TOWARDS SUPERCLOUDS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Dan Williams

January 2013

TOWARDS SUPERCLOUDS

Dan Williams, Ph.D.

Cornell University 2013

Cloud computing has emerged as an economically attractive utility model for computational resources. An increasing number of industries, from businesses to governments, are embracing cloud computing. However, the economic benefits of the cloud computing model come at a price: the loss of control over how services and applications can use computing resources. In other words, cloud computing is fundamentally *provider-centric*. Cloud providers (the providers of computational resources), not cloud users (the consumers of computational resources), dictate rules and policies governing how computational resources can be used. For large enterprise application workloads, adhering to cloud provider rules and policies may be prohibitive. This dissertation explores the question: how can large enterprise workloads efficiently utilize and control computational resources from a variety of providers in the cloud computing model?

The main contributions of this dissertation relate to a fundamental change to the cloud computing model. Instead of a provider-centric model, we propose a *user-centric* model in which the cloud user can maintain control over how computational resources obtained from cloud providers can be used. We have devised a new abstraction, called *cloud extensibility*, to enable the implementation of provider-level functionality by cloud users. Leveraging cloud extensibility, we describe steps towards a user-centric cloud computing model that grants cloud users—including large enterprises—control over resources obtained from

one or more cloud providers. We call this new model the *supercloud* model.

More specifically, we focus on three key areas in which current provider-centric cloud computing models do not expose the necessary control or lack the features to support large enterprise workloads without significant reconfiguration effort. First, clouds are not interoperable, restricting workloads to a single provider and hindering incremental migration to the cloud. Second, clouds lack support for complex enterprise network configurations, including flow policies between application components and low-level network features (e.g., IP addresses, multicast, VLANs). Finally, high utilization of cloud resources cannot be applied through techniques like oversubscription, and existing techniques do not apply well to common workload patterns.

We subsequently make three contributions, embodied in the design, implementation and evaluation of three systems that leverage cloud extensibility. Cloud extensibility itself is instantiated in the first system, a nested virtualization layer called the *Xen-Blanket*. The Xen-Blanket additionally enables cloud interoperability by homogenizing existing cloud interfaces and services. The second system, *VirtualWire*, provides a virtual network abstraction to support complex enterprise networks in which the cloud user manages the network control logic. Finally, we present *Overdriver*, a system that enables high resource utilization through memory oversubscription and the handling of the resulting—often transient and unpredictable—memory overload. Together, these three systems are important steps towards superclouds.

BIOGRAPHICAL SKETCH

Dan Williams attended the University of Rochester from 1999 to 2003 for his undergraduate studies, earning degrees in Computer Science and Mathematics, before enrolling in the Ph.D. program in the Computer Science Department in Cornell University in 2003. There he learned a lot about building secure operating systems from Gün Sirer and Fred Schneider before joining Hakim Weatherspoon's group to focus on cloud computing. In 2010, thanks to Hakim's IBM faculty award, Dan began to collaborate with Hani Jamjoom at the IBM T. J. Watson Research Center in Hawthorne, NY. In June of that year, Dan went to Hawthorne for a summer internship, which worked out so well that he stayed at IBM. He is now a Research Staff Member in Hani's group.

for my family

ACKNOWLEDGEMENTS

First, I would like to thank the professors at Cornell who have profoundly touched my life and career. Hakim Weatherspoon, my advisor, took me on as a student when it would have been easier not to, taught me how to become an independent researcher, and helped me see the importance of questions over implementation in the research process. Gün Sirer showed me how to be a "kernel hacker", helped me climb the steep systems learning curve, and gave me a rare opportunity to participate in the building of an operating system from scratch. Paul Merrill transformed my approach to and appreciation of music, a lifelong obsession, by teaching me how music theory is applied in jazz improvisation. Finally, I thank Andrew Myers and Steve Pond for serving on my committee and always making themselves available to chat; I regret not taking more advantage of their time and generosity.

Second, many graduate students and post-docs were an excellent resource and source of companionship in the different phases of my graduate experience. Rather than an endless list of names, I would like to thank the entire systems lab in all its incarnations, as well as my Upson 4104 officemates throughout the years. I would like to extend a special thank you to Alan Shieh, Kevin Walsh, and Patrick Reynolds for teaching me so much, and to Tudor Marian for his warm welcome into Hakim's group.

Finally, Hani Jamjoom assumed a mentor, manager, and advisor role for me at IBM Research. Hani's unwavering faith in my abilities and enthusiasm for our research were unquestionably essential to my success as a researcher, and I cannot thank him enough for the opportunities he opened within IBM. His technical advice, broad career advice, and life advice have helped me find my way through graduate school and beyond.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A vast array of industries—government, medical, financial, military and consumer—heavily rely on the computational power of data centers. Services and applications, including storage and analytics on data from payroll and inventory, email, content distribution, and Web front ends for customer interaction are all powered by computational resources in data centers. Increasingly, data centers are providing computational resources as an on-demand computing utility in a model that has come to be known as the *cloud computing* model, or simply *cloud computing*.

Unfortunately, the attractive on-demand characteristics offered by cloud computing come at a price: a loss of control over resources. Services and applications running "in the cloud" must follow rules defined by the provider of the resources, and different providers may insist on different rules. In other words, cloud computing is fundamentally *provider-centric*, or governed by rules and policies defined by resource providers. Unfortunately, the larger and more complex the application workload, the more effort is required to conform the workload to provider rules. Enterprise workloads, which are defined by applications *and* the services and infrastructure required to support them, are large and complex. For example, a customer facing Web service may interact with components that monitor the health of the service, upgrade software components, patch vulnerabilities, balance load between workers, and enforce firewall rules or intrusion detection invariants on the network. Conforming an enterprise workload to provider rules is prohibitive and can lead to lock-in. In this dissertation, we ask the research question: *how can large enterprise workloads ef-*

ficiently utilize and control large quantities of on-demand resources from a variety of providers in the cloud computing model?

Through research and exploration of this question, we have devised a new abstraction, *cloud extensibility*, that fundamentally changes the way that resources can be utilized and controlled in the cloud computing model. With the cloud extensibility abstraction, the enterprise utilizing cloud resources can define its own, custom rules that fit its workload. As such, cloud extensibility transforms the provider-centric cloud computing model into a fundamentally *user-centric* model. In this dissertation we describe a new, user-centric cloud computing model based on cloud extensibility that we call the *supercloud* model, and investigate how it can enable enterprise workloads to take advantage of cloud computing.

1.1 Superclouds: A Universal System

As part of the growing trend towards the commoditization of computing resources, cloud computing is often compared to other utility models, like electricity. Akin to power generators, cloud providers offer massive amounts of computing resources. However, unlike the electricity utility model, in which consumers are generally agnostic to where power is generated, cloud users (consumers) are tightly coupled to the providers' infrastructures and must adhere to the varying specifications (e.g., virtualization stack and management APIs) of the corresponding cloud provider. As it stands, the current cloud computing model resembles the "War of the Currents" of the late 1880s [89], where direct current (DC) power distribution was tightly coupled to a local generator.

We investigate a cloud infrastructure that is akin to Westinghouse's "universal system"—the foundation of modern day power generation, distribution, and commoditization. In the "universal system," Westinghouse showed how power—using Tesla's transformer—can be generated and consumed in many different voltages [89]. Consumers, in such a model, only care about the availability of power as it can simply be transformed into the correct voltage. This way, consumers are able be completely agnostic to where and how the electricity is being generated. The "universal system," thus, demonstrated how to decouple power generation from consumption. This decoupling was a corner piece to creating power distribution networks and, ultimately, the commoditization of electricity.

In a similar fashion, the supercloud model described in this dissertation forms a cloud distribution layer that is not bound to any provider or physical resources. On the surface, users of a supercloud see a collection of computing resources, similar to the current cloud computing model (Section 1.2). Beneath the surface, the supercloud "transforms" multiple underlying cloud offerings into a universal cloud abstraction. The supercloud specifies and fully controls the entire cloud stack, independent from the providers' infrastructure. A supercloud, thus, decouples cloud providers and users.

By decoupling the task of managing physical infrastructure from the service abstraction implemented by the supercloud, computing resources are further pushed toward commoditization. Superclouds enable a service abstraction market on top of cloud providers (currently inhabited by companies like Rightscale [58]) to grow by introducing the potential for more flexibility and control. Furthermore, superclouds enable an environment within which novel systems and hypervisor-level research and experimentation can be performed. Superclouds enable the research and development of new, seamless multi-cloud applications and new, highly customized cloud environments. As the market for customized, cloud-agnostic services grows, superclouds will affect not just cloud users, but the entire cloud ecosystem.

1.2 Cloud Computing Background

The cloud computing model is characterized in part by access to computing resources as a *utility*. In this respect, cloud computing can be considered an instantiation of a research vision that dates back to the 1960's and the first time-sharing systems [59], called utility computing [69,77].

Computing resources made available as a utility in the cloud computing model are owned and managed by *cloud providers*. Cloud providers, sometimes referred to simply as *clouds*, run massive data centers consisting of racks upon racks of servers connected in a network. At the time of writing, each server in a data center may contain one or more central processing units (CPUs) with dozens of computational cores, hundreds of gigabytes (GB) of memory, terabytes (TB) of storage in the form of magnetic disks or solid state drives (SSDs), and several network interface cards (NICs) that connect the server to one or more networks supporting bandwidths exceeding 10 gigabits per second (Gbps).

In the cloud computing model, as defined by the National Institute of Standards and Technology (NIST) [112], cloud providers offer these computational resources as a service. The focus of this dissertation is on a service model known as Infrastructure-as-a-Service (IaaS).[1] In an IaaS cloud, a cloud provider makes computational resources available in the form of a virtual machine abstraction. A *virtual machine* (VM) is a computer implemented in software that executes instructions to run programs, just like a physical machine. As such, a VM contains a complete software stack, from operating system to application. The representation of a VM on disk is called a *VM image*, whereas a running instantiation of the VM image is called a *VM instance*.

The cloud provider runs software on each server, called a *hypervisor* or *virtual machine monitor* (VMM), that runs and manages one or more VMs in strict isolation from one another. The hypervisor multiplexes physical resources among VMs by assigning virtual resources to VMs. Virtual resources are software equivalents of physical resources such as CPU, memory, and network bandwidth, and are ultimately backed by physical resources. The total amount of virtual resources assigned to VMs can exceed the physical resources, and this strategy is often employed to increase physical resource utilization. A primer on *virtualization*, the technique in which a hypervisor supports the VM abstraction, appears in Appendix A.

The entity that obtains resources (CPU, network, storage) from a cloud provider is called a *cloud user*. In an IaaS cloud, a cloud user obtains resources in the form of access to a VM created by the cloud provider. The cloud user pays for the VM per some—often small—unit of time. For example, Table 1.1 shows the price per hour for several different VM configurations from the Ama-

[1]NIST defines three service models for cloud computing. In the Software as a Service (SaaS) model an application run by the provider is offered as a service. In the Platform as a Service (PaaS) model, a framework that supports programs written in a particular language or paradigm is offered as a service. The Infrastructure as a Service (IaaS) model is the most flexible, because fundamental computing resources (e.g., CPU, memory, network, storage) are offered as a service.

Amazon EC2				
Type	CPU (ECUs)	Memory (GB)	Disk (GB)	Price ($/hr)
Small	1	1.7	160	0.08
Large	4	7.5	850	0.32
Extra Large	8	15	1690	0.64
Cluster 4XL	33.5	23	1690	1.30
Rackspace				
Type	CPU (vCPUs)	Memory (GB)	Disk (GB)	Price ($/hr)
1GB	1	1	40	0.06
8GB	4	8	320	0.48
15GB	6	15	620	0.90
30GB	8	30	1200	1.20

Table 1.1: The cost per hour of VMs from Amazon EC2 and Rackspace in September 2012. For EC2, an ECU is an "EC2 Compute Unit", the equivalent CPU capacity of a 1.0 – 1.2 GHz 2007 Opteron or 2007 Xeon processor. For Rackspace, vCPUs are the equivalent of 2.5 GHz Xeon processors.

zon Elastic Compute Cloud (EC2) [29] and Rackspace [31], two popular cloud providers. This fine-grained, on-demand pricing model allows cloud users to scale up or down their applications by requesting or releasing VMs. The cloud user also interacts with the hypervisor-level software run by the cloud provider through provider-specific interfaces. For example, some providers allow the cloud user to invoke a provider function to save the running memory state of a VM to disk. Other features offered by the provider may simplify management for a cloud user or improve performance.

A cloud user may be an individual, a small organization, or a large enterprise. Thus, the amount of resources required by the cloud user can vary greatly. A 2011 survey of over 400 small to large enterprises found that 90% were interested in the cloud for a variety of reasons, including agility, scalability, and cost [8]. A cloud user running large and complex workloads will face more challenges adapting to the cloud computing model than those running small and simple workloads. Therefore, in the next section and throughout this dissertation, we investigate research questions and present abstractions that enable cloud computing to support would-be cloud users that are themselves large enterprises.

1.3 Challenges for Enterprise Workloads in the Cloud

Enterprise workloads, or the workloads for which large enterprises need computational resources are made up of applications and the services and infrastructure required to support them. They can consist of hundreds or thousands of machines that run applications (e.g., Web, payroll, inventory) and services (e.g., monitoring, software patching and update) to manage deployments at large scale. Enterprise workloads also involve sophisticated network components (e.g., protocol accelerators, firewalls, intrusion detection systems) and custom network configurations to route traffic between applications and services. Vast quantities of resources are typically provisioned to run enterprise workloads. We ask the questions: can part or all of an enterprise workload migrate to the cloud? Can the cloud computing model support large enterprise workloads?

Large enterprises face numerous challenges that prevent them from fully embracing cloud computing. First, enterprise workloads must often be re-engineered to run on a particular cloud provider and interact with the cloud's management infrastructure. Such engineering effort is costly and must be done independently for each cloud provider. Second, the complex networks common in enterprise workloads are rarely supported in existing clouds. Finally, enterprise deployments rarely utilize their provisioned resources efficiently, making it difficult for them to utilize cloud resources in an efficient and cost-effective manner.

To address these challenges, we investigate and apply a new *cloud extensibility* abstraction. Cloud extensibility enables cloud users to modify the cloud by augmenting its hypervisor-level functionality. As described in Chapter 3, cloud extensibility can be enabled on current clouds by leveraging an additional layer of virtualization via *nested virtualization*. Ultimately, by applying cloud extensibility to existing clouds, enterprises may overcome the challenges that hinder the migration of their workloads to the cloud. The remainder of this section describes each challenge in more detail, as well as how we leverage cloud extensibility to address each challenge.

1.3.1 Lack of interoperability between clouds.

Fundamentally, clouds do not offer uniform and compatible interfaces and environments that are necessary for enterprise workloads to span multiple clouds. In other words, they lack *homogeneity*. In particular, a single VM image cannot be deployed—unmodified—on any IaaS cloud. Even worse, there is no consistent set of hypervisor-level services across providers. Existing efforts towards multi-cloud homogeneity such as standardization (e.g., Open Virtualization Format [63]) are inherently *provider-centric*. Standardization approaches will likely be limited to simple cloud attributes like VM image format. More complex standards that facilitate interoperability between clouds may never come to fruition.

The challenge is in enabling a *cloud user* to create a uniform environment across heterogeneous clouds, without special support from the provider, thereby enabling the VMs comprising enterprise workloads to be deployed on any IaaS cloud. Specifically, we address the research question: *How can a cloud user homogenize clouds without relying on providers to implement standards or any additional support?*

To support enterprise workloads, an approach towards interoperability between clouds must maintain the powerful VM abstraction that defines IaaS clouds, enabling the same VM image to run on multiple clouds. Furthermore, the technique must afford the cloud user the flexibility to implement hypervisor-level services that may span multiple clouds. For example, it must be possible to implement a service that performs live VM migration between clouds. Finally, the technique must not rely on standardization so as to be applicable to existing clouds. Unfortunately, current techniques do not meet these requirements. Middleware, such as IBM's Altocumulus [109] system homogenizes IaaS clouds like Amazon EC2 [29] into a Platform-as-a-Service (PaaS) [112] abstraction, which is a higher level abstraction that lacks the flexibility of the VM abstraction provided by IaaS clouds. Rightscale ServerTemplates [58] can run on any cloud but cannot utilize hypervisor-level features (e.g., live VM migration) between clouds. Eucalyptus [68] and OpenStack [30] are open-source cloud computing systems that can enable private infrastructures to share an API with Amazon EC2 and Rackspace respectively, but also do not allow the user to implement their own multi-cloud hypervisor-level features. Finally, the RESER-

VOIR project [131] relies on standardization.

With cloud extensibility, a user can *homogenize* cloud offerings. By homogenize, we mean a user can provide a uniform interface and set of services across heterogeneous cloud providers, thus facilitating interoperability between them. Instead of standardization, which is by definition provider-centric, we propose a *user-centric* approach that gives users an unprecedented level of control over the virtualization layer. As an instance of this approach, we introduce the *Xen-Blanket*, a thin, deployable virtualization layer that can homogenize diverse, existing cloud infrastructures. We have deployed the Xen-Blanket across a public cloud, Amazon's EC2; a separate enterprise cloud; and a third private setup at Cornell University. The Xen-Blanket is an instantiation of cloud extensibility; it enables cloud users to implement hypervisor-level functionality on third-party clouds, including live VM migration, oversubscription, and ultimately can reduce costs for users. We describe our approach to address the lack of interoperability between clouds, embodied in the Xen-Blanket, in more detail in Chapter 4.

1.3.2 Lack of control in cloud networks.

Enterprise deployments are notoriously difficult to extract from the network infrastructure they run on. They are not simply the sum of the installed applications (e.g., web server + application server + database), but also include *flow policies* that are encoded in arcane network configurations and middleboxes [146] (e.g., firewall, intrusion detection system, and load balancers). Furthermore, they contain dependencies on low-level network features (e.g., IP addresses, multicast, and VLANs). Virtual network abstractions provided by systems like VL2 [75], NetLord [118], and Nicira [14] are rich and support many of the intricacies of enterprise workloads. However, in these systems, the *control logic*, responsible for encoding flow policies in the virtual network, is implemented by the provider. Users are fundamentally limited; they can only interact with the control logic using cloud-specific, high-level APIs.

