightly higher base response time because some read-only requests are processed concurrently with requests which involve writes. Since PipeCloud cannot distinguish between the packets related to read-only or write requests, it must conservatively buffer both types. However, even with some requests being unnecessarily delayed, PipeCloud's overall performance for reads is very close to synchronous DRBD.

PipeCloud's greatest strength shows when we observe the response time of requests that involve at least one database write in Figure 7.11(b). PipeCloud's ability to overlap work with network delays decreases the median response time by 50%, from over 600 ms to less than 300 ms. Only 3% of requests to PipeCloud take longer than one second; with synchronous replication that rises nearly 40% . This improved performance allows PipeCloud to be used with much more stringent performance SLAs.

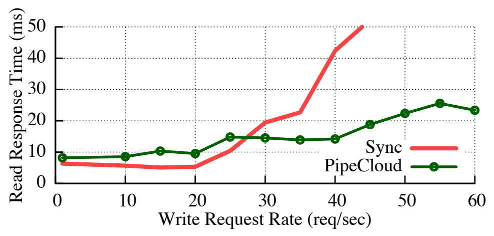


Figure 7.12. PipeCloud's black box network buffering causes read requests to initially see higher delay than Sync, but pipelining supports a much larger write workload than Sync.

7.6.3 Impact of Read and Write Rates

This experiment explores how the network buffering in PipeCloud can unnecessarily delay read-only requests that are processed concurrently with writes. We use our CompDB web application in a single-VM setup and send a constant stream of 100 read requests per second as well as a variable stream of write requests that insert records into a protected database. The read requests return static data while the writes cause a record to be inserted to the database. There is a 50 ms RTT between primary and backup. Figure 7.12 shows how the performance of read requests is impacted by the writes. When there is a very low write request rate, the response time of Sync and PipeCloud are very similar, but as the write rate rises, PipeCloud sees more read-only packets being delayed. However, the increased write rate also has a performance impact on the read requests in Sync because the system quickly becomes overloaded. PipeCloud is able to support a much higher write workload and still provide responses to read requests within a reasonable time. We believe that the trade-off provided by PipeCloud is a desirable one for application designers: a small reduction in read performance at low request rates is balanced by a significant reduction in write response times and support for higher overall throughput.

7.6.4 Multi-tier Sensitivity Analysis

To verify PipeCloud's ability to hide replication latency by overlapping it with useful work, we performed an experiment in which we arbitrarily adjust the amount of computation in a multi-tier server. We use the CompDB application split into two tiers; the front

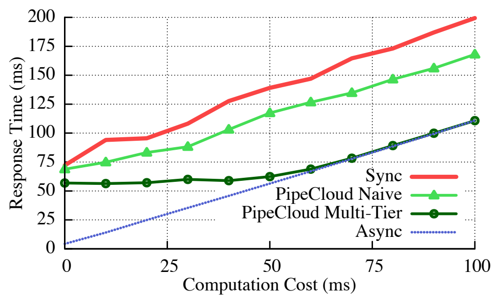


Figure 7.13. PipeCloud continues processing as writes are sent to the backup site, allowing it to provide equivalent performance to asynchronous replication if there is sufficient work to do.

tier performs a controlled amount of computation and the backend inserts a record into a database. We also compare PipeCloud against a naïve version that only applies pipelining to the DB tier. The RTT for the backup site is 50ms.

Figure 7.13 shows how the average response time changes as a function of the controlled amount of computation. As the computation cost increases, the synchronous and naïve PipeCloud approaches have a linear increase in response time since the front tier must wait for the full round trip before continuing further processing. However, when PipeCloud is applied across the two tiers, it is able to perform this processing concurrently with replication, essentially providing up to 50 ms of "free computation". For requests that require more processing than the round trip time, PipeCloud provides the same response time as an asynchronous approach, with the advantage of much stricter client RPO guarantees.

7.6.5 Protecting Multiple Databases

With current approaches, often only a single tier of an application is protected with DR because it is too expensive in terms of cost and performance to replicate the state of multiple application tiers. To evaluate PipeCloud's support for multiple servers with protected storage we consider a 3-tier deployment of our CompDB application configured so each tier includes both a web and database component. Figure 7.14 shows the average

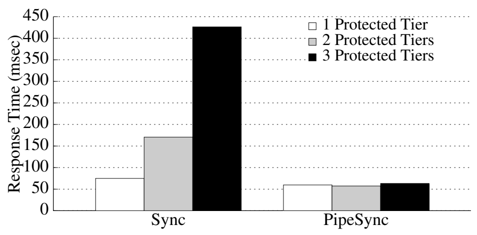


Figure 7.14. Each tier protected with synchronous replication increases response time by at least one RTT. Pipelining the replication of writes provides a much lower response time.

response time of requests to this application when either one, two, or all three of the tiers are protected by a DR system. There is a 50 ms RTT, and we use a single client in order to provide a best case response time. With synchronous replication, the response time increases by more than a round trip delay for every tier protected since the writes performed at each tier must be replicated and acknowledged serially.

PipeCloud on the other hand, is able to pipeline the replication processes across tiers, providing both better overall performance and only a minimal performance change when protecting additional tiers. When protecting all three tiers, PipeCloud reduces the response time from 426 ms to only 63 ms, a 6.7 times reduction. Being able to pipeline the replication of multiple application components allows PipeCloud to offer zero data loss guarantees to applications which previously would have resorted to asynchronous replication approaches simply due to the unacceptable performance cost incurred by serial, synchronous replication.