The challenge is in providing a virtual network abstraction within clouds such that an enterprise cloud user has the control to configure the virtual network, thereby enabling enterprise workloads—and their networks—to run on one or more clouds. Specifically, we address the research question: *What virtual network abstraction should a cloud provider expose to allow an enterprise deployment and its network to be migrated—without modification—to the cloud?*

To support enterprise workloads, an approach towards cloud networking must exhibit a rich network abstraction to cloud users. The exposed network abstraction must support features and protocols—layer 2 (data link layer) protocols, for example—common in enterprise workloads. The technique must not require modification of guest VMs, which can be prohibitive for large enterprise workloads. Finally, it must be possible and straightforward to specify the complex flow policies encoded in network components that are part of enterprise workloads. Unfortunately, existing approaches do not meet these requirements. Rich virtual network abstractions [14] are emerging in current clouds, which leverage OpenFlow [110] and, more broadly, software defined networks (SDN) [78, 98, 126]. However, they are provider-centric. A cloud user must interact with a cloud-defined interface to configure the network and therefore can only configure the network in particular ways. Other systems, like Amazon Virtual Private Cloud (VPC) [1] do not support layer 2 protocols. vCider [22] and VPN-Cubed [24] support layer 2 protocols in the cloud and even provide some control over the network topology, but require configuration in the guest operating systems. CloudSwitch [3] operates in an isolation layer that avoids guest operating system configuration, but does not facilitate the implementation of flow policies in the cloud.

Cloud extensibility enables an enterprise user to deploy virtual infrastructure to support complex aspects of an enterprise workload, such as its network configuration. We propose and investigate a virtual network abstraction in which the cloud user—not the provider—is responsible for the virtual network control logic. As an instance of this approach, we present *VirtualWire*, a system that can subsequently reduce or eliminate the network re-engineering effort required to migrate an enterprise workload to the cloud. On VirtualWire, users directly run their own virtualized equivalent of network devices (such as switches, routers, middleboxes), and configure them using low-level device interfaces, identically to physical network devices. Behind an API that mimics the act of plugging networking cables into network interface cards, the provider's role is reduced to maintaining location-independent point-to-point connections. Using VirtualWire, we have migrated a 3-tier application and its complex network topology onto EC2 and achieved performance close to a native EC2 deployment, avoiding reconfiguration. VirtualWire maintains the network topology and flow policies even as components move. We have performed cross-cloud live migration of VMs and network components making up an enterprise workload between Cornell and EC2 with minimum downtime. We describe our approach that affords cloud users more control over cloud networks, embodied in VirtualWire, in more detail in Chapter 5.

1.3.3 Lack of efficient resource utilization.

Enterprise deployments often provision enough resources to support peak load conditions. As a consequence, resource utilization is low most of the time. Enterprise deployments present an opportunity for *oversubscription* in order to increase utilization. An enterprise workload is oversubscribed if the aggregate amount of resources actually allocated to its VMs is less than the amount of resources that were requested. Oversubscription can lead to *overload*, a situation in which resource demands exceed the amount of allocated resources. While overload can happen with respect to any resource, memory overload is particularly devastating to application performance. Existing techniques are heavyweight and not well suited to transient overload.

The challenge is in allowing an enterprise deployment to exploit oversubscription to increase resource utilization in the cloud. At the same time overload—particularly memory overload—must be managed, regardless of whether the overload is an unexpected transient burst or a predictable sustained phase change. Specifically, we address the research question: *How can enterprise workloads exploit memory oversubscription in the cloud, while handling performance degradation due to memory overload?*

To address the lack of efficient resource utilization through memory oversubscription, a viable approach must be able to preserve the performance of enterprise workloads even through times of overload. A technique should be able to react quickly to handle transient bursts. It should also be able to alleviate more sustained periods of unusually high load. Unfortunately, existing systems do not meet these requirements. Most existing approaches to handle memory overload due to oversubscription utilize live VM migration [57, 122] to adjust the physical resource utilization in the data center. Such techniques cannot react to transient overload, which is common in oversubscribed deployments. Sandpiper [158] initiates migrations based on utilization thresholds. Andreolini et al. [37] use a trend analysis, while Stage and Setzer [138] advocate long-term migration plans. Alternatively, Disco [73] and MemX [86] use network memory [27, 36, 62, 71, 106, 123] during memory overload, but cannot alleviate sustained periods of overload with high performance.

Cloud extensibility grants an enterprise cloud user the ability to oversubscribe cloud resources, such as memory, if desired. As such, the cloud user can implement novel strategies to handle memory overload, whether because of transient bursts or sustained phase changes. We propose and investigate an approach where overload is treated as a continuum that includes both transient and sustained overloads of various durations. As a result, overload mitigation approaches can also be viewed as a continuum, complete with tradeoffs with respect to application performance and data center overhead. As an instance of a continuum-based approach to handling overload, we present *Overdriver*, a system that adaptively takes advantage of these tradeoffs, using a threshold-based strategy to switch between using network memory to handle transient overloads and live VM migration to handle sustained overloads. Overdriver mitigates all overloads while maintaining close to well-provisioned performance. Furthermore, under reasonable oversubscription ratios, where transient overload constitutes the vast majority of overloads, Overdriver requires less excess space and generates significantly less network traffic than a migration-only approach. We describe our approach to address the lack of efficient cloud resource utilization, embodied in Overdriver, in more detail in Chapter 6.

1.4 Contributions Towards a Supercloud Model

Through investigation of the three challenges in the previous section (Section 1.3), this dissertation describes our research towards the universal system described in Section 1.1. In particular, our research contributions center around defining and exploring the *supercloud* model for cloud computing. Superclouds embody an approach to cloud computing that is fundamentally different from what currently exists. Rather than relying on cloud providers to dictate the format, functionality, and limitations of cloud computing, we propose that this responsibility be shifted to the cloud user. More specifically, we propose that cloud users control their own *supercloud*.

As shown in Figure 1.1, in the supercloud model, an enterprise cloud user maintains the ability to manage and efficiently run workloads on any cloud, regardless of whether the user owns the infrastructure. Rather than redesigning an enterprise deployment for the cloud, superclouds can be customized to fit the enterprise. The complexity of migrating enterprise workloads to the cloud using the supercloud model is dramatically reduced when compared to the current cloud model. Resources from multiple clouds appear to be part of a single cloud offering, all under the control of the (enterprise) cloud user. The cloud offering can support complex, custom enterprise networks and apply oversubscription techniques for efficiency. Furthermore, different enterprise cloud users can each deploy one or multiple superclouds that remain completely independent from one another.

The core contribution of this dissertation, then, is the investigation of the new supercloud model for cloud computing. We have researched this model

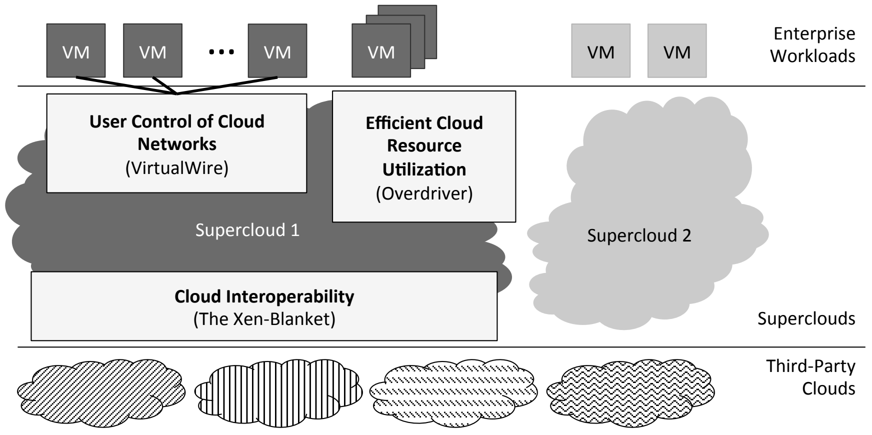

VM
VM
VM
VM
VM
VM
Enterprise Workloads
User Control of Cloud Networks (VirtualWire)
Efficient Cloud Resource Utilization (Overdriver)
Supercloud 1
Supercloud 2
Cloud Interoperability (The Xen-Blanket)
Superclouds
Third-Party Clouds

Figure 1.1: The steps towards superclouds that are detailed in this dissertation are illustrated in this Figure. Each system is a part of a supercloud and addresses a specific challenge pertaining to the migration of enterprise deployments to the cloud.

by introducing *cloud extensibility*, which enables a cloud user to implement hypervisor-level functionality. We describe cloud extensibility in detail, show how it can be applied across heterogeneous cloud providers, and directly or indirectly leverage it to instantiate the vision of superclouds. We then make three key contributions towards superclouds, described in detail in Chapters 4–6, respectively. Each contribution appears in Figure 1.1 in the context of a supercloud. We present:

- **Cloud Interoperability:** a user-centric approach to homogenize clouds, or provide a uniform interface and set of services across heterogeneous cloud providers, embodied by the Xen-Blanket,
- **User Control of Cloud Networks:** a virtual networking abstraction for the cloud that enables the cloud user to implement the network control logic and therefore control all aspects of the cloud network, embodied in VirtualWire, and

- **Efficient Cloud Resource Utilization:** an approach to handle memory overload that enables high utilization of cloud resources through oversubscription by handling the entire continuum of transient to sustained overload, embodied in Overdriver.

1.5 Organization

The remainder of this dissertation is organized as follows. The scope of the problem—including a characterization of enterprise workloads—and methodology is described in Chapter 2. Cloud extensibility, the enabling abstraction for superclouds, is presented in Chapter 3. Chapter 4 addresses the lack of interoperability in clouds and presents the Xen-Blanket. Leveraging the Xen-Blanket, Chapter 5 tackles the networking aspect of enterprise workloads and their deployment on third-party clouds with VirtualWire. Chapter 6 describes efficient use of memory resources through oversubscription and handling overload with Overdriver. Chapter 7 surveys related work, Chapter 8 describes future work, and Chapter 9 concludes.

CHAPTER 2

SCOPE AND METHODOLOGY

This chapter describes the problem scope and methodology of this dissertation. We are researching how large enterprises can efficiently utilize cloud resources from a variety of cloud providers. In order to define the scope of the problem, we first characterize enterprise workloads in the context of research challenges. Then, we describe our methodology for evaluating and validating our research contributions.

2.1 Scope: Understanding Enterprise Workloads

Enterprise workloads typically consist of a set of applications and the services and infrastructure required to support them. Enterprise workloads can include storage systems, data analytics, payroll and inventory applications, email servers, Web proxies, and Wide Area Network (WAN) optimizers among other components. Further, enterprise workloads may consist of hundreds or thousands of servers that communicate with each other [136] in complex patterns that reflect elaborate deployment architectures. Even a seemingly simple, standalone customer facing Web service may interact with a number of services. For instance, it is common to run monitoring agents in the virtualization stack [25], in the OS [108], or in the application itself. Agents may be running to manage and schedule software upgrades or interact with other systems to apply security patches. Additionally, enterprise workloads include infrastructure, such as network middleboxes [52] that may balance load between workers or enforce firewall rules and intrusion detection invariants on the network.

In this section, we define the scope of the problem. In particular, we describe how the characteristics of enterprise workloads make them unsuitable for current state-of-the-art clouds (Section 1.2). We identify the characteristics of enterprise workloads in the context of the three challenges enumerated in Section 1.3. First, in Section 2.1.1, we describe how cloud interoperability and the flexibility to define the cloud environment is crucial for enterprise cloud users. Second, in Section 2.1.2, we investigate the network infrastructure that is included in an enterprise workload and motivate the need for control over cloud networks. Finally, in Section 2.1.3, we examine the opportunities for oversubscription in enterprise workloads and characterize the overload that must be handled in order to achieve efficient resource utilization.

2.1.1 Enterprise Workloads Need Cloud Interoperability

An enterprise cloud user would achieve several benefits from a uniform hypervisor interface that encompasses many different cloud providers. If a single VM image can be deployed on every cloud, tasks common to enterprise workloads, such as image management, upgrading, and patching, are simplified. If any service offered by one cloud were available in any other cloud, enterprises would not feel locked in to a particular vendor. In a 2011 survey of over 400 enterprises, 25% cited provider lock-in and lack of interoperability as the top inhibitors preventing them from moving to the cloud [8]. *Homogeneity*, by which we refer to a uniform environment between heterogeneous clouds, would enable hypervisor-level resource management techniques and cloud software stacks that truly span providers, offering enterprises the control to manage cloud resources as they see fit to exploit their full potential.

Existing clouds lack homogeneity in three ways. First, VM images—the on-disk representations of VMs and the building blocks of cloud applications—cannot be easily instantiated on different clouds. Despite proposed standards, cloud providers continue to use different formats. For example, Amazon EC2 uses the Amazon Machine Image (AMI) format, while Rackspace uses the Open Virtualization Format (OVF) [63]. Second, clouds are diverse in terms of the services they provide to VMs. For example, Amazon EC2 provides tools such as CloudWatch (integrated monitoring), AutoScaling, and Elastic Load Balancing, whereas Rackspace contains support for VM migration to combat server host degradation and CPU bursting to borrow cycles from other instances. Third, a class of resource management opportunities that exist in a private cloud setting—in particular, tools that operate at the hypervisor level—are not consistently available between providers. For example, there is no unified set of tools with which enterprises can specify VM co-location on physical machines [159], page sharing between VMs [81, 145], or resource oversubscription [154].

The desire for a homogeneous interface across cloud providers is *not* a call for standardization. We distinguish between *provider-centric* and *user-centric* homogenization. Standardization is an example of provider-centric homogenization, in which every cloud provider must agree on an image format, services, and management interfaces to expose to enterprise cloud users. Standards are emerging; for example, Open Virtualization Format (OVF) [63] describes how to package VM images and `virtio` defines paravirtualized device interfaces. However, until *all* clouds (e.g., Amazon EC2, Rackspace, Google Compute Engine, Microsoft Azure, to name a few) adopt these standards, VM configurations will continue to vary depending on the cloud they run on. Even worse, it is unlikely—probably infeasible—that the vast array of current and future services available to VMs will become standardized across all clouds. Attempts at standardization often lead to a set of functionality that represents the "least common denominator" across all participating providers. Many enterprises will still demand services that are not in the standard set and cloud providers will continue to offer services that differentiate their offering. As a result, standardization, or provider-centric homogenization, is not sufficient.

In contrast, we consider user-centric homogenization, in which enterprise cloud users can homogenize the cloud and customize it to match their needs. User-centric homogenization allows enterprises to select their own VM image format and services, then transform every cloud to support it. The enterprise is not tied into a "least common denominator" of functionality, but quite the opposite: even completely customized services and image formats can be deployed. The enterprise can then develop management tools that work for *their* VMs across *their* (now homogenized) cloud. For example, an enterprise cloud user can experiment with new features like high availability [61] across clouds and perhaps achieve high availability even in the case of an entire provider failing.

Finally, any system that implements user-centric homogenization should be universally deployable on clouds as they currently exist to have greatest impact. A system that enables user-centric homogenization cannot be dependent on emerging features that are not standard across all clouds. For example, at the time of writing this dissertation, low-overhead nested virtualization primitives in the Turtles Project [43] have been incorporated in two popular VMMs: Xen [41] and KVM [96]. However, these primitives are not exposed by any cloud providers, and therefore cannot be assumed by systems providing user-centric

Function	Application
Policy-based routing through middleboxes	• Network Intrusion Detection Systems (Honypots, Snort [132]) • Firewalls, load balancers, transparent proxies • Protocol Acceleration (Snoop [39], Interceptor [9], BigIP [2]) • IPv6 Tunnels [53]
Layer 2 support	• VLAN (VLAN Trunking Protocol [55], Isolation, VoIP) • Broadcast (Microsoft Network Load Balancing [13]) • Fail-over (PaceMaker [15], Sun Cluster [16]) • Discovery (Link Layer Discovery Protocol [28]) • Storage Area Network (Fibre Channel over Ethernet [7], ATA over Ethernet [87])
(L2 and L3) Multicast support	• Network Virtualization (VXLAN [105]) • Fail-over (Linux-HA [11], Hot Standby Router Protocol [104],Virtual Router Redundancy Protocol [120]) • Load Balancing (Microsoft Network Load Balancing [13])

Table 2.1: Features used by enterprise deployments, commonly unsupported by existing clouds

homogenization. Chapter 4 describes a system that homogenizes existing cloud interfaces and services, thus facilitating enterprise workloads even across multiple clouds.

2.1.2 Enterprise Workloads Need To Control the Network

Enterprise deployments require not just a virtual network abstraction in the cloud, but they require access to the control logic for the virtual network. To support this claim, we examined data from a large organization that routinely performs migration of enterprise workloads between data centers and various cloud infrastructures.[1] Using this data, we conducted a qualitative study of twenty-six large (production) enterprise migration efforts, each ranging from 800 to over 6,000 (physical and virtual) machines.[2] Our study included de-

[1]The name of the large enterprise is omitted for anonymity.

[2]We note that these were a combination of physical-to-virtual and virtual-to-virtual migrations as part of data center outsourcing engagements.

ployed applications, network elements (e.g., firewalls, switches, routers), logical dependencies, security and resource requirements, service level agreements, and projected outages. We also conducted a number of interviews with experts involved in the migration efforts. In this subsection, we highlight our key findings. Unfortunately, due to the sensitivity of the data, we only provide qualitative reports, rather than raw data.

Network Dependencies. Enterprise deployments not only have strong dependencies on network protocols (examples of which appear in Table 2.1), but also on how the network is managed. Administrators and the tools they use often rely on vendor-specific features such as the Cisco command-line interface (CLI) [21] and the Encapsulated Remote Switched Port Analyzer (ERSPAN) [4]. Migrating to a cloud requires (1) network protocol support by the provider, and (2) modification of existing configuration and management tools to work with the cloud APIs. We observed that applications and their management tools often need to be fundamentally re-architected to fit the target cloud model.

Network-Encoded Flow Policies. Within the network of an enterprise deployment, network flows travel—often transparently—through middleboxes, such as firewalls and protocol accelerators (Table 2.1). These *flow policies* are encoded in low-level, local configurations of network components, including switches, routers, and middleboxes themselves. Migrating enterprise workloads to a cloud requires (1) extracting these local configurations and (2) translating them into the semantics (and APIs) exposed by the cloud. These requirements imply that the cloud provider's virtual network abstraction must contain robust support for flow policies, which is not currently the case across all mainstream clouds.