7.6.6 Failure evaluation

7.6.6.1 Comparing replication strategies

PipeCloud seeks to provide performance on-par with asynchronous, but it also seeks to assure clients of the same consistency guarantee as a synchronous approach. Figure 7.15 compares the consistency guarantees provided by Sync, Async, and Pipelined Synchrony. As in Section 7.6.1, a single MySQL database is backed up to a secondary site 50 msecs away. Out of a set of 500 database inserts that occur immediately prior to a disaster, we

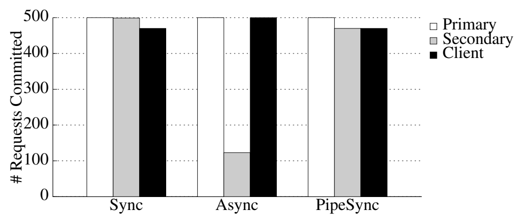


Figure 7.15. Primary, secondary, and client view of an application state when a disaster strikes after 500 MySQL updates. Both sync- and pipelined sync-based approaches guarantee that the client view is bounded by the state of the secondary.

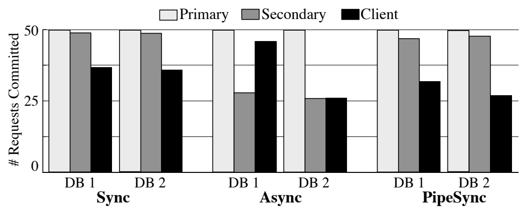


Figure 7.16. When protecting multiple databases, PipeCloud still guarantees that clients do not receive replies unless data has been durably committed to the backup.

examine the number of records recorded at the primary and secondary sites and the number of confirmations received by the client. Sync guarantees that the secondary view (SV) is almost identical to the primary (PV), but allows the client (CV) to lag behind the secondary. With Async, the client is not limited by the rate of confirmations from the secondary site, causing the client to receive many unsafe replies (CV ¿ SV). However, with Pipelined synchrony, these unsafe replies do not occur because network packets are buffered until the secondary's acks are received; as a result, PipeCloud is able to provide clients with the same guarantee as synchronous—the data for any response they receive will always have been safely committed to the backup site.

7.6.6.2 Recovering to EC2

Next we emulate a disaster scenario in which the secondary site takes over request processing for a CompDB stack, configured with a frontend node that performs database insertions on a master-master database split between two backend nodes. We run the primary site within our local testbed and use Amazon EC2 VMs to run both the Backup and the failover servers. The network latency measured from our primary site in western Massachusetts to the backup server in EC2's northern Virginia site was 16 ms. We use EBS volumes for the two protected DB disks. Prior to the failure, a single EC2 VM acting as the Backup Server applies the write streams received from both of the protected VMs to the EBS volumes.

Upon failure, the Backup Server disables replication and unmounts the EBS volumes. It uses the EC2 API to instantiate three new VM instances, and connects the two backup volumes to the newly started VMs. During bootup, the database VMs perform a consistency check by launching the `mysqld` process. Once this is complete, the application resumes processing requests. The table below details the time (in seconds) required for each of these steps; in total, the time from detection until the application is active and ready to process requests took under two minutes.

	Detach	Reattach	Boot	**Total**
Time (s)	27	13	75	**115**

Figure 7.16 shows the consistency views for the last fifty requests sent to each of the DB masters prior to the failure. As in the single writer case described in the previously, PipeCloud's multi-writer DR system is able to provide the consistency guarantee that client view will never exceed what is safely replicated to the secondary.

This experiment illustrates the potential for automating the recovery process using cloud platforms such as EC2. The API tools provided by such clouds automate steps such as provisioning servers and reattaching storage. While the potential to reduce recovery time using cloud automation is very desirable, it remains to be seen if commercial cloud platforms can provide the availability and durability required for a disaster recovery site.

7.7 Related Work

Disaster Recovery is a key business functionality and a widely researched field. We have previously mentioned the work by Keeton and Wilkes on treating asynchronous replication as an optimization problem, and balancing the two primary concerns: financial objectives and RPO deltas [66]. In Seneca [60] the space of asynchronous replication of storage devices is studied, with a focus on optimizations such as write and acknowledgment buffering and coalescing. A similar study is carried out, although at the file system level, in SnapMirror [96]. Recent studies show a large potential for high yield of write coalescing in desktop workloads [110]. We have not directly leveraged the insights of write coalescing in our work due to our stringent zero-RPO objectives.

Another area of interest for storage durability is achieving survivability by architecting data distribution schemes: combinations of striping, replication, and erasure-coding are used to store data on multiple sites [147]. These concerns become paramount in cloud storage, in which the durability and availability guarantees of providers are, to say the least, soft. Two recent systems attack these deficiencies by leveraging multiple cloud providers: DepSky [12] provides privacy through cryptography, erasure coding and secret sharing, while Skute [15] aims to optimize the cost/response time tradeoff. We highlight that while availability and durability are increased, so is latency—unfortunately. Replication chaining techniques previously used in industry [38] may complement these techniques and ameliorate the latency overhead, possibly at much higher cost.

A long tradition of distributed systems research underpins our work. We use logical clocks to track causality, originally introduced by Leslie Lamport [74]. Further, we use techniques traditionally associated with eventual consistency [13, 73] to enforce pipelined synchronous replication throughout the nodes that make up a distributed service. We employ a vector clock-style approach [83] to allow each node to represent its knowledge of the system at the moment of producing data, and we allow nodes to propagate their knowledge to peers, as in anti-entropy or gossiping protocols [120]. Previous work in inferring causality in a distributed systems of black-boxes [2] focused on performance diagnosis as opposed to consistency enforcement during replication.

The concept of speculative execution has been used to reduce the impact of latency in a variety of domains [136, 92]. These approaches typically require application support for rolling back state if speculation must be cancelled. Pipelined synchrony also uses speculative execution, but it must cancel speculated work only if the primary fails; since it is the primary which performs the speculation, the roll back process is implicit in the failover to the secondary site. This allows PipeCloud to perform speculation and rollback in a black box manner without requiring any special support from clients, nor the protected application or OS. External synchrony [92] is a related system, as it shows that in many cases the benefits of synchronous and asynchronous IO can be simultaneously reaped by intelligently overlapping IO with processing. Their treatment is focused on file system activity in a single host, and requires operating system support for tracking dependencies between processes. Remus [32] implements VM lockstep replication for LAN-based fault tolerance using similar concepts. We expand these themes to WAN replication, disaster recovery, and multi-tier causality tracking.

7.8 PipeCloud Conclusion

Cloud computing platforms are desirable to use as backup locations for Disaster Recovery due to their low cost. However, the high latency between enterprise and cloud data centers can lead to unacceptable performance when using existing replication techniques. This chapter has shown how pipelined synchronous replication overcomes the deleterious effects of speed-of-light delays by overlapping or pipelining computation with replication to a cloud site over long distance WAN links. Pipelined synchrony offers the much sought after goal: performance of asynchronous replication with the same guarantees to clients as synchronous replication. It does so by ensuring network packets destined for external entities are only released once the disk writes they are dependent on have been committed at both the primary and the backup. Our evaluation of PipeCloud demonstrates dramatic performance benefits over synchronous replication both in throughput and response time for a variety of workloads. MySQL database throughput goes up by more than an order of magnitude and the median response time for the TPC-W web application drops by a half. Recreating failures also shows PipeCloud delivers on the promise of high performance coupled with the proper consistency in the client's view of storage. PipeCloud improves the reliability of data centers by providing high performance disaster recovery services without requiring any modifications at the OS or application levels.

CHAPTER 8

SUMMARY AND FUTURE WORK

8.1 Thesis Summary

This thesis has explored how virtualization technologies can be used to improve resource management, simplify deployment, and increase the resilience of modern data centers. We have proposed a set of automated, application agnostic techniques that exploit the speed and flexibility of virtualization to handle the scale and dynamics of data center applications.

Transitioning to Virtualized Data Centers: First we proposed an automated modeling technique to characterize the overheads of virtualization platforms. Our evaluation demonstrates the importance of modeling multiple types of I/O and achieves an error rate of less than 10%. Our approach showed how the overheads of a virtualization platform can change dramatically based on workload, and in the future these models could be useful for helping users compare different virtualization platforms to determine which will be the best fit for a given application.

VM Placement and Memory Provisioning: We developed a memory sharing based VM placement technique to help data center operators more effectively deploy applications into virtualized data centers. Our efficient memory fingerprinting technique is orders of magnitude faster than a brute force comparison, and automatically detects VMs with similar memory contents so they can be placed on a single machine. This work illustrates how the hypervisor is capable of transparently gathering useful data about virtual machines without making any assumptions about the applications or operating system running within.

As the scale of data centers and the diversity of applications within them increases, these application agnostic approaches will become even more important.

Dynamic Migration and Resource Management: Next we presented automated hotspot detection and mitigation algorithms that utilize dynamic VM resource allocation and live migration to prevent server overloads within a data center. This system effectively detects and mitigates hotspots in less than 20 seconds, and explores the benefits of having additional OS or application level monitoring data. We demonstrated that in many cases, a fully application agnostic approach is sufficient for responding to server overloads.

Cross Data Center Migration: We also have developed an optimized form of VM migration which can be used over WAN links. Our optimizations eliminate transmission of redundant data, reducing bandwidth consumption and migration time by as much as fifty percent. Just as virtual machine migration within the LAN has expanded the scope of resource provisioning within a data center from looking at a single server to considering groups of server racks, we believe WAN migration will enable joint resource management across groups of data centers, despite the low bandwidth and high latency links between them.

High Performance Disaster Resiliency: Finally, we discussed how to improve data center reliability in the face of unexpected disasters. We have proposed a new application agnostic replication technique that allows cloud data centers to offer high performance, economical disaster recovery services despite the high network latencies they typically incur. Our approach pipelines computation and data replication to improve performance, but guarantees consistency by delaying network packets bound for external clients until disk state has been safely committed to the backup. This increases throughput by an order of magnitude and lowers response times by half compared to synchronous approaches.

We have used a variety of approaches for dealing with the planning, management, and reliability problems facing data centers. Our contributions include new modeling techniques, high level, automated control systems, and low level mechanisms that improve the efficiency of virtualization platforms. In summary, this thesis has explored how virtualization can make data centers more efficient and reliable without requiring any knowledge of the applications running within them.

8.2 Future Work

In this section we discuss some future research directions that have emerged from the work in this dissertation.