Globally Unavailable View of Configurations. Enterprise deployments are further complicated by their sheer size, often spanning thousands of hosts and applications. We observed a consistent absence of a single knowledge base that captures all configurations. While there have been attempts to capture such a global view [48], they are rarely applied effectively. Migrating to a cloud thus requires (1) full understanding of global configurations and (2) ensuring that the migration fully adheres to any embedded policies (e.g., security, isolation, and QoS).

Reconfiguring an enterprise application to fit one of the existing cloud virtual network models can be avoided if the enterprise cloud user is responsible for its virtual network's control logic. In particular, given an appropriate virtual network abstraction, the enterprise can completely reproduce the intricacies of the physical network without translation or global network knowledge. Towards this goal, we discuss a design alternative that allows an application and virtualized analogue of its network components to be migrated—without modification—to the cloud in Chapter 5.

2.1.3 Enterprise Workloads Need Efficient Resource Utilization

Oversubscribing resources, or granting more resources than actually exist, is one technique to achieve efficiency in enterprise cloud workloads. However, oversubscription must not cause overload in the VMs comprising the workload. In this section, we first describe various causes of memory overload, with particular focus on overload due to oversubscription. Second, we justify the claim that an opportunity for memory oversubscription exists in enterprise deployments through analysis of data center logs from a large enterprise.[3] Finally, we experimentally examine the characteristics of overload caused by oversubscription to conclude that overload is a continuum, with transient overloads dominating.

Types of Overload. A VM is *overloaded* if the amount of memory allocated to the VM is insufficient to support the working set of the application component within the VM. There are two main causes of memory overload. The first cause of memory overload is that a VM does not request sufficient resources to handle its working set, including the case in which the running application component has a memory leak. We assume that the VM should have requested more resources to eliminate this type of overload.

We thus only focus on investigating, minimizing and possibly eliminating overload caused by oversubscription; we assume that the VM, if allocated all resources it requested, would not experience overload. We call the amount of memory that the cloud dedicates to a VM the *memory allocation*. If the VM's memory allocation is less than requested, then we say the machine hosting the VM is *oversubscribed*. Oversubscribing memory while preserving performance is possible because application components running in VMs do not require a constant amount of memory, but experience application-specific fluctuations in memory needs (e.g., change in working set). In practice, memory oversubscription can be accomplished by taking memory away from one VM to give to another through memory ballooning [145], transparent page sharing [145], or

[3]Once again, we omit the name of the enterprise to preserve anonymity.

other techniques [81]. If the aggregate memory demand of VMs sharing an machine exceeds the amount of memory on the machine, overload must be managed so that a VM continues to execute as if it has the amount of memory it requested.

Opportunities for Oversubscription. To justify the opportunity for memory oversubscription in enterprise workloads, we examine log data from a number of production enterprise data centers, which tend to be well-provisioned. The log data covers a number of performance metrics (including CPU, memory, and disk usage) for a large data center that hosts diverse applications, including Web, financial, accounting, and customer relationship management (CRM). The collected performance data is typically used by the various data centers to analyze application resource usage in order to identify resource contention and to assess the need for workload rebalancing.

Two indicators are generally used by data centers to identify whether a server is having memory overload problems: page scan rate and paging rate. Paging rate is the primary indicator because it captures the operating system's success in finding free pages. In addition, the page scan rate captures the rate at which the operating system is searching for free pages, providing an early indicator that memory utilization is becoming a bottleneck.

In well-provisioned data centers, overload is unpredictable, relatively rare, uncorrelated, and transient, indicating that an opportunity exists for memory oversubscription. To support this claim, we processed performance logs from 100 randomly selected servers. Each log is 24 hours long, while each point in the trace is the average paging rate over a fifteen-minute interval. This is the

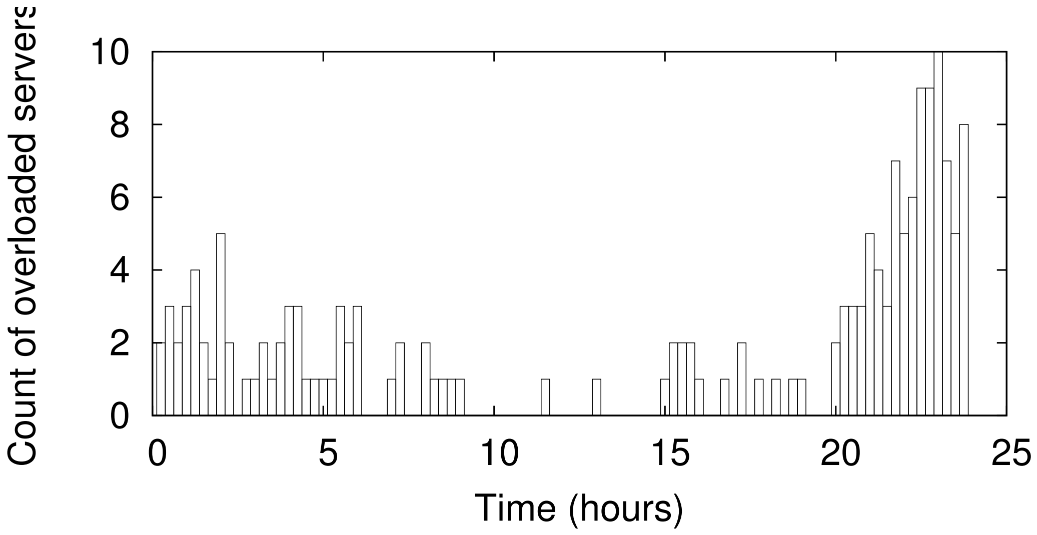


Figure 2.1: Count of simultaneously overloaded servers out of 100 randomly selected servers over a single representative day. Each point represents the number of overloaded servers during the corresponding 15 min. interval.

finest granularity of the log data; thus, sub-fifteen-minute information is not available to us without additional server instrumentation. To capture transient overload bursts that may appear as low paging rates when averaged over the entire fifteen-minute interval, we define overload as an interval with a non-zero paging rate.

We analyzed the data in three different ways. First, we looked at the prevalence of overload (irrespective of its duration) across the 100 servers. We observed that overload is rare: only 28 of the servers experience some kind of memory overload. Second, we studied the frequency of simultaneous overload. Figure 2.1 shows a time series plot of the count of overloaded servers over the 24-hour measurement period. The figure shows that at most 10 servers were simultaneously overloaded. However, the average over the 24-hour period is 1.76

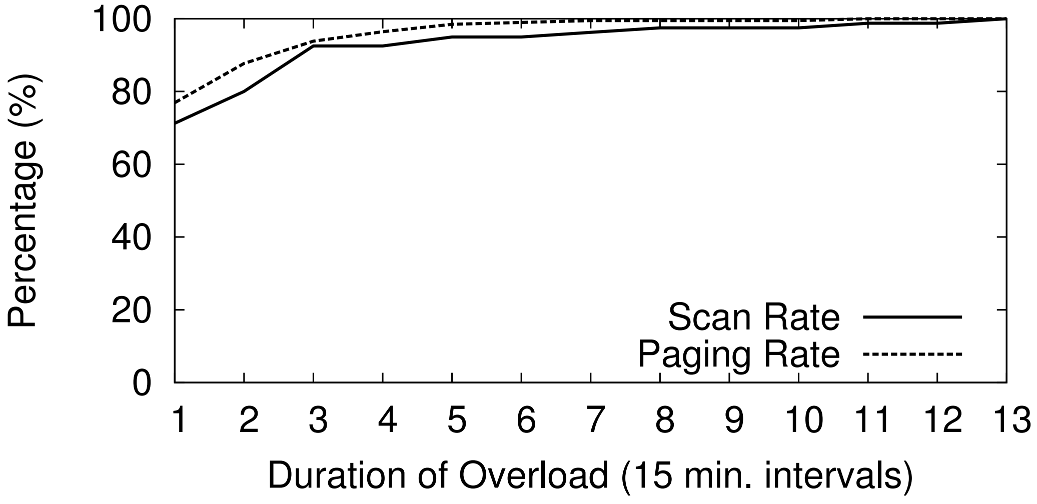


Figure 2.2: Memory overload distribution of 100 randomly selected servers over a single representative day.

servers, suggesting that servers sharing machines are unlikely to experience correlated overload. Finally, we studied the duration of overload. Figure 2.2 shows the cumulative distribution function (CDF) of the duration of memory overload (using both metrics—page rate and scan rate). By definition, the figure only looks at the servers that experienced overload in one or more 15-minute intervals. The figure shows that 71% were overloaded for one interval, 80% (71% + 9%) up to two intervals, 92.5% (71% + 9% + 12.5%) up to 3 intervals (15 min, 30 min, 45 min respectively). In other words, most overloads we studied were transient, but sustained overloads did exist.

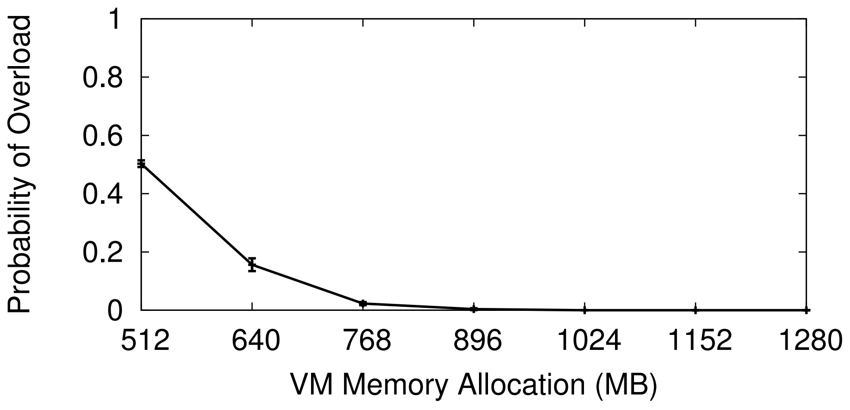
Overload Due to Oversubscription. To safely exploit memory oversubscription, we must understand the characteristics of overload as oversubscription is increased. We would like to analyze real data center logs again, however, we do not have access to traces from a data center that currently employs memory oversubscription.

Instead, we introduce a realistic application and workload in an environment within which we can experiment with different levels of oversubscription and gather fine-grained data at both application and system level. We use the SPECweb2009[4] banking benchmark to run on a LAMP[5] Web sever to provide a realistic client load. SPECweb2009 models each client with an on-off period [150], classified by bursts of activity and long stretches of inactivity. Each client accesses each Web page with a given probability, determined from analyzing trace data from a bank in Texas spanning a period of 2 weeks including 13+ million requests [139]. SPECweb2009 is intended to test server performance with a fixed client load, so, by default, client load is stable: whenever one client exits, another enters the system. This makes the benchmark act like a closed loop system. Real systems rarely experience a static number of clients, so, in order to better approximate real workloads, we use a Poisson process for client arrivals and departures. We choose Poisson processes for the clients as a conservative model; real systems would likely have more unpredictable (and transient) spikes.

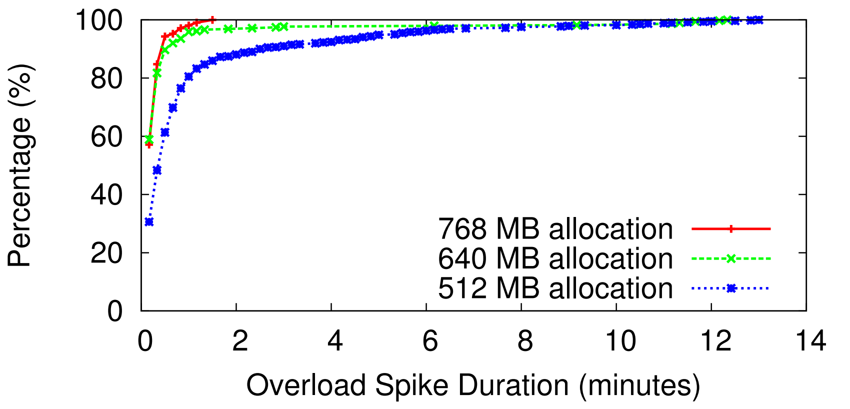
We next examine the effect of oversubscription on the duration of overload. To do so, we varied the VM's memory allocation to simulate oversubscription and ran the SPECweb2009 Web server with Poisson processes for client arrivals and departure set so that the arrival rate is 80% of the service rate. Each experiment lasted for 10 minutes. The measurement granularity within an experiment is 10 seconds. Each point in the graph is the average of 75 experiments.

[4]http://www.spec.org/web2009/

[5]Linux, Apache, MySQL, PHP



(a) Likelihood of overload.



(b) CDF of overload duration

Figure 2.3: These two graphs form a memory overload probability profile for the web server component of the SPECweb2009 banking application under a variety of different oversubscription levels, including both the frequency and duration of overload.

From this experiment, we construct a probability profile for the application VM under different memory allocations. As expected, Figure 2.3(a) shows an increase in the probability of overload as memory becomes constrained. However, in addition to the frequency of overload, we are also interested in the prevalence of each overload duration. Figure 2.3(b) shows a CDF of the duration of overload. We see that even at high memory oversubscription ratios, most overload is transient: 88.1% of overloads are less than 2 minutes long, and 30.6% of overloads are 10 seconds or less for an allocation of 512 MB. If the VM memory is increased to 640 MB, 96.9% of overloads are less than 2 minutes long, and 58.9% of overloads are 10 seconds.

To conclude, there is an opportunity to employ memory oversubscription within enterprise workloads; however, memory overload increases with memory oversubscription. Memory overload is not solely sustained or transient, but covers a spectrum of durations. Even at high oversubscription levels, transient overloads dominate. We design a system to safely oversubscribe memory in an enterprise cloud deployment in Chapter 6.

2.2 Methodology: Experimental Environments and Workloads

In this dissertation, we investigate how large enterprise workloads can efficiently utilize cloud resources from a variety cloud providers. For each challenge, our methodology involves the design, implementation and evaluation of a system. Each system is an instance of an aspect of our user-centric approach and a step towards an instance of a supercloud. The evaluation of each system provides evidence to support the claims made in our approach.

In this chapter, we describe some of the environments within which we evaluate our systems. We also describe several example applications that we use to simulate various aspects of enterprise workloads.

2.2.1 Cloud Environments

We have run our systems on three different environments: Amazon EC2 [29], an enterprise cloud, and a private setup at Cornell. However, to maintain consistency throughout the evaluation, we limit the evaluation to two environments. As described further in Chapters 3 and 4, we employ the use of *nested virtualization*, or one layer of virtualization on top of another. In the rest of this subsection, we describe the environments—both single-layer and nested—that we use throughout this dissertation.

Amazon EC2. Amazon EC2 is a popular public Infrastructure as a Service (IaaS) cloud. VM images must be in the Amazon Machine Image (AMI) format to be able to run on EC2. On EC2, we instantiate VMs in one of three different sizes. *Small* instances have 1.7 GB memory, 1 EC2 Compute Unit, 160 GB instance storage and a 32-bit platform. *Medium* instances have 3.75 GB memory, 2 EC2 Compute Units, 410 GB instance storage and a 64-bit platform. *Cluster Compute Quadruple Extra Large (Cluster 4XL)* instances have 23 GB memory, 33.5 EC2 Compute Units, 1690 GB of local instance storage and a 64-bit platform. Communication speed between VMs varies depending on the instance size. Small and medium instances are classified as "moderate" performance; they can achieve at most 1 Gbps throughput on the network. Cluster 4XL instances are connected to each other using a 10 Gbps Ethernet.

EC2 utilizes a customized version of the Xen hypervisor [41]. Therefore, it uses paravirtualization [41, 152] as a virtualization technique. A brief primer on paravirtualization appears in Appendix A. Both small and medium EC2 instances are both paravirtualized (PV) guests. Cluster 4XL instances are hardware assisted (HVM) guests, also referred to as fully virtualized.

Cornell. We have constructed a testbed at Cornell that uses physical rack mounted servers connected by a 1 Gbps network. Each physical server contains two six-core 2.93 GHz processors Intel Xeon X5670 processors,[6] 24 GB of memory, and four 1 TB disks. The software stack on each machine consists of a 64-bit Xen version 3.1.2 running a 64-bit CentOS 5.5 in the control domain. We can instantiate VMs with a variety of resource—memory and virtual CPU (VCPU)—configurations. Furthermore, in this setup, VMs can be instantiated in paravirtualized (PV) or fully virtualized (HVM) mode. With hardware virtualization support in the Xeon X5670 processor, HVM is expected to outperform PV, because of reduced hypervisor involvement. In particular, the processor includes support for extended page tables (EPT) in which a guest VM can manage its own page tables without involving the hypervisor.

An Enterprise Cloud. Using physical servers with similar specifications to those at Cornell, we have installed a virtualization environment that uses KVM [96] instead of Xen. In this setup, we only instantiate HVM guests.

Nested Environments. We leverage nested virtualization as a mechanism to enable cloud extensibility, as described further in Chapters 3 and 4. We depict the different nested combinations by specifying both the hypervisor (e.g., Xen, KVM) and the type of interface the guest is using (e.g., PV, HVM). For example, Figure 2.4 shows four common environments within which we evaluate our systems. *Native* represents an unmodified Linux [19] running on bare metal.

[6]Hyperthreading causes the OS to perceive 24 processors on the system.