Cloud Transition Planning: The virtualization overhead models produced by MOVE are useful for understanding how a single application's needs will change when it is moved to a virtual environment. Extending this to assist with the transition to a public cloud would require additional models that account for the type and cost of resources available in the cloud. A system which could determine the resource needs, expected cost, and performance impact of moving into a cloud platform would be of great use to businesses considering this transition.

The Changing Goals of Data Center Management: The resource management techniques proposed in Sandpiper seek to optimize application performance by distributing work across as many machines as possible. However, in some cases a data center operator may be more concerned with minimizing operational costs than maximizing performance. Further work is required to allow resource management systems such as Sandpiper and Memory Buddies to jointly optimize potentially competing goals such as performance and energy efficiency.

Modeling & Automating WAN Migrations: The optimized WAN VM migration tools provided by CloudNet can significantly lower migration cost, but there is no way for users to know in advance what the time and bandwidth cost of a migration will be. Having models that could predict this cost would enable automated control systems such as Sandpiper to be extended to manage pools of resources that span multiple data centers. Such a system could support seamless "cloud bursting" by dynamically replicating and migrating applications into the cloud, while accounting for both the performance and monetary cost of such a move.

Causal Storage: The replication system in PipeCloud enforces consistency across multiple components by requiring that causally related writes are performed in order (i.e., based on their count vectors). Further work is required to better understand how application level consistency requirements translate to the disk and network level events tracked by PipeCloud. This is particularly important for applications which do not perform all writes to storage synchronously, but may be able to provide application level notifications to the DR system to know when state must be buffered and released. A further interesting extension would be to consider how the vector count tagged writes at the DR site could be used as a "causal storage time machine" which would allow the disk state of distributed applications to be rolled back in time in a consistent way.

Taint Tracking: The count vector packet injection performed in Pipe Cloud is useful for tracking disk consistency across tiers, but it could also be used for other purposes. One possibility is to use the messages injected at the virtualization layer as a way of tracking potential intrusions. For example, the counters could track potential security violations; as these messages spread between machines it could give forensic investigators a way of tracking which machines might have been corrupted by an intrusion.

BIBLIOGRAPHY

[1] Aggarwal, B., Akella, A., Anand, A., Chitnis, P., Muthukrishnan, C., Nair, A., Ramjee, R., and Varghese, G. EndRE: An end-system redundancy elimination service for enterprises. In *Proceedings of NSDI* (2010).

[2] Aguilera, Marcos K., Mogul, Jeffrey C., Wiener, Janet L., Reynolds, Patrick, and Muthitacharoen, Athicha. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of SOSP* (2003).

[3] Ahmad, I., Anderson, J., Holler, A., Kambo, R., and Makhija, V. An analysis of disk performance in VMware ESX server and virtual machines. In *Proceedings of the Sixth Workshop on Workload Characterization (WWC'03)* (October 2003).

[4] Anand, Ashok, Sekar, Vyas, and Akella, Aditya. SmartRE: an architecture for coordinated network-wide redundancy elimination. *SIGCOMM Comput. Commun. Rev. 39*, 4 (2009), 87–98.

[5] `http://jakarta.apache.org/jmeter`.

[6] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. Oceano - sla-based management of a computing utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management* (May 2001).

[7] Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D., Katz, Randy H., Konwinski, Andrew, Lee, Gunho, Patterson, David A., Rabkin, Ariel, Stoica, Ion, and Zaharia, Matei. Above the clouds: A Berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[8] Aron, M., Druschel, P., and Zwaenepoel, W. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA* (June 2000).

[9] AT&T. Synaptic Storage as a Service.
https://www.synaptic.att.com/clouduser/services_storage.htm.

[10] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY* (October 2003), pp. 164–177.

[11] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the Art of Virtualization. In *Proc. of SOSP* (Bolton Landing, NY, Oct. 2003).

[12] Bessani, Alysson, Correia, Miguel, Quaresma, Bruno, Andre, Fernando, and Sousa, Paulo. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of Eurosys* (2011).

[13] Birrell, Andrew D., Levin, Roy, Schroeder, Michael D., and Needham, Roger M. Grapevine: an Exercise in Distributed Computing. *Communications of the ACM 25*, 4 (1982).

[14] Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[15] Bonvin, Nicolas, Papaioannou, Thanasis, and Aberer, Karl. A Self-Organized, Fault-Tolerant and Scalable Replication scheme for Cloud Storage. In *Proc. of SOCC* (2010).

[16] Box, G. P., Jenkins, G. M., and Reinsel, G. C. *Time Series Analysis Forecasting and Control Third Edition*. Prentice Hall, 1994.

[17] Bradford, Robert, Kotsovinos, Evangelos, Feldmann, Anja, and Schiöberg, Harald. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments* (San Diego, California, USA, 2007), ACM, pp. 169–179.

[18] Breitgand, David, Kutiel, Gilad, and Raz, Danny. Cost-aware live migration of services in the cloud. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (New York, NY, USA, 2010), SYSTOR '10, ACM.

[19] Broder, Andrei, and Mitzenmacher, Michael. Network applications of Bloom filters: A survey. *Internet Mathematics 1*, 4 (2003), 485–509.

[20] Brown, Aaron B., and Seltzer, Margo I. Operating system benchmarking in the wake of lmbench: a case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1997), SIGMETRICS '97, ACM, pp. 214–224.

[21] Bugnion, Edouard, Devine, Scott, and Rosenblum, Mendel. DISCO: Running Commodity Operating Systems on Scalable Multiprocessors. In *SOSP* (1997), pp. 143–156.