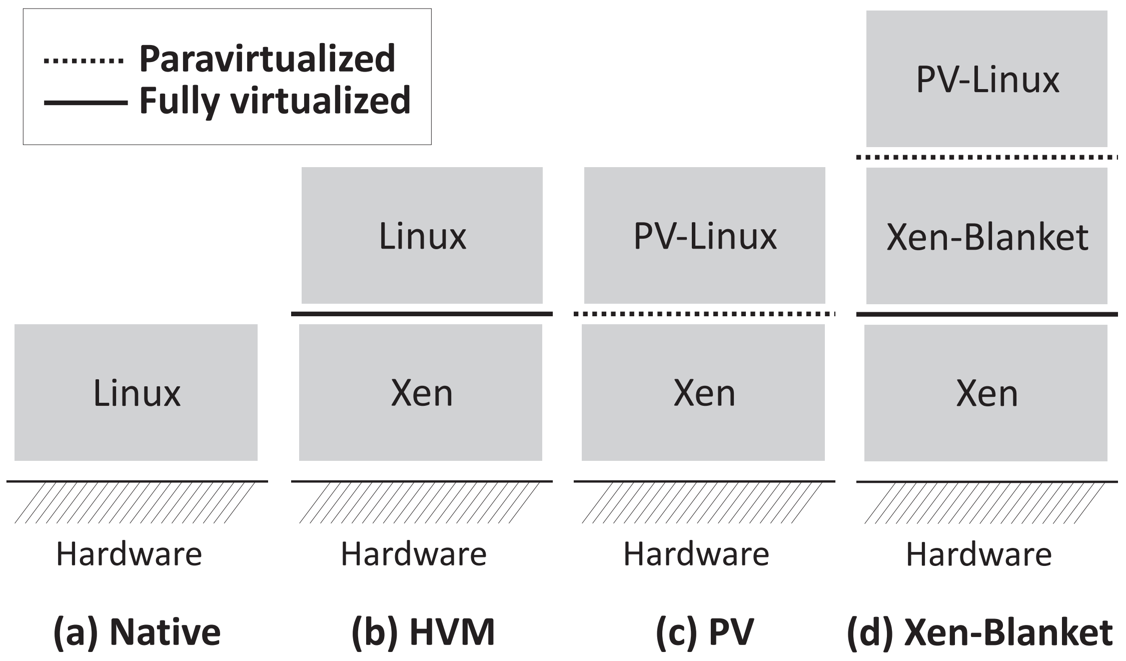


Figure 2.4: Environments with 0, 1, and 2 layers of virtualization

HVM represents a standard single-layer Xen-based virtualization setup using full, hardware-assisted virtualization. *PV* represents a standard single-layer Xen-based virtualization setup using paravirtualization. As further described in Chapter 4, *Xen-Blanket* consists of a paravirtualized virtualization setup inside of an Xen-based HVM setup. We have performed experiments using KVM as the hypervisor with comparable results, but—for the most part—focus on a single hypervisor for a consistent evaluation.

2.2.2 Sample Workloads

We have designed and implemented several systems to support our claims pertaining to the migration of enterprise workloads to the cloud. Enterprise workloads are often sensitive and difficult to obtain for experimentations. Those that may be available are also difficult to evaluate at scale. In this subsection, we describe the common applications and tools we use to simulate particular aspects of enterprise workloads on our systems.

SPECweb2009. As described in the previous section, SPECweb2009[7] is a common 3 tier Web application benchmark. We use the banking application, which consists of a front-end Web server, a scalable application logic tier, and a back-end database. In SPECweb2009, the back-end database is simulated; no data is actually stored. The tiers may be run each in a different VM or all in the same VM. SPECweb2009 generates load for a configurable number of customers, or simultaneous sessions. Using an on-off model [150], classified by bursts of activity and long stretches of inactivity, each customer places load on the applications. The customer pattern of access used in the SPECweb2009 benchmark is based from an analysis of trace data from a bank in Texas spanning a period of 2 weeks including 13+ million requests [139].

Using SPECweb2009, we can simulate an enterprise application, part of an enterprise workload, under different load conditions. By running experiments with steady client load, the performance of the application is classified in terms of request latency. A valid SPECweb2009 run involves 95% of the page requests to compete under a "good" time threshold (2s) and 99% of the requests to be under a "tolerable" time threshold (4s).

RUBiS. Like SPECweb2009, RUBiS [66] is a benchmark with three VMs representing the Web tier, application server tier, and database tier, respectively. Instead of the application being a bank, as in SPECweb2009, RUBiS is an online

[7]http://www.spec.org/web2009/

auction application benchmark. The performance of the application is measured similarly to SPECweb2009. Based on latency, the number of "good requests" (latency within 2000 ms) are calculated under various client loads (simultaneous sessions).

To better represent the network complexity that can arise in enterprise workloads, we add two VMs running software firewalls (`iptables` [12]) between each of the 3 RUBiS tiers. Like many real applications, the VMs each have a hard-coded configuration—route table entries and IP addresses—that complicates their migration to the cloud using traditional methods.

`netperf`. To stress the networking aspects of enterprise workloads, we use `netperf`, a tool that generate Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) traffic to measure either throughput or latency. Using `netperf` we send traffic through complex network paths, including physical and virtual switches, routers, and other network components. UDP traffic is generated with UDP_STREAM or UDP_RR modes for latency measurements. Similarly, TCP traffic is generated with TCP_STREAM or TCP_RR modes. We generally send 1400 byte packets to avoid Maximum Transmission Unit (MTU) issues.

2.3 Summary

Enterprise workloads can be complex both in terms of the applications they involve as well as the infrastructure they are dependent on. First, enterprise workloads require a flexible, uniform environment that spans multiple cloud providers. They may need to implement their own hypervisor-level services that span clouds to facilitate management or migration to the cloud. Second, enterprise workloads have complex network topologies and policies that may be encoded deep inside low-level device configurations. Finally, enterprise workloads contain opportunities for memory oversubscription, but exhibit a range of memory overload behavior. In particular, memory overload may consist of a transient burst or may occur over a sustained period of time.

Enterprise workloads with these characteristics face serious challenges before they can be migrated to the cloud. A user-centric approach grants the enterprise cloud user the control it needs to migrate to the cloud, without costly redesign. In the remainder of this dissertation, we describe the design, implementation and evaluation of the systems we used to validate our user-centric supercloud approach, using the environments and under the workloads described in this chapter.

CHAPTER 3

CLOUD EXTENSIBILITY: AN ENABLING ABSTRACTION

"Nature is a mutable cloud which is always and never the same."

—Ralph Waldo Emerson

Whereas Infrastructure as a Service (IaaS) clouds once provided a simple bare-bones VM abstraction, they have now evolved to include diverse, feature-rich offerings. On the surface, this is advantageous: cloud users on Amazon EC2, for example, enjoy tools such as CloudWatch (integrated monitoring), AutoScaling, and Elastic Load Balancing. Beneath the surface, however, users are constrained: cloud provider features are rapidly becoming synonymous with vendor lock-in [32, 38], which is a symptom of a larger problem. Users are completely dependent on the provider for any hypervisor-level features. Tools and techniques at the hypervisor-level—enabling increased portability, availability [61], security [65], efficiency [154] and performance [92]—are impossible for cloud users to implement themselves. As a direct consequence, cloud users cannot build superclouds, or user-defined cloud environments, on existing clouds.

The current state of clouds resembles a point in the evolution of operating system (OS) kernels. In particular, extensible systems (such as exokernels [67], SPIN [45], and VINO [135]) emerged to solve certain limitations in monolithic kernels. These systems were motivated, in part, by applications' inability to define their own tailored hardware abstractions, just as applications on existing clouds are unable to define their own virtual and physical hardware abstractions.

In this chapter, we argue that the abstraction of an extensible cloud is essential for building superclouds. At the same time, we point out that the deployment of an extensible cloud must not depend on support from cloud providers. For example, current calls for standardization across multiple cloud providers may never be implemented due to social and legal challenges. In this dissertation, we focus on the technical challenges and related scientific contributions.

3.1 The Existing Cloud Abstraction

The uses of clouds are extensive: users range from a person deploying a single VM to an entire information technology (IT) department or enterprise deploying hundreds or thousands of VMs. A rich array of services, third-party cloud management tools [58], and middleware [51,109] operate at the VM-level to provide useful high-level functionality to cloud users. However, beneath this superficial VM-level veneer, the deployment of efficient, portable, innovative applications—especially by large enterprise cloud users attempting to efficiently manage a workload that spans clouds—is being hindered by two main shortcomings that manifest as a lack of control: immutable hypervisors and buried hardware.

Immutable Hypervisors: By immutable hypervisors, we mean that the user cannot change the hypervisors. The hypervisor, or virtual machine monitor (VMM), in existing clouds is controlled by the provider, leaving users with little or no say as to what hypervisor-level functionality is implemented or exposed. For example, no cloud currently exists with a hypervisor that allows users to

Abstraction Level	Feature
Existing Clouds	Application monitoring Auto-scaling Non-live migration
Mutable Hypervisor	Page sharing [81, 145] Overdriver [154] Revirt [65] Remus [61] Live migration [57] Cross-provider live migration
Exposed Hardware	vSnoop [92] Superpages [121] Page coloring [93] Non fate-sharing Unsupported paravirtualization

Table 3.1: Cloud abstractions and extensions they enable

maximally utilize their VMs through techniques like page sharing [81, 145] or aggressive oversubscription [154]. Live VM migration [57] between multiple clouds—public or private—is virtually impossible. Innovative hypervisor-level techniques for high availability [61] or intrusion detection [65] are unavailable, while further customization and experimentation at this level is stifled.

Buried Hardware: Existing clouds bury the details of hardware beneath a virtual machine abstraction. Users must depend on the provider to expose everything from efficient I/O interfaces to physical fate-sharing information. Moreover, users cannot implement hardware-dependent tricks to squeeze the best performance out of the rented resources. Time-sensitive tasks, such as TCP acknowledgment [92], are difficult. Superpage [121] utilization is not efficient on virtual memory that may not be contiguous, and performance opportunities like page coloring [93] are also lost.

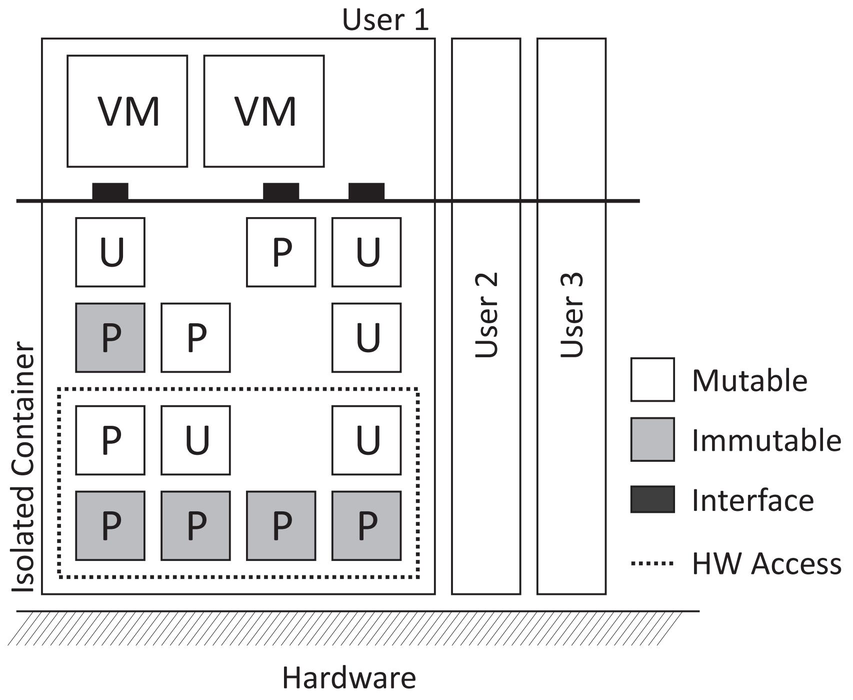


Figure 3.1: General extensibility architecture. **U** and **P** denote user and provider installed modules, respectively.

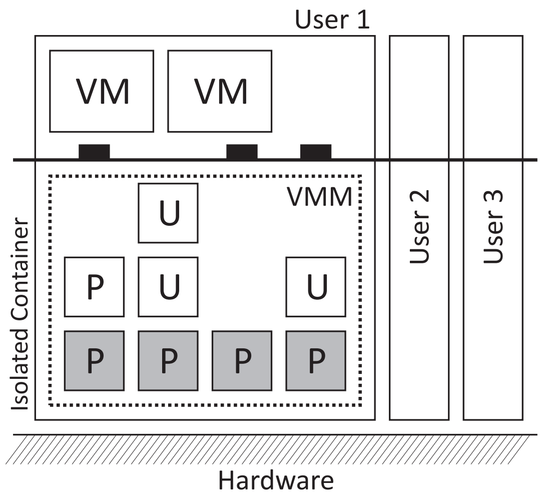
Table 3.1 provides examples of cloud extensions, some of which may or may not be currently available. However, there is a large set of features—spanning performance, security, and portability—that require a mutable hypervisor. That is, some features require users to modify the hypervisor. A further set of performance-related features require control or visibility at the hardware level. It is important to note that despite advocating for mutability, we believe that IaaS clouds should continue to provide a VM abstraction. This approach is fundamentally different from that of cloud operating systems that expose an OS process instead of VM abstraction [151].

3.2 The Design Space for Extensible Clouds

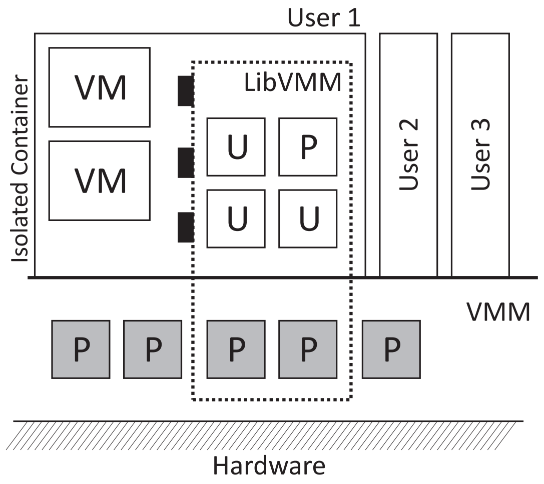
The general components of an extensible cloud are shown in Figure 3.1. Like existing IaaS clouds, an extensible cloud ultimately exposes a VM-like interface, upon which cloud users can run VMs. Unlike existing clouds, the VM-like interface can be customized and user-defined hypervisor-level functionality can be introduced to directly interface with the hardware. Thus, an extensible cloud exposes both a mutable hypervisor and the underlying hardware.

We consider the hypervisor to be made up of a number of modules that interact to make up the inner workings of the cloud provider. The provider likely implements modules that multiplex hardware and enforce protection, such as isolated containers that protect cloud users from one another (Figure 3.1). Modules that implement functionality essential to the operation of the cloud, such as protection and accounting, are immutable; users cannot modify them. Some modules may be modified by users and are therefore mutable; others, implementing innovative or experimental interfaces, may be supplied by the cloud user. For example, the cloud user may modify a VM migration module to ignore temporary state or may implement a networking module from scratch that enables layer-2 connectivity. Modules depicted within a dotted box require access to the hardware.

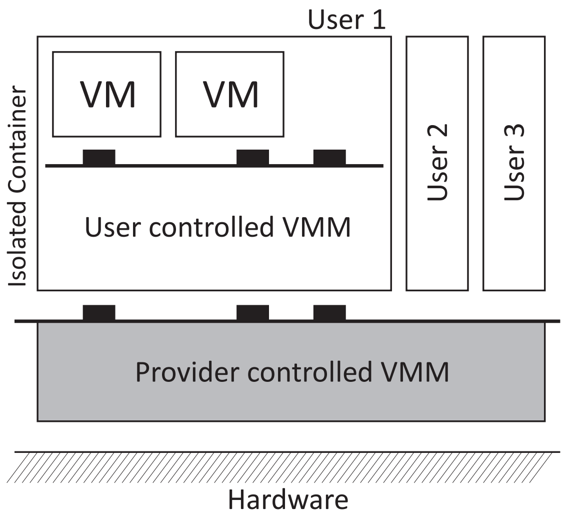
There are a number of design alternatives for arranging the components in Figure 3.1. These alternatives, shown in Figure 3.2, are largely inspired by seminal work on extensible kernels.



(a) Download extension modules into VMM



(b) Export hardware through VMM



(c) Add another VMM

Figure 3.2: Three design alternatives for extensible clouds. The shading scheme is identical to Figure 3.1.

Download Extensions into the VMM: SPIN [45] and VINO [135] are two systems from the 1990's in which extensions, or grafts, can be downloaded into the kernel, and run safely. Safety is provided mostly at the language level, using techniques like safe languages (Modula-3) and software fault isolation. An extensible cloud architecture that adopts this design is shown in Figure 3.2(a). The hypervisor becomes mutable by allowing user-defined or user-modified modules to be downloaded into the kernel. Similar to the extensible OSs, this must be done safely, such that other modules, especially immutable provider modules, are protected. Since modules are executing in the hypervisor with privilege, they can be granted direct access to the hardware.

Expose Hardware through the VMM: Exokernel [67] is a system that achieves extensibility by exposing hardware directly to applications, to the extent that the actual hardware names and addresses are visible to applications. Management of the hardware, traditionally done by the OS kernel, is performed by a library OS (libOS) that can be completely custom-built and linked into the application. The kernel, on the other hand, only enforces protection between applications, which can be complex [91]. With the renewed interest in virtualization in the early 2000's, paravirtualization revisited many of these ideas, with the Denali isolation kernel [152] exposing much of the hardware, and implementing the traditional OS as a library, linked into the application. Xen [41] also adopted a paravirtualization approach and argued that full virtualization is not desirable when a guest OS needs to see real physical resources. Figure 3.2(b) shows a design in which the cloud provider exposes hardware to a "libVMM" under the control of a cloud user—analogous to a libOS on top of an Exokernel. The hardware is not buried, but exposed to the libVMM, which is completely mutable.

Add Another VMM: Interest in nested virtualization is increasing as virtualization becomes ever more ubiquitous. Furthermore, nested virtualization has been shown to perform well; for example, the Turtles Project [43] has achieved performance within 6–8% of single-level virtualization for some workloads. Figure 3.2(c) shows how nested virtualization can be leveraged for extensibility in the cloud. When a user leases a VM instance, it installs a mutable, second layer hypervisor to run on top of the cloud provider's VM abstraction. The providers' and users' modules are implemented in the first and second layer hypervisors, respectively. It should be noted that the techniques used by the Turtles Project require modifications to the bottom-most hypervisor in order to expose virtualization hardware extensions to VMs.

Nested virtualization fundamentally differs from the previous two design alternatives. On the one hand, using nested virtualization, the hardware remains buried under the virtual machine abstraction, preventing a cloud user from implementing the class of performance-enhancements described in Section 3.1. On the other hand, nested virtualization has enormous potential for incremental deployment, even without provider cooperation. As an alternative to existing nested virtualization systems that require lower-layer VMM modifications, techniques such as paravirtualization can be applied inside a standard fully virtualized or even paravirtualized guest VM. As such, this mutable, second layer hypervisor implements an extensible cloud that can be deployed on existing cloud platforms without requiring provider involvement. Nested virtualization offers a compelling extensible cloud architecture. However, can such an architecture perform well?

3.3 Will a Deployable Extensible Cloud Perform?

As discussed above, the nested virtualization approach has the advantage of rapid deployment on existing clouds. In many cases, nested virtualization can

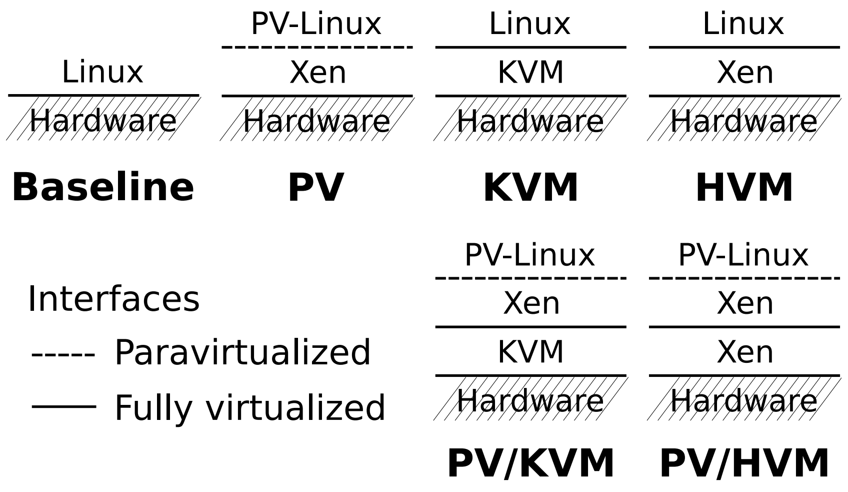


Figure 3.3: Virtualization configurations

achieve reasonable performance without provider support or involvement.

For the following experiments, we assume standard, hardware-assisted hypervisors that run at the lowest layer, similar to the fully virtualized (HVM) instances available from Amazon EC2. Existing clouds do not contain the hypervisor extensions required to virtualize the processor features supporting virtualization [43]. Instead, we must use other techniques to implement the second layer hypervisor, such as paravirtualization (e.g., Xen [41]), binary translation (e.g., VMWare [140]), or full emulation (e.g., QEMU).[1] Our experiments focus on using paravirtualized Xen,[2] and were performed at Cornell on our machines with 24 GB of RAM and dual 6-core 2.93 GHz Intel Xeon X5670 processors. The virtualization configurations that we compare are described in Figure 3.3.

Table 3.2 shows the results of some `lmbench` [111] microbenchmarks over the various setups (Figure 3.3). As expected, all arithmetic operations, like dou-

[1]Results from experimentation with QEMU are not shown due to the poor performance of full emulation, which was up to two orders of magnitude slower.

[2]While Xen paravirtualization somewhat limits compatibility, we note that it is very popular; for instance, a large fraction of Amazon EC2's offerings are indeed paravirtualized.

		Single			Nested	
	Baseline	PV	HVM	KVM	PV / HVM	PV / KVM
double div (ns)	7.19	7.55	7.61	7.41	7.57	7.35
null call (μs)	.19	.37	.21	.20	.37	.38
fork proc (μs)	65.17	249.70	78.89	86.52	280.39	336.93

Table 3.2: Microbenchmarks using `lmbench`

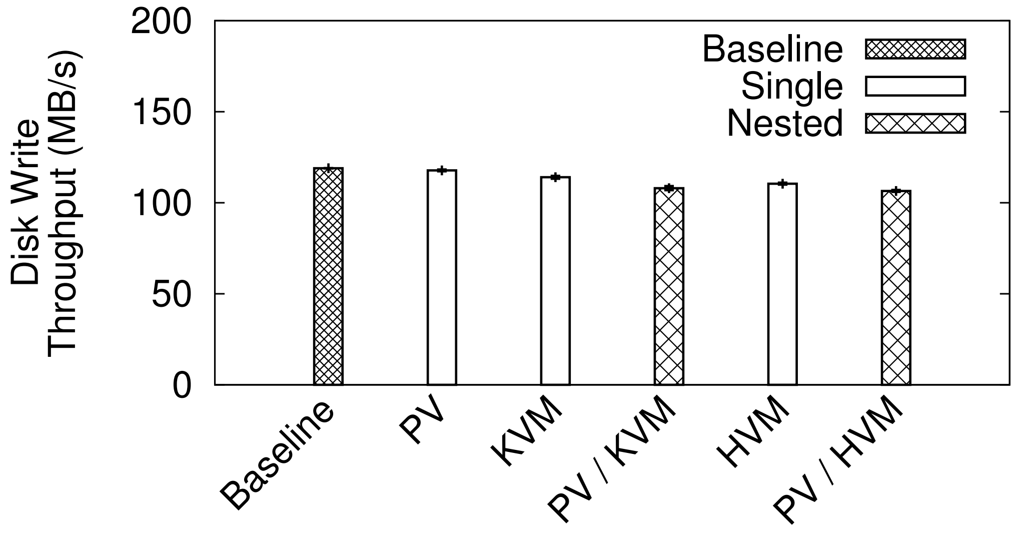


Figure 3.4: Disk throughput using `dd`

ble division (shown), are minimally affected by virtualization. Simple operations like null system calls, are achieved in fully virtualized configurations without hypervisor support, and thus have matching performance. Paravirtualization, on the other hand, invokes the hypervisor on the system call, which must be redirected back up to the guest OS. Nesting does not introduce any extra overhead beyond that of paravirtualization (PV). However, in nested environments, process fork generates 12–30% additional overhead over PV by inducing traps into the lowest layer hypervisor.

I/O typically stresses a virtualized system because of the inefficiencies of

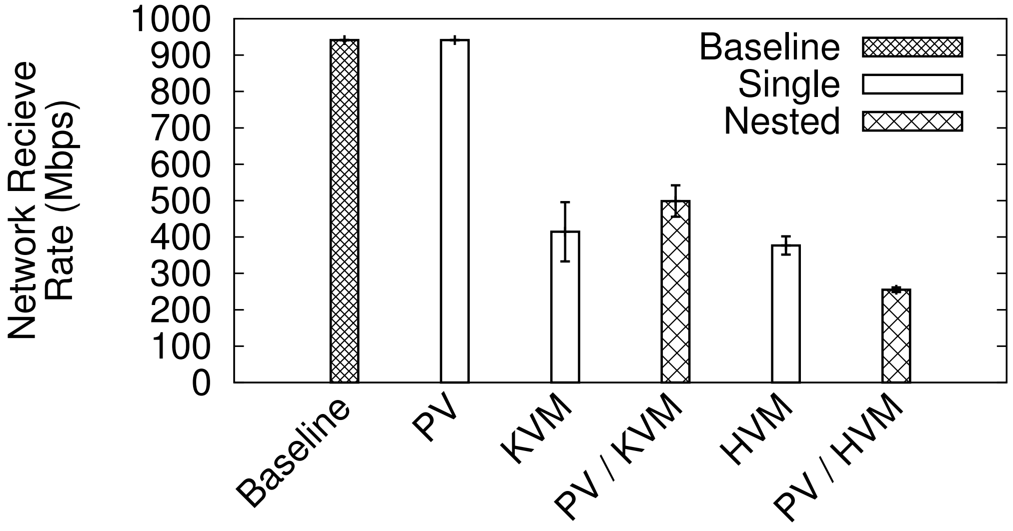
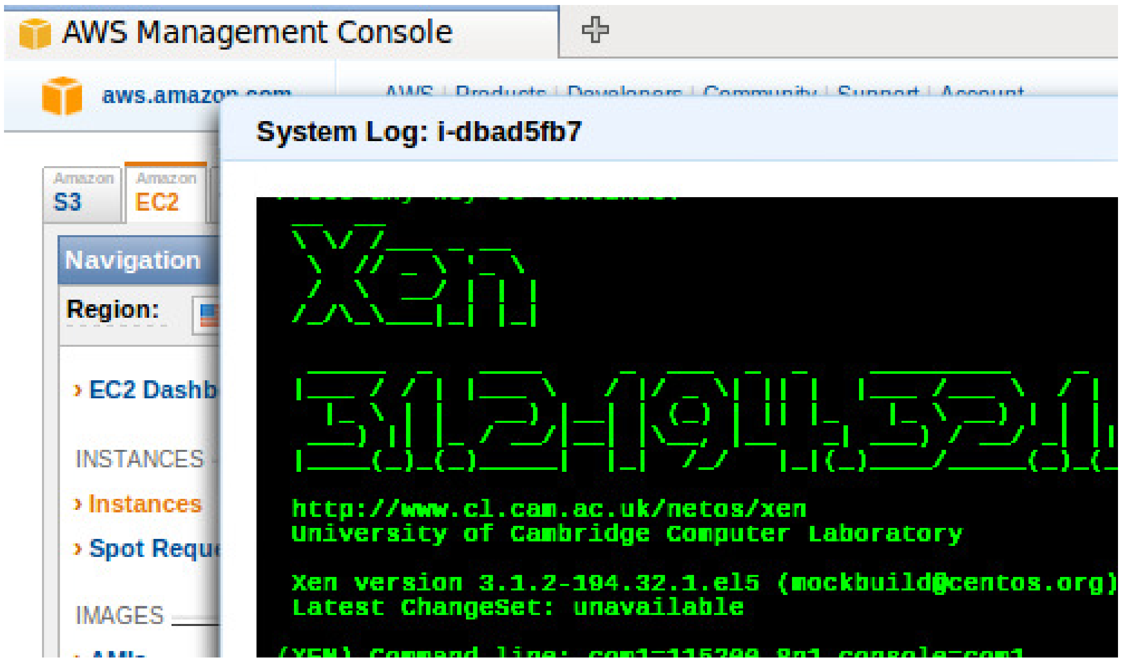


Figure 3.5: Network receive throughput using `netperf`

emulation required to fully virtualize I/O devices and handle interrupts. To determine the overhead introduced by the second-layer hypervisor, we conducted experiments with disk I/O and network I/O. In the first experiment, we measure the performance of disk I/O by writing 1.6 GB of data to a disk partition with the standard Unix tool `dd` using blocks of size 256 K. Each result is the average of 5 trials. Figure 3.4 shows the throughput of the disk. We find that, for disk I/O, nested virtualization does not cause significant overhead, achieving 90% of native throughput, largely due to caching.

In the second experiment, we measure the performance of network I/O. The network device typically generates more interrupts and requires more OS interaction than a disk, making it a more stressful test for virtualized environments. We ran `netperf` in each setup in order to determine how fast a guest could receive TCP network traffic generated from another machine across a 1 Gbps network. Each result is the average of 10 trials. Figure 3.5 shows the results.



```
http://www.cl.cam.ac.uk/netos/xen
University of Cambridge Computer Laboratory

Xen version 3.1.2-194.32.1.el5 (mockbuild@centos.org)
Latest ChangeSet: unavailable
```

Figure 3.6: Screenshot of Xen booting on Amazon EC2

Considering a single layer of virtualization, it is immediately obvious that full virtualization (HVM, KVM) incurs a dramatic performance hit. HVM performs particularly poorly, achieving only 40% of the baseline throughput.[3] PV, which bypasses device emulation by using paravirtualization, achieves performance matching the baseline. This suggests that the poor performance of the nested setups is largely due to the device emulation of the first layer hypervisor. For instance, whereas the single layer of KVM virtualization reduced throughput by 47%, the second layer only reduces it by a further 9%.

[3]This is a known limitation of network virtualization in Xen due to inefficient I/O remapping [114].

3.4 Summary

Cloud providers are diverse and complex and lend themselves toward vendor lock-in. Users cannot implement sophisticated, custom applications that require VMM- and HW-level control. In particular, cloud users cannot build superclouds. In this chapter, we discussed cloud extensibility as an approach to address these limitations.

Transforming existing clouds into extensible clouds is within reach using nested virtualization as an extensibility layer. In experiments, we showed that nested virtualization overhead is within acceptable limits given its relative ease of deployment. However, we also discovered that paravirtualized I/O drivers, especially network drivers, are essential to eliminate the bottleneck presented by device emulation.

In the following chapters, we show how to overcome I/O bottlenecks and demonstrate an extensible cloud deployed on existing clouds, including Amazon EC2 (Figure 3.6), an enterprise cloud, and private cloud infrastructure at Cornell. We leverage extensibility to build superclouds; in particular, extensibility enables the homogenization, support for enterprise networks, and efficiency through oversubscription on existing clouds.

CHAPTER 4

TOWARDS CLOUD INTEROPERABILITY: THE XEN-BLANKET

To create superclouds, it is crucially important that a VM image, and subsequently a VM instance, can run—unmodified—on any cloud. Furthermore, hypervisor-level features, such as VM monitoring, must be available on all clouds. In this chapter, we investigate cloud extensibility and use it to create a uniform cloud environment across clouds. In other words, we use cloud extensibility to homogenize multiple clouds as the first step towards superclouds.

Instead of relying on cloud providers to change their environments, we explore a *user-centric* approach, in which users are able to run their unmodified VMs on any cloud without any special provider support. Towards this goal, we present the Xen-Blanket, a system that transforms existing heterogeneous clouds into a uniform user-centric homogeneous offering. The Xen-Blanket consists of a second-layer hypervisor that runs as a guest inside a VM instance on a variety of public or private clouds, forming a *Blanket layer*. The Blanket layer exposes a homogeneous interface to second-layer guest VMs, called *Blanket guests*, and is completely user-centric and customizable. The Xen-Blanket, therefore, is an instantiation of an extensible cloud. Inside the Blanket layer, users can implement hypervisor-level techniques and management tools, like VM migration, page sharing, and oversubscription. Meanwhile, the Blanket layer contains *Blanket drivers* that allow it to run on heterogeneous clouds while hiding interface details of the underlying clouds from Blanket guests.

Existing nested virtualization techniques (like the Turtles project [43]) focus on an efficient use of hardware virtualization primitives by both layers of virtualization. This requires the underlying hypervisor—controlled by the cloud provider—to expose hardware primitives, like Intel's VMX [103] and AMD SVM [160]. At the time of writing this dissertation, no publicly available cloud currently offers such primitives. In contrast, the Xen-Blanket can be deployed on existing third-party clouds, requiring no special support, and instead using software techniques for the Blanket layer virtualization. Thus, the contributions of the Xen-Blanket are fundamentally different: the Xen-Blanket enables competition and innovation for products that span multiple clouds, whether support is offered from cloud providers or not. Further, the Xen-Blanket enables the use of unsupported features such as oversubscription, CPU bursting, VM migration, and many others.

The Xen-Blanket has been deployed on both Xen-based and KVM-based hypervisors, on public and private infrastructures within Amazon EC2, a separate enterprise cloud, and Cornell University. The Xen-Blanket has successfully homogenized these diverse environments. For instance, we have migrated VMs to and from Amazon EC2 with no modifications to the VMs. Furthermore, the user-centric design of the Xen-Blanket affords users the flexibility to oversubscribe resources such as network, memory, and disk. As a direct result, a Xen-Blanket image on EC2 can host 40 CPU-intensive VMs for 47% of the price per hour of 40 small instances with matching performance. Blanket drivers achieve good performance: network drivers can receive packets at line speed on a 1 Gbps link, while disk I/O throughput is within 12% of single level paravirtualized disk performance. Despite overheads of up to 68% for some benchmarks, Web server macrobenchmarks can match the performance of single level virtualization (i.e., both are able to serve an excess of 1000 simultaneous clients) while increasing CPU utilization by only 1%.

In this chapter, we make four main contributions:

- We describe how user-centric homogeneity can be achieved at the hypervisor level to enable multi-cloud deployments, even without any provider support.
- We enumerate key extensions to Xen, including a set of *Blanket drivers* and hypervisor optimizations, that transform Xen into an efficient, homogenizing Blanket layer on top of existing clouds, such as Amazon EC2.
- We demonstrate how the Xen-Blanket can provide an opportunity for substantial cost savings by enabling users to oversubscribe their leased resources.
- We discuss our experience using hypervisor-level operations that were previously impossible to implement in public clouds, including live VM migration between an enterprise cloud and Amazon EC2.

This chapter is organized as follows. Section 4.1 introduces the concept of a Blanket layer, and describes how the Xen-Blanket provides a user-centric homogenized layer, with the implementation details of the enabling Blanket drivers in Section 4.2. Some overheads and advantages of the Xen-Blanket are quantified in Section 4.3, while qualitative practical experience is described in Section 4.4. The chapter is summarized in Section 4.5.

4.1 The Xen-Blanket

The Xen-Blanket leverages nested virtualization to form a *Blanket layer*, or a second layer of virtualization software that provides a user-centric homogeneous

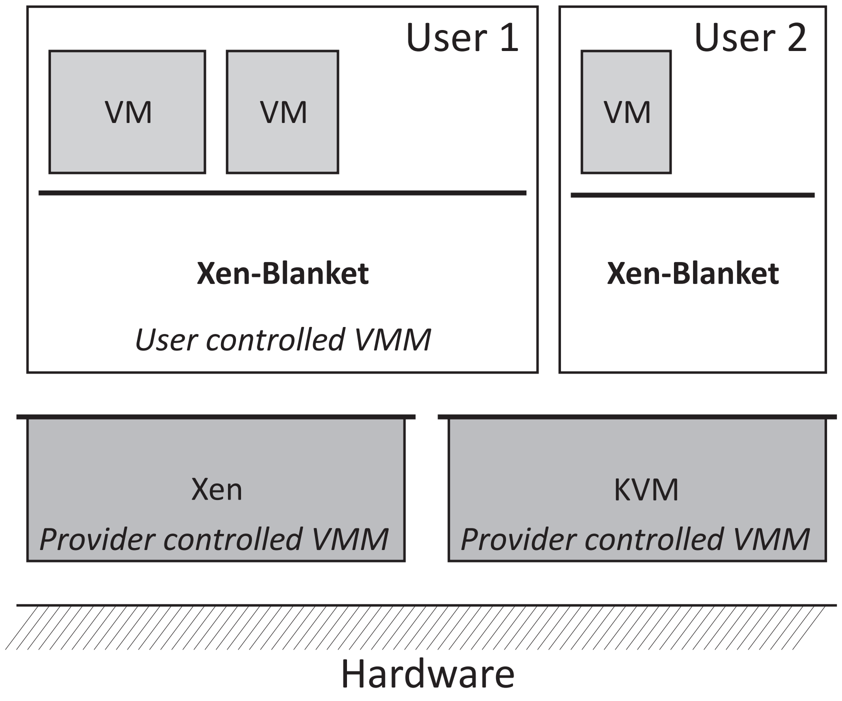


Figure 4.1: The Xen-Blanket, completely controlled by the user, provides a homogenization layer across heterogeneous cloud providers without requiring any additional support from the providers.

cloud interface, as depicted in Figure 4.1. A Blanket layer embodies three important concepts. First, the *bottom half* of the Blanket layer communicates with a variety of underlying hypervisor interfaces. No modifications are expected or required to the underlying hypervisor. Second, the *top half* of the Blanket layer exposes a single VM interface to Blanket (second-layer) guests such that a single guest image can run on any cloud without modifications. Third, the Blanket layer is completely under the control of the user, so functionality typically implemented by providers in the hypervisor, such as live VM migration, can be implemented in the Blanket layer.