[22] Buyya, Rajkumar, Ranjan, Rajiv, and Calheiros, Rodrigo N. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing* (2010).

[23] Cecchet, Emmanuel, Chanda, Anupam, Elnikety, Sameh, Marguerite, Julie, and Zwaenepoel, Willy. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *4th ACM/IFIP/USENIX International Middleware Conference* (June 2003).

[24] Chandra, Abhishek, Gong, Weibo, and Shenoy, Prashant. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (San Diego, CA, USA, 2003), ACM, pp. 300–301.

[25] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2001), p. 103116.

[26] Chen, Xu, Mao, Z Morley, and Van der Merwe, Jacobus. ShadowNet: a platform for rapid and safe network evolution. In *USENIX Annual Technical Conference* (2009).

[27] Cherkasova, L., and Gardner, R. Measuring CPU overhead for I/O processing in the xen virtual machine monitor. In *USENIX Annual Technical Conference* (Apr. 2005).

[28] Church, K., Hamilton, J., and Greenberg, A. On delivering embarassingly distributed cloud services. *Hotnets VII* (2008).

[29] Cisco Active Network Abstraction. `http://www.cisco.com`.

[30] Clark, Christopher, Fraser, Keir, Hand, Steven, Hansen, Jacob Gorm, Jul, Eric, Limpach, Christian, Pratt, Ian, and Warfield, Andrew. Live Migration of Virtual Machines. In *Proc. of NSDI* (Boston, MA, May 2005).

[31] Clidaras, Jimmy, Stiver, David, and Hamburgen, William. Water-Based data center (patent application 20080209234).

[32] Cully, Brendan, Lefebvre, Geoffrey, Meyer, Dutch, Feeley, Mike, Hutchinson, Norm, and Warfield, Andrew. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of NSDI* (2008).

[33] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. In *Symposium on Operating Systems Principles* (2004).

[34] Draper, N. R., and Smith, H. *Applied Regression Analysis*. John Wiley & Sons, 1998.

[35] Drbd. `http://www.drbd.org/`.

[36] Duffield, N. G., Goyal, Pawan, Greenberg, Albert, Mishra, Partho, Ramakrishnan, K. K., and Van der Merwe, Jacobus E. Resource management with hoses: point-to-cloud services for virtual private networks. *IEEE/ACM Transactions on Networking 10*, 5 (2002).

[37] Amazon ec2 elastic load balancing. http://aws.amazon.com/elasticloadbalancing/.

[38] EMC. Symmetrix Remote Data Facility (SRDF) Product Guide. `http://www.scribd.com/doc/35959543/Symmetrix-SRDF-Product-Guide`.

[39] Fraser, Keir, Hand, Steven, Neugebauer, Rolf, Pratt, Ian, Warfield, Andrew, and Williamson, Mark. Reconstructing I/O. *Technical Report* (2004).

[40] Garfinkel, Tal, Pfaff, Ben, Chow, Jim, Rosenblum, Mendel, and Boneh, Dan. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (Bolton Landing, NY, USA, 2003), ACM, pp. 193–206.

[41] Gmach, Daniel, Rolia, Jerry, Cherkasova, Ludmila, and Kemper, Alfons. Capacity management and demand prediction for next generation data centers. *Web Services, IEEE International Conference on 0* (2007), 43–50.

[42] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)* (December 1999), pp. 154–169.

[43] Grit, Laura, Irwin, David, , Yumerefendi, Aydan, and Chase, Jeff. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *Workshop on Virtualization Technology in Distributed Computing (VTDC)* (November 2006).

[44] Guo, Deke, Wu, Jie, Chen, Honghui, and Luo, Xueshan. Theory and Network Applications of Dynamic Bloom Filters. In *INFOCOM* (2006).

[45] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006), Melbourne, Australia* (November 2006).

[46] Gupta, D., Gardner, R., and Cherkasova, L. XenMon: QoS monitoring and performance profiling tool. Tech. Rep. HPL-2005-187, HP Labs, 2005.

[47] Gupta, Diwaker, Lee, Sangmin, Vrable, Michael, Savage, Stefan, Snoeren, Alex C., Varghese, George, Voelker, Geoffrey M., and Vahdat, Amin. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM 53*, 10 (2010), 85–93.

[48] Hajjat, M., Sun, X., Sung, Y., Maltz, D., Rao, S., Sripanidkulchai, K., and Tawarmalani, M. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of SIGCOMM* (2010).

[49] Harney, Eric, Goasguen, Sebastien, Martin, Jim, Murphy, Mike, and Westall, Mike. The efficacy of live virtual machine migrations over the internet. In *Proceedings of the 3rd VTDC* (2007).

[50] Holland, P. W., and Welsch, R. E. Robust regression using iteratively reweighted least-squares. In *Communications in Statistics - Theory and Methods* (October 2007).

[51] HP Integrity Essentials Capacity Advisor. `http://h71036.www7.hp.com/enterprise/cache/262379-0-0-0-121.html`.

[52] HP-UX Workload Manager. `http://www.hp.com/products1/unix/operating/wlm`.

[53] Hsieh, Paul. *Hash functions*. http://www.azillionmonkeys.com/qed/ hash.html.

[54] Huang, Wei, Gao, Qi, Liu, Jiuxing, and Panda, Dhabaleswar K. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing* (2007), IEEE Computer Society, pp. 11–20.