The bottom half of the Xen-Blanket ensures that the Xen-Blanket can run across a number of different clouds without requiring changes to the underlying cloud system or hypervisor. The bottom half is trivial if the following two assumptions hold on all underlying clouds. First, if device I/O is emulated, which means the hypervisor exposes an interface to virtual devices identical to physical devices, then the Blanket hypervisor does not need to be aware any provider-specific device interfaces. Second, if hardware-assisted full virtualization for x86 (called HVM in Xen terminology) is available, then the Blanket hypervisor can run unmodified. However, these assumptions limit the number of clouds that the Blanket layer can cover; for example, we are not aware of any public cloud that satisfies both assumptions.

The Xen-Blanket relaxes the emulated device assumption by interfacing with a variety of underlying cloud device I/O interfaces, each of which use a virtualization technique called paravirtualization (See Appendices A and B for further detail). Paravirtualized device I/O has proved essential for performance and is required by some clouds, such as Amazon EC2. However, there is currently no standard paravirtualized device I/O interface. For example, Xen-based clouds, including Amazon EC2, require device drivers to communicate with Xen-specific subsystems[1] (the details of Xen appear in Appendix B), whereas KVM-based systems expect device drivers to interact with the hypervisor through `virtio` interfaces. The Xen-Blanket supports such non-standard interfaces by modifying the bottom half to contain cloud-specific *Blanket drivers*.

On the other hand, the Xen-Blanket does rely on support for hardware-assisted full virtualization for x86 on all clouds. Currently, this assumption somewhat limits deployment opportunities. For example, a large fraction of both Amazon EC2 and Rackspace instances expose paravirtualized, not HVM interfaces, with Amazon EC2 only offering an HVM interface to Linux guests

[1]Since 2011, Xen has supported a `virtio` device interface [125], but as of 2012, no Xen-based cloud providers we are aware of support `virtio`.

in 4XL-sized cluster instances. EC2 does, however, expose an HVM interface to other sized instances running Windows, which we believe can also be converted to deploy the Xen-Blanket. Further efforts to relax the HVM assumption are discussed as future work in Chapter 8.3.2.

The top half of the Blanket layer exposes a consistent VM interface to (Blanket) guests. Guest VMs therefore do not need any modifications in order to run on a number of different clouds. In order to maximize the number of clouds that the Xen-Blanket can run on, the top half of the Xen-Blanket does not depend on state of the art nested virtualization interfaces (e.g., the Turtles Project [43], Graf and Roedel [74]). The Xen-Blanket instead relies on other x86 virtualization techniques, such as paravirtualization or binary translation. For our prototype Blanket layer implementation we chose to adopt the popular open-source Xen hypervisor, which uses paravirtualization techniques when virtualization hardware is not available. The Xen-Blanket subsequently inherits the limitations of paravirtualization, most notably the inability to run unmodified operating systems, such as Microsoft Windows.[2] However, this limitation is not fundamental. A Blanket layer can be constructed using binary translation (e.g., a VMWare [140]-Blanket), upon which unmodified operating systems would be able to run. Blanket layers can also be created with other interfaces, such as Denali [152], alternate branches of Xen, or even customized hypervisors developed from scratch.

The Xen-Blanket inherits services that are traditionally located in the hypervisor or privileged management domains and allows the user to run or modify them. For instance, users can co-locate VMs [159] on a single Xen-Blanket

[2]Despite the limitations of paravirtualization and the increasingly superior performance of hardware assisted virtualization, paravirtualization remains popular. Many cloud providers, including Amazon EC2 and Rackspace, continue to offer paravirtualized Linux instances.

instance, share memory pages between co-located VMs [81, 145], and oversubscribe resources [154]. If Xen-Blanket instances on different clouds can communicate with each other, live VM migration or high availability [61] across clouds become possible.

4.2 Blanket Drivers

The Xen-Blanket contains Blanket drivers for each of the heterogeneous interfaces exposed by existing clouds. In practice, the drivers that must be implemented are limited to dealing with paravirtualized device interfaces for network and disk I/O. As described in Section 4.1, Blanket drivers reside in the bottom half of the Xen-Blanket and are treated by the rest of the Xen-Blanket as drivers interacting with physical hardware devices. These "devices" are subsequently exposed to guests through a consistent paravirtualized device interface, regardless of which set of Blanket drivers was instantiated.

This section is organized as follows: we present background on how paravirtualized devices work on existing clouds. Then, we describe the detailed design and implementation of Blanket drivers. Finally, we conclude with a discussion of hypervisor optimizations for the Xen-Blanket and a discussion of the implications of evolving virtualization support in hardware and software.

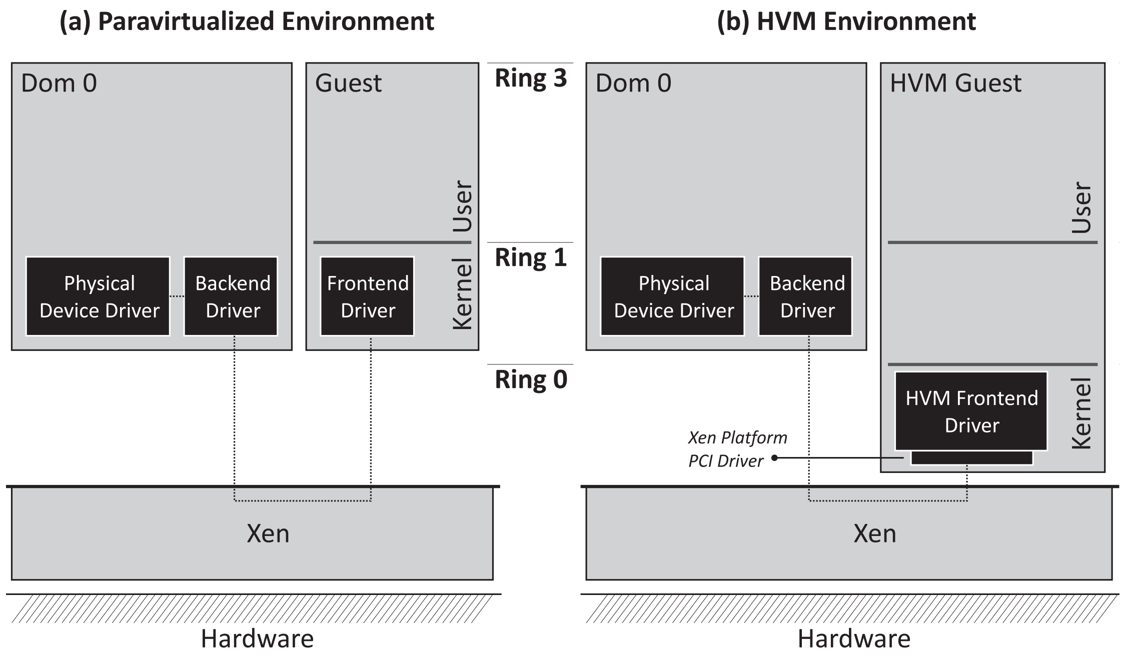


Figure 4.2: Guests using paravirtualized devices implement a front-end driver that communicates with a back-end driver (a). In HVM environments, a Xen Platform PCI driver is required to set up communication with the back-end (b). The Xen-Blanket modifies the HVM front-end driver to become a *Blanket driver*, which, with support of *Blanket hypercalls*, runs in hardware protection ring 1, instead of ring 0 (c).

4.2.1 Background

To understand Blanket drivers, we first give some background as to how paravirtualized device drivers work in Xen-based systems.[3] First, we describe device drivers in a fully paravirtualized Xen, depicted in Figure 4.2(a). The Xen-Blanket uses paravirtualization techniques in the Blanket hypervisor to provide guests with a homogeneous interface to devices. Then, we describe paravirtualized device drivers for hardware assisted Xen (depicted in Figure 4.2(b)), an underlying hypervisor upon which the Xen-Blanket successfully runs.

[3]A discussion of the paravirtualized drivers on KVM, which are similar, is postponed to the end of Section 4.2.2.

Xen does not contain any physical device drivers itself; instead, it relies on device drivers in the operating system of a privileged guest VM, called Domain 0, to communicate with the physical devices. The operating system in Domain 0 multiplexes devices, and offers a paravirtualized device interface to guest VMs. The paravirtualized device interface follows a *split driver* architecture, where the guest runs a *front-end* driver that is paired with a *back-end* driver in Domain 0. Communication between the front-end and back-end driver is accomplished through shared memory ring buffers and an event mechanism provided by Xen. Both the guest and Domain 0 communicate with Xen to set up these communication channels.

In hardware assisted Xen, or HVM Xen, paravirtualized device drivers are called PV-on-HVM drivers. Unlike paravirtualized Xen, guests on HVM Xen can run unmodified, so by default, communication channels with Xen are not initialized. HVM Xen exposes a *Xen platform Peripheral Component Interconnect (PCI) device*, which acts as a familiar environment wherein shared memory pages are used to communicate with Xen and an interrupt request (IRQ) line is used to deliver events from Xen. So, in addition to a front-end driver for each type of device (e.g. network, disk), an HVM Xen guest also contains a Xen platform PCI device driver. The front-end drivers and the Xen platform PCI driver are the only Xen-aware modules in the HVM guest.

4.2.2 Design & Implementation

The Xen-Blanket consists of a paravirtualized Xen inside of either a HVM Xen or KVM guest. We will center the discussion around Blanket drivers for Xen, and discuss the conceptually similar Blanket drivers for KVM at the end of this subsection. Figure 4.2(c) shows components of Blanket drivers. The Blanket layer contains both a Xen hypervisor as well as a privileged *Blanket Domain 0*. Guest VMs are run on top of the Blanket layer, each containing standard paravirtualized front-end device drivers. The Blanket Domain 0 runs the corresponding standard back-end device drivers. The back-end drivers are multiplexed into the Blanket drivers, which act as set of front-end drivers for the underlying hypervisors.

There are two key implementation issues that prohibit standard PV-on-HVM front-end drivers from acting as Blanket drivers.[4] First, the Xen hypercalls required to bootstrap a PV-on-HVM PCI platform device cannot be performed from the Blanket Domain 0 hosting the Blanket drivers because the Blanket Domain 0 does not run with the expected privilege level of an HVM guest OS. Second, the notion of a physical address in the Blanket Domain 0 is not the same as the notion of a physical address in a native HVM guest OS.

Performing Hypercalls

Typically, the Xen hypervisor proper runs in hardware protection ring 0, while Domain 0 and other paravirtualized guests run their OS in ring 1 with user spaces in ring 3. HVM guests, on the other hand, are designed to run unmodified, and can use non-root mode from the hardware virtualization extensions to run the guest OS in ring 0 and user space in ring 3. In the Xen-Blanket, in non-root mode, the Blanket Xen hypervisor proper runs in ring 0, while the Blanket

[4]Our implementation also required renaming of some global variables and functions to avoid namespace collisions with the second-layer Xen when trying to communicate with the bottom-layer Xen.

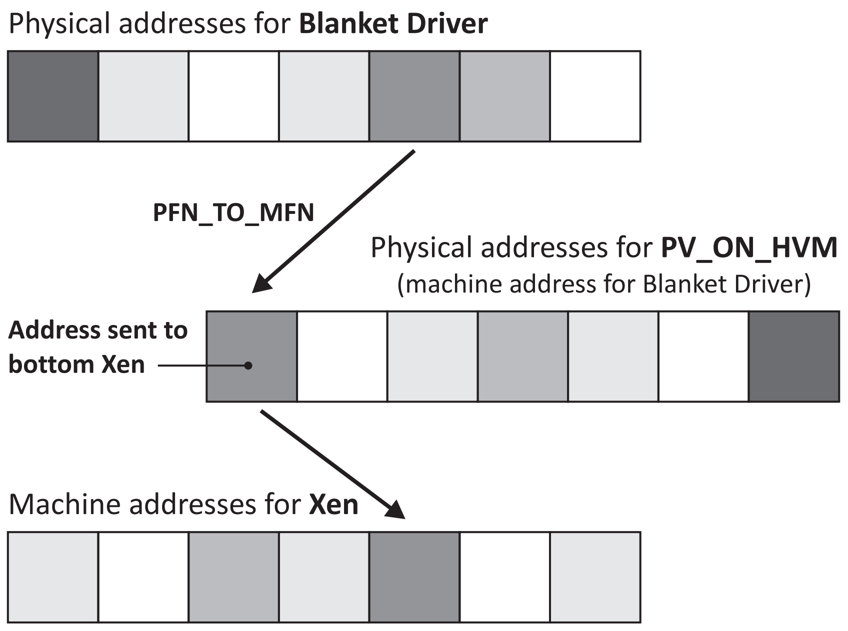

Physical addresses for **Blanket Driver**
PFN_TO_MFN
Physical addresses for **PV_ON_HVM**
(machine address for Blanket Driver)
Address sent to bottom Xen
Machine addresses for **Xen**

Figure 4.3: The PV-on-HVM drivers can send physical addresses to the underlying Xen, whereas the Blanket drivers must first convert physical addresses to machine addresses.

Domain 0 runs in ring 1, and user space runs in ring 3 (Figure 4.2(c)).

In normal PV-on-HVM drivers, hypercalls, in particular `vmcall` instructions, are issued from the OS in ring 0. In the Xen-Blanket, however, Blanket drivers run in the OS of the Blanket Domain 0 in ring 1. The `vmcall` instruction must be issued from ring 0. We overcome this by augmenting the second-layer Xen to contain *Blanket hypercalls* that issue their own hypercalls to the underlying Xen on behalf of the Blanket Domain 0.

Physical Address Translation

Guest OSs running on top of paravirtualized Xen, including Domain 0, have a notion of physical frame numbers (PFNs). The PFNs may or may not match the actual physical frame numbers of the machine, called machine frame numbers

(MFNs). The relationship between these addresses is shown in Figure 4.3. However, the guest can access the mapping between PFNs and MFNs, in case it is necessary to use a real MFN, for example, to utilize DMA from a device. HVM guests are not aware of PFNs vs. MFNs. Instead, they only use physical frame numbers and any translation necessary is done by the underlying hypervisor.

For this reason, PV-on-HVM device drivers pass physical addresses to the underlying hypervisor to share memory pages with the back-end drivers. In the Xen-Blanket, however, the MFN from the Blanket Domain 0's perspective, and thus the Blanket drivers', matches the PFN that the underlying hypervisor expects. Therefore, Blanket drivers must perform a PFN-to-MFN translation before passing any addresses to the underlying hypervisor, either through hypercalls or PCI operations.

Blanket Drivers for KVM

The implementation of Blanket drivers for KVM is very similar. Paravirtualized device drivers in KVM use the `virtio` framework, in which a PCI device is exposed to guests, similar to the Xen platform PCI device. Unlike the Xen platform PCI device, all communication with the underlying KVM hypervisor can be accomplished as if communicating with a physical PCI device. In particular, no direct hypercalls are necessary, simplifying the implementation of Blanket drivers. The only modifications required to run `virtio` drivers in the Xen-Blanket are the addition of PFN-to-MFN translations.

4.2.3 Hypervisor Optimizations

The Xen-Blanket runs in non-root mode in an HVM guest container. As virtualization support improves, the performance of software running in non-root mode becomes close to running on bare metal. For example, whereas page table manipulations would cause a `vmexit`, or trap, on early versions of Intel VT-x processors, a hardware feature called extended page tables (EPT) has largely eliminated such traps. However, some operations continue to generate traps, so designing the Blanket layer to avoid such operations can often provide a performance advantage.

For example, instead of flushing kernel pages from the TLB on every context switch, the x86 contains a bit in the `cr4` control register called the "Page Global Enable" (PGE). Page Global Enable allows certain pages to be mapped as "global" so that they do not get flushed automatically. Xen enables then disables the PGE bit in order to flush the global TLB entries before doing a domain switch between guests. Unfortunately, these `cr4` operations each cause `vmexit`s to happen, generating high overhead for running Xen in an HVM guest. By not using PGE and instead flushing all pages from the TLB on a context switch, `vmexit`s are avoided, because of the EPT processor feature in non-root mode.

4.2.4 Implications of Future Hardware and Software

As discussed above the virtualization features of the hardware, such as EPT, can have a profound effect on the performance of the Xen-Blanket. Further improvements to the HVM container, such as the interrupt path, may eventually replace hypervisor optimizations and workarounds or enable even better performing

Blanket layers.

Other hardware considerations include thinking about non-root mode as a place for virtualization. For example, features that aided virtualization before hardware extensions became prevalent, such as memory segmentation, should not die out. Memory segmentation is a feature in 32 bit x86 processors that paravirtualized Xen leverages to protect Xen, the guest OS, and the guest user space in the same address space to minimize context switches during system calls. The 64 bit x86_64 architecture has dropped support for segmentation except when running in 32 bit compatibility mode. Without segmentation, two address spaces are needed to protect the three contexts from each other, and two context switches are required on each system call, resulting in performance loss.