[55] Microsoft hyper-v server. www.microsoft.com/hyper-v-server.

[56] IBM Enterprise Workload Manager. `http://www.ibm.com/developerworks/autonomic/ewlm/`.

[57] IBM Tivoli Performance Analyzer. `http://www.ibm.com/software/tivoli/products/performance-analyzer/`.

[58] Isard, Michael, Budiu, Mihai, Yu, Yuan, Birrell, Andrew, and Fetterly, Dennis. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal, 2007), ACM, pp. 59–72.

[59] Jain, Navendu, Dahlin, Michael, and Tewari, Renu. Using Bloom Filters to Refine Web Search Results. In *WebDB* (2005), pp. 25–30.

[60] Ji, Minwen, Veitch, Alistair, and Wilkes, John. Seneca: Remote Mirroring Done Write. In *Proc. of Usenix ATC* (2003).

[61] Jin, Hai, Deng, Li, Wu, Song, Shi, Xuanhua, and Pan, Xiaodong. Live virtual machine migration with adaptive memory compression. In *Cluster* (2009).

[62] Jones, Stephen T., Arpaci-Dusseau, Andrea C., and Arpaci-Dusseau, Remzi H. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM Press, pp. 14–24.

[63] Juniper Networks, Configuration and Diagnostic Automation Guide. `http://www.juniper.net`.

[64] Kamra, Abhinav, Misra, Vishal, and Nahum, Erich. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *International Workshop on Quality of Service (IWQoS)* (June 2004).

[65] Katz, Randy. IEEE spectrum: Tech titans building boom. http://www.spectrum.ieee.org/green-tech/buildings/tech-titans-building-boom.

[66] Keeton, Kimberly, Santos, Cipriano, Beyer, Dirk, Chase, Jeffrey, and Wilkes, John. Designing for Disasters. In *Proc. of FAST* (2004).

[67] King, Samuel T., Dunlap, George W., and Chen, Peter M. Operating system support for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (San Antonio, Texas, 2003), USENIX Association, pp. 6–6.

[68] Kleinrock, L. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc., 1976.

[69] Kloster, Jacob, Kristensen, Jesper, and Mejlholm, Arne. On the Feasibility of Memory Sharing: Content-Based Page Sharing in the Xen Virtual Machine Monitor. Master's thesis, Department of Computer Science, Aalborg University, June 2006.

[70] Kozuch, M., and Satyanarayanan, M. Internet suspend and resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY* (June 2002).

[71] Kulkarni, Purushottam, Shenoy, Prashant J., and Gong, Weibo. Scalable Techniques for Memory-efficients CDN Simulations. In *WWW* (2003).

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

[73] Ladin, Rivka, Liskov, Barbara, Shrira, Liuva, and Ghemawat, Sanjay. Providing High Availability Using Lazy Replication. *ACM TOCS 10*, 4 (1992).

[74] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978).

[75] Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, M., Tantawi, A., and Youssef, A. Performance Management for Cluster Based Web Services. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management* (2003), vol. 246, pp. 247–261.

[76] libvirt. *The Virtualization API*. http://libvirt.org.

[77] Lim, K., Ranganathan, P., Chang, J., Patel, C., Mudge, T., and Reinhardt, S. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on* (2008), pp. 315–326.

[78] Linux-HA. Heartbeat. http://linux-ha.org/wiki/Heartbeat.

[79] Liu, Jiuxing, Huang, Wei, Abali, Bulent, and Panda, Dhabaleswar K. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Boston, MA, 2006), USENIX Association, pp. 3–3.

[80] Lu, Pin, and Shen, Kai. Virtual machine memory access tracing with hypervisor exclusive cache. In *Usenix Annual Technical Conference* (June 2007).

[81] Luo, Xucheng, Qin, Zhiguang, Geng, Ji, and Luo, Jiaqing. IAC: Interest-Aware Caching for Unstructured P2P. In *SKG* (2006), p. 58.

[82] Magenheimer, Daniel J., and Christian, Thomas W. vblades: optimized paravirtualization for the itanium processor family. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3* (Berkeley, CA, USA, 2004), USENIX Association, pp. 6–6.

[83] Mattern, Friedemann. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms* (1989).

[84] McVoy, Larry, and Staelin, Carl. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), USENIX Association, pp. 23–23.

[85] Menasce, Daniel A., and Bennani, Mohamed N. Autonomic Virtualized Environments. In *IEEE International Conference on Autonomic and Autonomous Systems* (July 2006).

[86] Menon, Aravind, Santos, Jose Renato, Turner, Yoshio, Janakiraman, G. (John), and Zwaenepoel, Willy. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (Chicago, IL, USA, 2005), ACM, pp. 13–23.

[87] Milos, Grzegorz, Murray, Derek G., Hand, Steven, and Fetterman, Michael. Satori: Enlightened Page Sharing. In *Proceedings of the USENIX Annual Technical Conference* (2009).

[88] Mosberger, David, and Jin, Tai. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance* (June 1998), ACM, pp. 59—67.

[89] MySQL. `http://www.mysql.com`.

[90] Nahum, E. Deconstructing specweb. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution* (2002).

[91] Nelson, Michael, Lim, Beng-Hong, and Hutchins, Greg. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference* (2005).

[92] Nightingale, Edmund B., Veeraraghavan, Kaushik, Chen, Peter M., and Flinn, Jason. Rethink the Sync. In *Proc. of OSDI* (2006).