On the software side, support for nested virtualization of unmodified guests [43] may begin to be adopted by cloud providers. While this development could eventually lead to fully virtualized Blankets such as a KVM-Blanket, relying on providers to deploy such a system is provider-centric: every cloud must incorporate such technology before a KVM-Blanket becomes feasible across many clouds. It may be possible, however, for exposed hardware virtualization extensions to be leveraged as performance accelerators for a system like the Xen-Blanket.

4.3 Evaluation

We have built Blanket drivers and deployed the Xen-Blanket on two underlying hypervisors, across three cloud providers. In this section, we first examine the

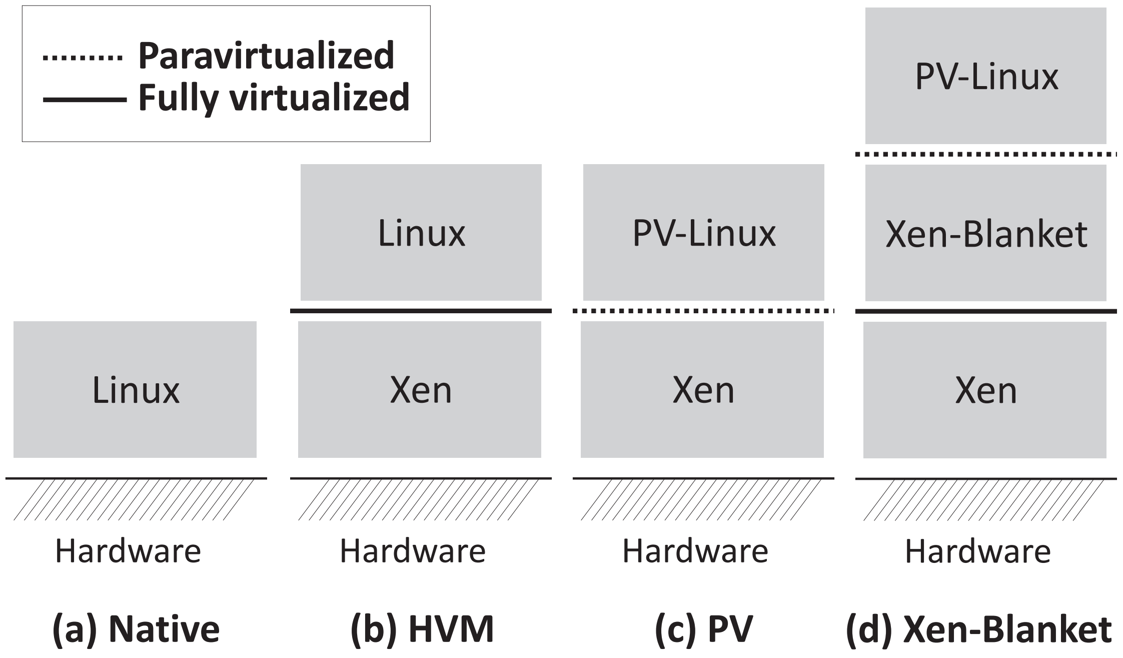


Figure 4.4: We run benchmarks on four different system configurations in order to examine the overhead caused by the Xen-Blanket. *Native* represents an unmodified CentOS 5.4 Linux. *HVM* represents a standard single-layer Xen-based virtualization solutions using full, hardware-assisted virtualization. *PV* represents a standard single-layer Xen-based virtualization solutions using paravirtualization. *Xen-Blanket* consists of a paravirtualized setup inside of our Xen-Blanket HVM guest.

overhead incurred by the Xen-Blanket. Then, we describe how increased flexibility resulting from a user-centric homogenization layer can result in significant cost savings—47% of the cost per hour—on existing clouds, despite overheads.

4.3.1 Overhead

Intuitively, we expect some amount of degraded performance from the Xen-Blanket due to the overheads of running a second-layer of virtualization. We compare four different scenarios, denoted by *Native*, *HVM*, *PV*, and *Xen-Blanket*

(Figure 4.4). The Native setup ran an unmodified CentOS 5.4 Linux. The next two are standard single-layer Xen-based virtualization solutions using full, hardware-assisted virtualization (HVM, for short) or paravirtualization (PV, for short), respectively. The fourth setup (Xen-Blanket) consists of a paravirtualized setup inside an HVM guest.[5] All experiments in this subsection were performed on a pair of machines connected by a 1 Gbps network, each with two six-core 2.93 GHz Intel Xeon X5670 processors,[6] 24 GB of memory, and four 1 TB disks. Importantly, the virtualization capabilities of the Xeon X5670 include extended page table support (EPT), enabling a guest OS to modify page tables without generating `vmexit` traps. With the latest hardware virtualization support, HVM is expected to outperform PV, because of reduced hypervisor involvement. Therefore, since the Xen-Blanket setup contains a PV setup, PV can be roughly viewed as a best case for the Xen-Blanket.

System Microbenchmarks

To examine the performance of individual operations, such as null system calls, we ran `lmbench` [111], a microbenchmark suite, in all setups. In order to distinguish the second-layer virtualization overhead from CPU contention, we ensure that one CPU is dedicated to the guest running the benchmark. To clarify, one VCPU backed by one physical CPU is exposed to the guest during single-layer virtualization experiments, whereas the Xen-Blanket system receives two VCPUs backed by two physical CPUs: one is reserved for the second-layer Domain 0 (see Figure 4.2(c)), and the other one for the second-layer guest.

[5]We have also run experiments on KVM with comparable results, but focus on a single underlying hypervisor for a consistent evaluation.

[6]Hyperthreading causes the OS to perceive 24 processors on the system.

	Native	**HVM**	**PV**	**Xen-Blanket**
Processes (μs)				
null call	0.19	0.21	0.36	0.36
null I/O	0.23	0.26	0.41	0.41
stat	0.85	1.01	1.19	1.18
open/close	1.33	1.43	1.84	1.86
slct TCP	2.43	2.79	2.80	2.86
sig inst	0.25	0.39	0.54	0.53
sig hndl	0.90	0.79	0.94	0.94
fork proc	67	86	220	258
exec proc	217	260	517	633
sh proc	831	1046	1507	1749
Context Switching (μs)				
2p/0K	0.40	0.55	2.85	3.07
2p/16K	0.44	0.57	3.03	3.46
2p/64K	0.45	0.66	3.18	3.46
8p/16K	0.74	0.85	3.60	4.00
8p/64K	1.37	1.18	4.14	4.53
16p/16K	1.05	1.10	3.80	4.14
16p/64K	1.40	1.22	4.08	4.47
File & Virtual Memory (μs)				
0K file create	4.61	4.56	4.99	4.97
0K file delete	3.03	3.18	3.19	3.14
10K file create	14.4	18.1	19.9	28.8
10K file delete	6.17	6.02	6.01	6.08
mmap latency	425.0	820.0	1692.0	1729.0
prot fault	0.30	0.28	0.38	0.40
page fault	0.56	0.99	2.00	2.10

Table 4.1: The Xen-Blanket achieves performance within 3% of PV for simple `lmbench` [111] operations, but incurs overhead up to 30% for file creation microbenchmarks.

Table 4.1 shows the results from running `lmbench` in each of the setups. For simple operations like a null syscall, the performance of the Xen-Blanket is within 3% of PV, but even PV is slower than native or HVM. This is because

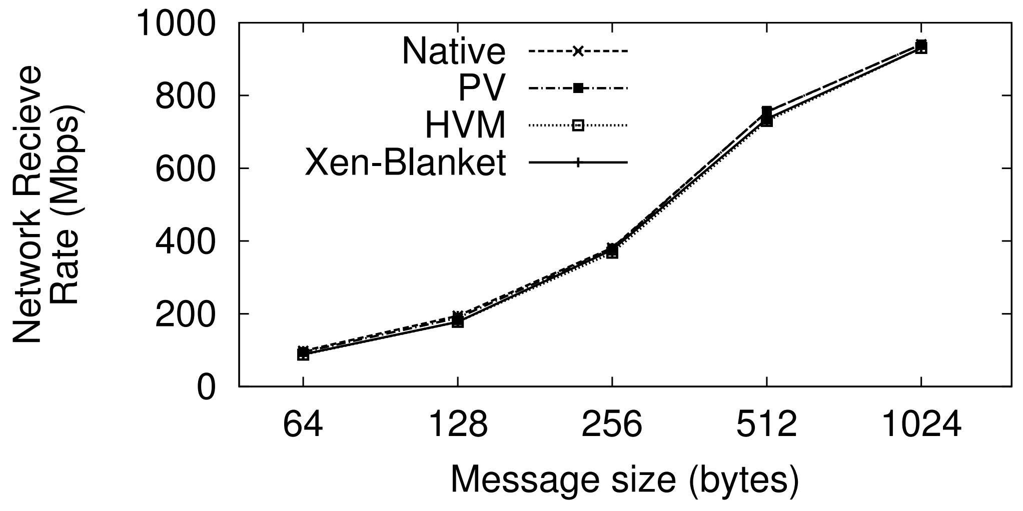


Figure 4.5: Network I/O performance on the Xen-Blanket is comparable to a single layer of virtualization.

a syscall in any paravirtualized system first switches into (the top-most) Xen before being bounced into the guest OS. We stress that, for these operations, nesting Xen does not introduce additional overhead over standard paravirtualization. All context switch benchmarks are within 12.5% of PV, with most around 8% of PV. Eliminating `vmexits` caused by the second-layer Xen is essential to achieve good performance. For example, if the second-layer Xen uses the `cr4` register on every context switch, overheads increase to 70%. Worse, on processors without EPT, which issue `vmexits` much more often, we measured overheads of up to 20×. For maximizing performance, it is crucial to use modern hardware and to carefully design second-layer hypervisor software to avoid expensive instructions.

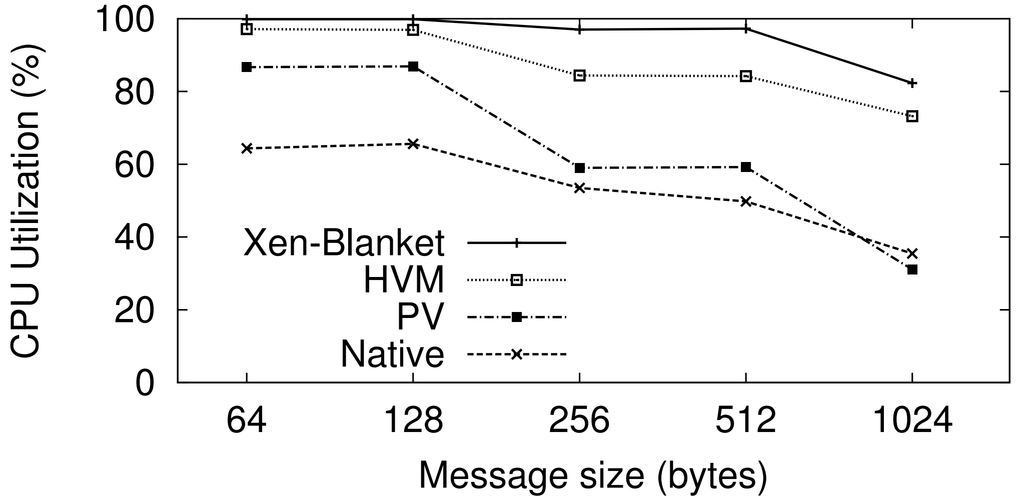


Figure 4.6: CPU utilization while receiving network I/O on the Xen-Blanket is within 15% of a single layer of virtualization.

Blanket Drivers

Device I/O is often a performance bottleneck even for single-layer virtualized systems. Paravirtualization is essential for performance, even in fully-virtualized environments. To examine the network and disk performance of the Xen-Blanket, we assign each of the configurations one VCPU (we disable all CPUs except for one in the native case). Figure 4.5 and Figure 4.6 show the UDP receive throughput and the corresponding CPU utilization[7] under various packet sizes. We use `netperf` [90] for the throughput measurement and `xentop`[8] in the underlying Domain 0 to measure the CPU utilization of the guest (or Xen-Blanket and guest). The CPU utilization of the native configuration is determined using `top`. Despite the two layers of paravirtualized de-

[7]Errorbars are omitted for clarity: all CPU utilization measurements were within 1.7% of the mean.

[8]`xentop` is a CPU monitoring tool that is packaged with Xen [26].

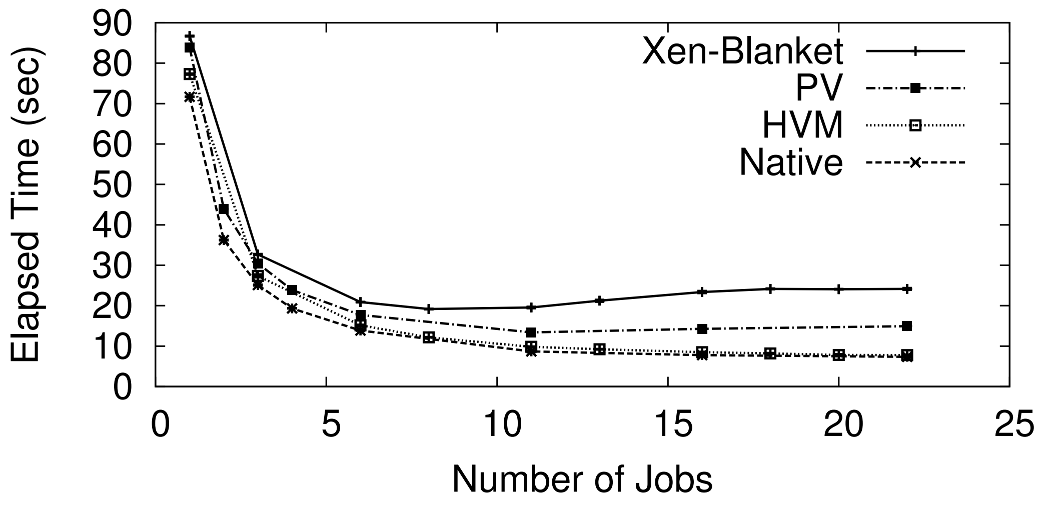


Figure 4.7: The Xen-Blanket can incur up to 68% overhead over PV when completing a `kernbench` benchmark.

vice interfaces, guests running on the Xen-Blanket can still match the network throughput of all other configurations for all packet sizes, and receive network traffic at full capacity over a 1 Gbps link. The Xen-Blanket does incur more CPU overhead because of the extra copy of packets in the Blanket layer.

We also ran the standard Unix tool `dd` to get a throughput measure of disk I/O. System caches at all layers were flushed before reading 2GB of data from the root filesystem. Native achieved read throughput of 124.6 MB/s, HVM achieved 86.3 MB/s, PV achieved 76.6 MB/s, and the Xen-Blanket incurred an extra overhead of 12% over PV, with disk read throughput of 67.6 MB/s. Unlike naïve approaches that use device emulation, previously evaluated in Section 3.3, Blanket drivers achieve good performance for I/O operations.

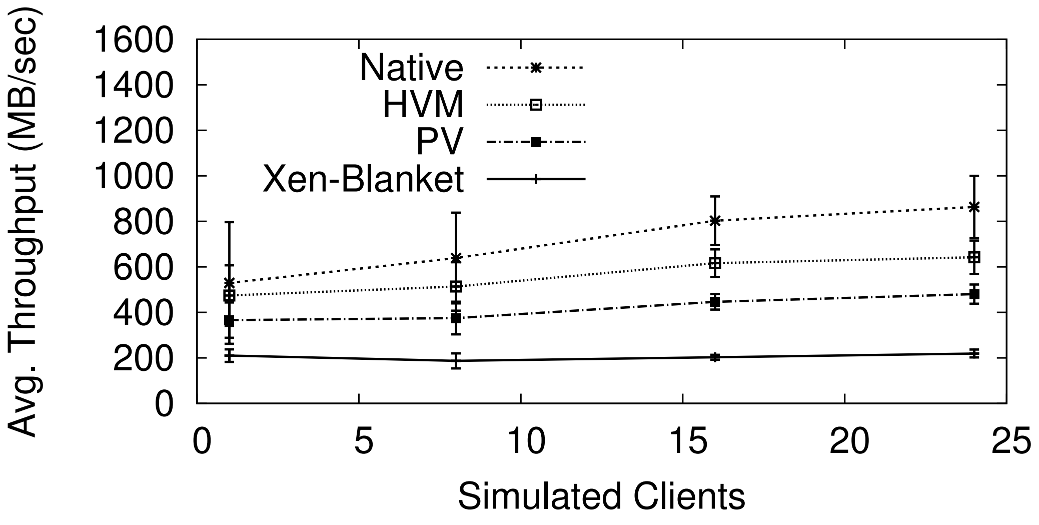


Figure 4.8: The Xen-Blanket can incur up to 55% overhead over PV when performing the `dbench` filesystem benchmark.

Macrobenchmarks

Macrobenchmarks are useful for demonstrating the overhead of the system under more realistic workloads. For these experiments, we dedicate 2 CPUs and 8 GB of memory to the lowest layer Domain 0. The remaining 16 GB of memory and 22 CPUs are allocated to single layer guests. In the case of the Xen-Blanket, we allocate 14 GB of memory and 20 CPUs to the Blanket guest, dedicating the remainder to the Blanket Domain 0. Unlike the microbenchmarks, resource contention does contribute to the performance measured in these experiments.

The `kernbench`[9] CPU throughput benchmark operates by compiling the Linux kernel using a configurable number of concurrent jobs. Figure 4.7 shows the elapsed time for the kernel compile. With a single job, the Xen-Blanket stays

[9]http://freecode.com/projects/kernbench, version 0.50

within 5% of PV, however, performance falls to about 68% worse than PV for high concurrency. The performance loss here can be attributed to a high number of `vmexits` due to APIC (Advanced Programmable Interrupt Controller) operations to send inter-processor-interrupts (IPIs) between VCPUs. Despite this overhead, the flexibility of the Xen-Blanket enables reductions in cost, as described in Section 4.3.2.

The `dbench`[10] filesystem benchmark generates load on a filesystem based on the standard `NetBench` [113] benchmark. Figure 4.8 show the average throughput during load imposed by various numbers of simulated clients. Figure 4.9 shows the average latency for ReadX operations, where ReadX is the most common operation during the benchmark. PV and the Xen-Blanket both experience significantly higher latency than HVM. The advantage of HVM can be attributed to the advantages of hardware memory management because of extended page tables (EPT). The Xen-Blanket incurs up to 55% overhead over PV in terms of throughput, but the latency is comparable.