[93] Osman, Steven, Subhraveti, Dinesh, Su, Gong, and Nieh, Jason. The design and implementation of zap: A system for migrating computing environments. In *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[94] Padala, Pradeep, Zhu, Xiaoyun, Wang, Zhikui, Singhal, Sharad, and Shin, Kang G. Performance evaluation of virtualization technologies for server consolidation. Tech. rep., HP Labs, 2007.

[95] Parallels. www.parallels.com.

[96] Patterson, Hugo, Manley, Stephen, Federwisch, Mike, Hitz, Dave, Kleiman, Steve, and Owara, Shane. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proc. of FAST* (Monterey, CA, Jan. 2002).

[97] Pinheiro, Eduardo, Weber, Wolf-Dietrich, and Barroso, Luiz André. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 2–2.

[98] Powell, M., and Miller, B. Process Migration in DEMOS/MP. *Operating Systems Review 17*, 5 (1983), 110–119.

[99] Project, Apache Open For Business. http://ofbiz.apache.org.

[100] Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. No power struggles: A unified multi-level power management architecture for the data center. *ASPLOS, March* (2008).

[101] Ranjan, S., Rolia, J., Fu, H., and Knightly, E. Qos-driven server migration for internet data centers. In *Proceedings of IWQoS 2002, Miami Beach, FL* (May 2002).

[102] Rightscale. Amazon EC2 Outage: Summary and Lessons Learned. http://blog.rightscale.com/2011/04/25/amazon-ec2-outage-summary-and-lessons-learned.

[103] Riteau, Pierre, Morin, Christine, and Priol, Thierry. Shrinker: Efficient Wide-Area Live Virtual Machine Migration using Distributed Content-Based Addressing. Research Report RR-7198, INRIA, 02 2010.

[104] Rolia, Jerry, Cherkasova, Ludmila, Arlitt, Martin, and Andrzejak, Artur. A capacity management service for resource pools. In *Proceedings of the 5th international workshop on Software and performance* (New York, NY, USA, 2005), WOSP '05, ACM, pp. 229–237.

[105] Ruth, Paul, Rhee, Junghwan, Xu, Dongyan, Kennell, Rick, and Goasguen, Sebastien. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *IEEE International Conference on Autonomic Computing (ICAC)* (June 2006).

[106] Santos, Jose Renato, Turner, Yoshio, Janakiraman, G., and Pratt, Ian. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 29–42.

[107] Sapuntzakis, Constantine P., Chandra, Ramesh, Pfaff, Ben, Chow, Jim, Lam, Monica S., and Rosenblum, Mendel. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002).

[108] Seltzer, Margo, Krinsky, David, Smith, Keith, and Zhang, Xiaolan. The case for application-specific benchmarking. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems* (Washington, DC, USA, 1999), HOTOS '99, IEEE Computer Society, pp. 102–.

[109] Seltzsam, Stefan, Gmach, Daniel, Krompass, Stefan, and Kemper, Alfons. Autoglobe: An automatic administration concept for service-oriented database applications. In *Proceedings of the 22nd International Conference on Data Engineering* (Washington, DC, USA, 2006), ICDE '06, IEEE Computer Society, pp. 90–.

[110] Shamma, Mohammad, Meyer, Dutch T., Wires, Jake, Ivanova, Maria, Hutchinson, Norman C., and Warfield, Andrew. Capo: Recapitulating Storage for Virtual Desktops. In *Proc. of FAST* (2011).

[111] The SPEC java server benchmark. `http://spec.org/jbb2005/`.

[112] Staelin, Carl, and McVoy, Larry. mhz: anatomy of a micro-benchmark. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1998), ATEC '98, USENIX Association, pp. 13–13.

[113] Stewart, C., and Shen, K. Performance modeling and system management for multi-component online services. In *Symposium on Networked Systems Design and Implementation* (2005).

[114] Stewart, Christopher, Kelly, Terence, Zhang, Alex, and Shen, Kai. A dollar from 15 cents: cross-platform management for internet services. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Boston, Massachusetts, 2008), pp. 199–212.

[115] Sugerman, Jeremy, Venkitachalam, Ganesh, and Lim, Beng-Hong. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (2001), USENIX Association, pp. 1–14.

[116] Sundaram, V, Wood, T., and Shenoy, P. Efficient data migration in self-managing storage systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC-06), Dublin, Ireland* (June 2006).

[117] Sundararaj, A., Gupta, A., and Dinda, P. Increasing Application Performance in Virtual Environments through Run-time Inference and Adaptation. In *Fourteenth International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[118] Sundararaj, Ananth I., and Dinda, Peter A. Towards virtual networks for virtual machine grid computing. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium* (2004).

[119] Sysstat-7.0.4. `http://perso.orange.fr/sebastien.godard/`.

[120] Terry, Douglas B., Theimer, Marvin M., Petersen, Karin, Demers, Alan J., Spreitzer, Mike J., and Hauser, Carl H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of SOSP* (Copper Mountain, CO, Dec. 1995).

[121] Theimer, M. M., L., K. A., and Cheriton, D. R. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* (Dec. 1985), pp. 2–12.

[122] TPC. the tpcw benchmark. Website. `http://www.tpc.org/tpcw/`.

[123] Travostino, Franco, Daspit, Paul, Gommans, Leon, Jog, Chetan, de Laat, Cees, Mambretti, Joe, Monga, Inder, van Oudenaarde, Bas, Raghunath, Satish, and Wang, Phil Yonghui. Seamless live migration of virtual machines over the MAN/WAN. *Future Generation Computer Systems* (Oct. 2006).

[124] Urgaonkar, B., Rosenberg, A., and Shenoy, P. Application placement on a cluster of servers. In *Internernational Journal of Foundations of Computer Science* (October 2007), vol. 18, pp. 1023–1041.

[125] Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. Dynamic provisioning for multi-tier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), Seattle, WA* (June 2005).

[126] Urgaonkar, Bhuvan, Shenoy, Prashant, and Roscoe, Timothy. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev. 36* (December 2002), 239–254.

[127] Van der Merwe, J., Cepleanu, A., D'Souza, K., Freeman, B., Greenberg, A., Knight, D., McMillan, R., Moloney, D., Mulligan, J., Nguyen, H., Nguyen, M., Ramarajan, A., Saad, S., Satterlee, M., Spencer, T., Toll, D., and Zelingher, S. Dynamic connectivity management with an intelligent route service control point. In *Proceedings of the 2006 SIGCOMM workshop on Internet network management*.

[128] Virtual machine mobility with VMware VMotion and Cisco Data Center Interconnect Technologies. http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns836/white_paper_c11-557822.pdf, Sept. 2009.

[129] VMmark: A Scalable Benchmark for Virtualized Systems. `www.vmware.com/pdf/vmmark_intro.pdf`.

[130] Vmware capacity planner. `www.vmware.com/products/capacity_planner/`.

[131] Vmware esx bare-metal hypervisor. www.vmware.com/products/vi/esx.

[132] Vmware high availability product page. www.vmware.com/products/vi/vc/ha.html.

[133] Waldspurger, Carl A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 181–194.

[134] Waldspurger, Carl A., and Waldspurger, Carl A. Lottery and stride scheduling: Flexible proportional-share resource management. In *In Proc. First Symposium on Operat-ing Systems Design and Implementation* (1995), pp. 2–90.

[135] Wang, Jian, Wright, Kwame-Lante, and Gopalan, Kartik. XenLoop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing* (Boston, MA, USA, 2008), ACM, pp. 109–118.

[136] Wester, Benjamin, Cowling, James, Nightingale, Edmund B., Chen, Peter M., Flinn, Jason, and Liskov, Barbara. Tolerating latency in replicated state machines through client speculation. In *Proceedings of NSDI* (Berkeley, CA, USA, 2009), USENIX Association.

[137] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)* (Dec. 2002).

[138] Whitepaper, VMware. Drs performance and best practices.

[139] Wood, T., Gerber, A., Ramakrishnan, K., Van der Merwe, J., and Shenoy, P. The case for enterprise ready virtual private clouds. In *Proceedings of the Usenix Workshop on Hot Topics in Cloud Computing (HotCloud), San Diego, CA* (June 2009).

[140] Wood, T., Lagar-Cavilla, H. A., Ramakrishnan, K., Shenoy, P., and Van der Merwe, J. PipeCloud: Using causality to overcome speed-of-light delays in cloud-based disaster recovery. Tech. Rep. UM-CS-2011-015, University of Massachusetts Amherst Computer Science, 2011.

[141] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Sandpiper: Black-box and gray-box resource management for virtual machines. In *Computer Networks Journal (ComNet) Special Issue on Virtualized Data Centers* (2009).

[142] Wood, Timothy, Cecchet, Emmanuel, Ramakrishnan, K.K., Shenoy, Prashant, and Van der Merwe, Jacobus. Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges. In *Proc. of HotCloud* (Boston, MA, June 2010).

[143] Wood, Timothy, Cherkasova, Ludmila, Ozonat, Kivanc, and Shenoy, Prashant. Profiling and modeling resource usage of virtualized applications. In *International Middleware Conference* (2008).

[144] Wood, Timothy, Ramakrishnan, K. K., Shenoy, Prashant, and van der Merwe, Jacobus. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2011), VEE '11, ACM, pp. 121–132.

[145] Wood, Timothy, Shenoy, Prashant J., Venkataramani, Arun, and Yousif, Mazin S. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Networked Systems Design and Implementation (NSDI '07)* (2007).

[146] Wood, Timothy, Tarasuk-Levin, Gabriel, Shenoy, Prashant, Desnoyers, Peter, Cecchet, Emmanuel, and Corner, Mark. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)* (Washington, DC, USA, March 2009).

[147] Wylie, Jay J., Bakkaloglu, Mehmet, Pandurangan, Vijay, Bigrigg, Michael W., Oguz, Semih, Tew, Ken, Williams, Cory, Ganger, Gregory R., and Koshla, Pradeep K. Selecting the Right Data Distribution Scheme for a Survivable Storage System. Tech. rep., CMU, 2001. CMU-CS-01-120.

[148] Zhang, Q., Cherkasova, L., and Smirni, E. A Regression-Based analytic model for dynamic resource provisioning of Multi-Tier applications. In *Proc. ICAC* (2007).

[149] Zhang, Qi, Cherkasova, Ludmila, Mathews, Guy, Greene, Wayne, and Smirni, Evgenia. R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware* (New York, NY, USA, 2007), Springer-Verlag New York, Inc., pp. 244–265.

[150] Zhao, Weiming, and Wang, Zhenlin. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (Washington, DC, USA, 2009), ACM, pp. 21–30.