Finally, we ran the banking workload of SPECweb2009 for a web server macrobenchmark. For each experiment, a client workload generator VM running on another machine connected by a 1 Gbps link drives load for a server that runs PHP scripts. As SPECweb2009 is a Web server benchmark, the back-end database is simulated. A valid SPECweb2009 run requires 95% of the page requests to compete under a "good" time threshold (2s) and 99% of the requests to be under a "tolerable" time threshold (4s). Figure 4.10 shows the number of "good" transactions for various numbers of simultaneous sessions. VMs running in both PV and Xen-Blanket scenarios can support an identical number of

[10]http://dbench.samba.org/, version 4.0

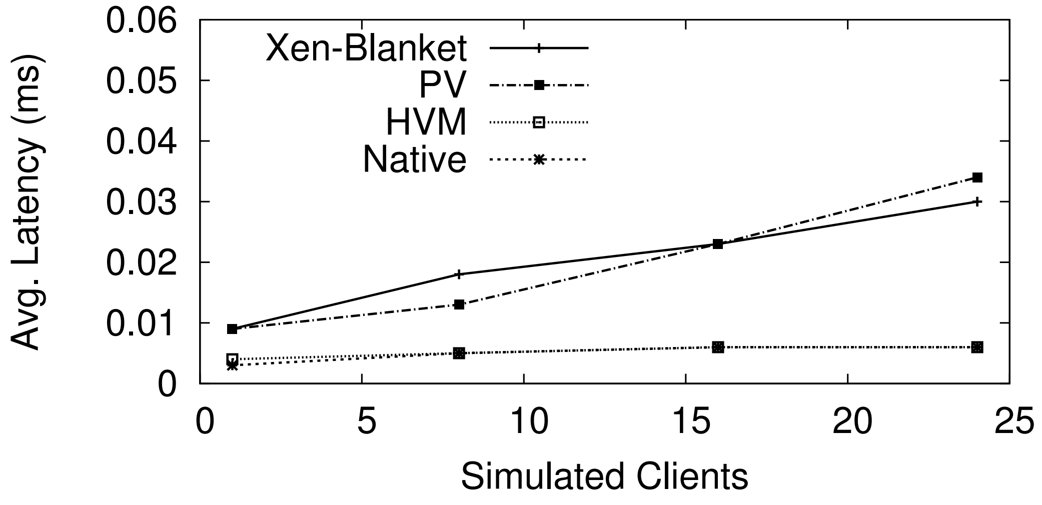


Figure 4.9: The average latency for ReadX operations during the `dbench` benchmark for Xen-Blanket remains comparable to PV.

simultaneous sessions.[11] This is because the benchmark is I/O bound, and the Blanket drivers ensure efficient I/O for the Xen-Blanket. The SPECweb2009 instance running in the Xen-Blanket does utilize more CPU to achieve the same throughput, however: average CPU utilization rises from 4.3% to 5.1% under 1000 simultaneous client sessions. Of the benchmarks shown, SPECweb2009 best represents a real enterprise workloads; on this benchmark, the Xen-Blanket performs well.

[11]PV and Xen-Blanket run the same VM and thus the same configuration of this complex benchmark. We omit a comparison with native and HVM to avoid presenting misleading results due to slight configuration variation.

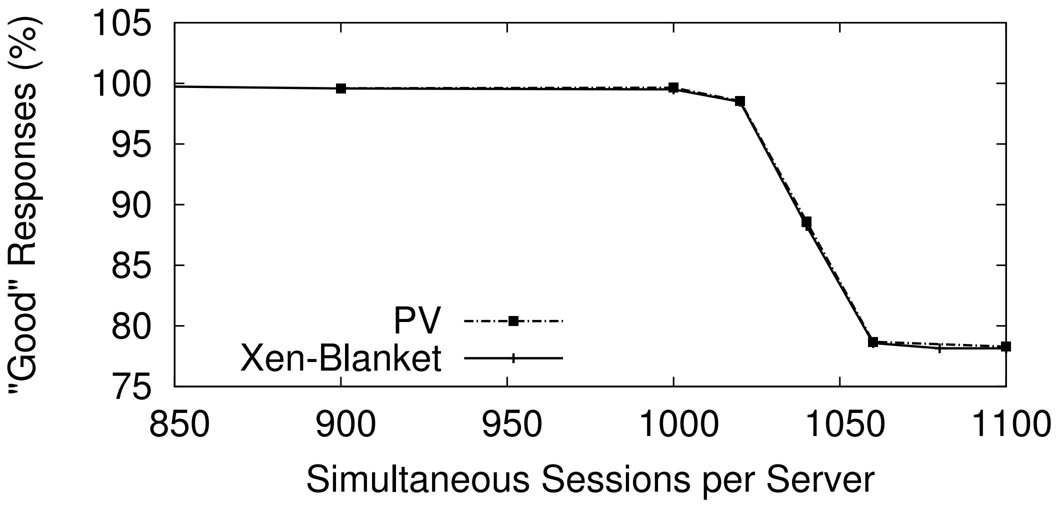


Figure 4.10: The Xen-Blanket performs just as well as PV for the SPECweb2009 macrobenchmark.

4.3.2 User-defined Oversubscription

Even though running VMs in the Xen-Blanket does incur overhead, its user-centric design gives a cloud user the flexibility to utilize cloud resources substantially more efficiently than possible on current clouds. Efficient utilization of cloud resources translates directly into monetary savings. In this subsection, we evaluate oversubscription on the Xen-Blanket instantiated within Amazon EC2 and find CPU-intensive VMs can be deployed for 47% of the cost of small instances.

Table 4.2 shows the pricing per hour on Amazon EC2 to rent a small instance or a quadruple extra large cluster compute instance (cluster 4XL). Importantly, while the cluster 4XL instance is almost a factor of 19 times more expensive than a small instance, some resources are greater than 19 times more abundant

Type	CPU (ECUs)	Memory (GB)	Disk (GB)	Price ($/hr)
Small	1	1.7	160	0.085
Cluster 4XL	33.5	23	1690	1.60
Factor	33.5×	13.5×	10×	18.8×

Table 4.2: The resources on Amazon EC2 instance types do not scale up uniformly with price. The user-centric design of Xen-Blanket allows users to exploit this fact.

(e.g. 33.5 times more for CPU) while other resources are less than 19 times more abundant (e.g 10 times more for disk). This suggests that if a cloud user has a number of CPU intensive VMs normally serviced as small instances, it may be more cost-efficient to rent a cluster 4XL instance and oversubscribe the memory and disk. This is not an option provided by Amazon; however, the Xen-Blanket is user-centric and therefore gives the user the necessary control to implement such a configuration. A number of would-be small instances can be run on the Xen-Blanket within a cluster 4XL instance, using oversubscription to reduce the price per VM.

To illustrate this point, we ran a CPU-intensive macrobenchmark, `kernbench`, simultaneously in a various numbers of VMs running inside a single cluster 4XL instance with the Xen-Blanket. We also ran the benchmark inside a small EC2 instance for a comparison point. The benchmark was run without concurrency in all instances for consistency, because a small instance on Amazon only has one VCPU. Figure 4.11 shows the elapsed time to run the benchmark in each of these scenarios. Each number of VMs on the Xen-Blanket corresponds to a different monetary cost. For example, to run a single VM, the cost is $1.60 per hour. 10 VMs reduce the cost per VM to $0.16 per hour, 20 VMs

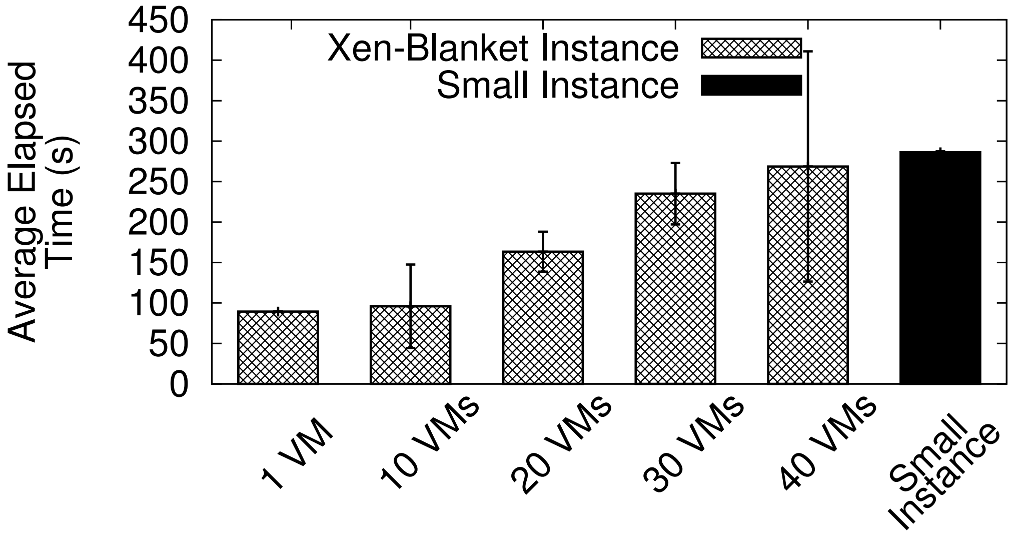


Figure 4.11: The Xen-Blanket gives the flexibility to oversubscribe such that each of 40 VMs on a single 4XL instance can simultaneously complete compilation tasks in the same amount of time as a small instance.

to \$0.08 per VM per hour, 30 VMs to \$0.06 per VM per hour, and 40 VMs to \$0.04 per VM per hour. Running a single VM, the benchmark completes in 89 seconds on the Xen-Blanket, compared to 286 seconds for a small instance. This is expected, because the cluster 4XL instance is significantly more powerful than a small instance. Furthermore, the average benchmark completion time for even 40 VMs remains 33 seconds faster than for a small instance. Since a small instance costs \$.085 per VM per hour, this translates to 47% of the price per VM per hour. It should be noted, however, that the variance of the benchmark performance significantly increases for large numbers of VMs on the same instance.

In some sense, the cost benefit of running CPU intensive instances inside the Xen-Blanket instead of inside small instances simply exploits an artifact of Amazon's pricing scheme. However, other benefits from oversubscription are

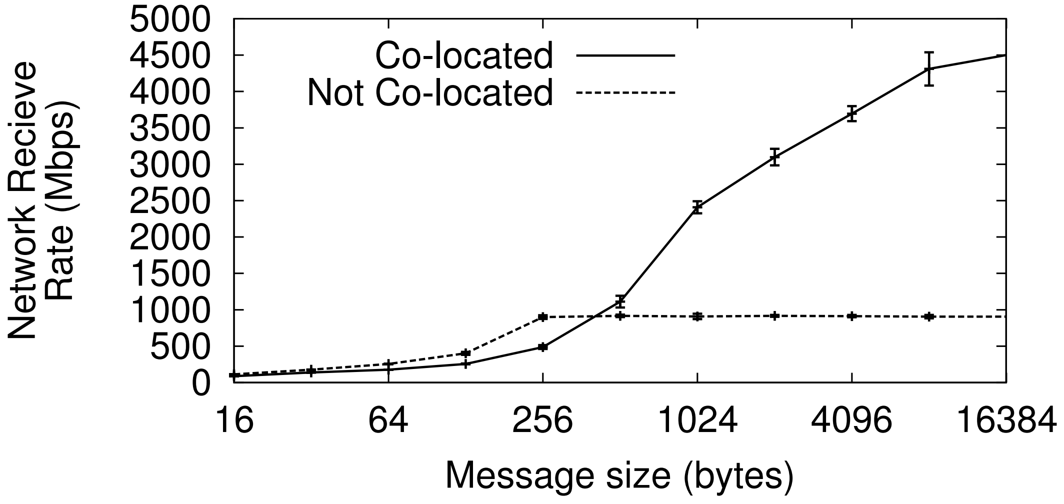


Figure 4.12: Co-location of VMs to improve network bandwidth is another simple optimization made possible by the user-centric approach of the Xen-Blanket.

possible, especially when considering VMs that have uncorrelated variation in their resource demands. Every time one VM experiences a burst of resource usage, others are likely quiescent. If VMs are not co-located, each instance must operate with some resources reserved for bursts. If VMs are co-located, on the other hand, a relatively small amount of resources can be shared to be used for bursting behavior, resulting in less wasted resources.

Co-location of VMs also affect the performance of enterprise applications, made up of a number of VMs that may heavily communicate with one another [136]. To demonstrate the difference that VM placement can make to network performance, we ran the `netperf` TCP benchmark between two VMs. In the first setup, the VMs were placed on two different physical servers on the same rack, connected by a 1 Gbps link. In the second, the VMs were co-located on the same physical server. Figure 4.12 shows the network throughput.

The co-located servers are not limited by the network hardware connecting the physical machines. By enabling co-location, the Xen-Blanket can increase inter-VM throughput by a factor of 4.5. This dramatic result is without any modification to the VMs. The user-centric design of the Xen-Blanket enables other optimization opportunities, including CPU bursting, page sharing and resource oversubscription, that can offset the inherent overhead of the approach.

4.4 Experience with Multi-Cloud Migration

The Xen-Blanket homogenizes and simplifies the process of migrating a VM between two clouds managed by two different providers. While it is currently possible to migrate VMs between multiple clouds, the process is cloud-specific and fundamentally limited. For example, it is currently impossible to live migrate [57, 122] a VM between cloud providers. We give a qualitative comparison to illustrate the difficulty faced in migrating a Xen VM from our private Xen environment to Amazon EC2 with and without the Xen-Blanket. We also show how one can reintroduce live migration across multi-clouds using the Xen-Blanket. In our experiment, we use a VM housing a typical legacy LAMP-based[12] application that contained non-trivial customizations and approximately 20 GB of user data.

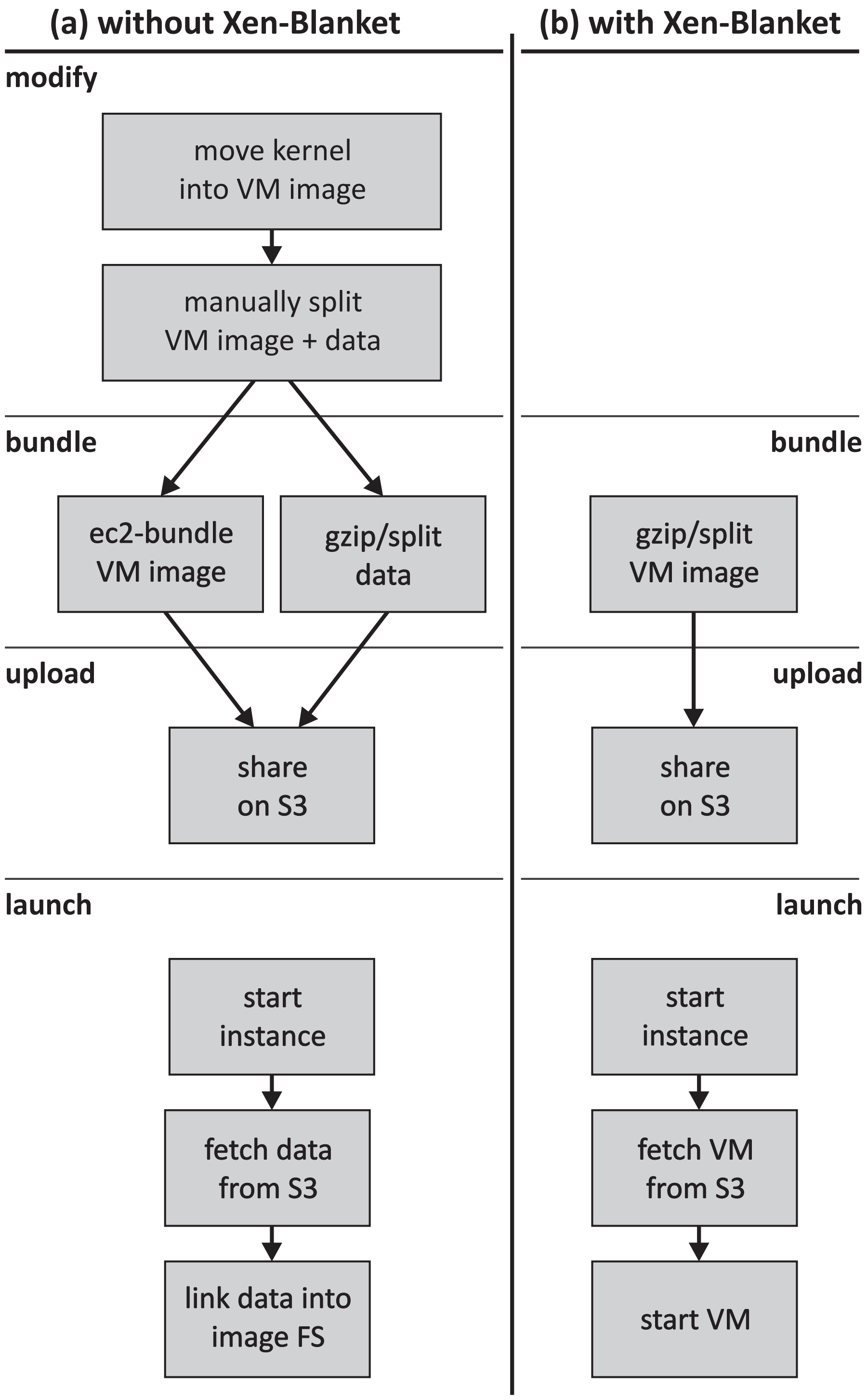


Figure 4.13: Comparison between the steps it takes to migrate (offline) an image into Amazon's EC2 with and without the Xen-Blanket

4.4.1 Non-Live Multi-Cloud Migration

Figure 4.13 summarizes the four steps involved in a migration: *modifying* the VM's disk image to be compatible with EC2, *bundling* or compressing the image to be sent to EC2, *uploading* the bundled image, and *launching* the image at the new location. In both scenarios, bundling, uploading and launching took one person about 3 hrs. However, the modify step caused the scenario without the Xen-Blanket to be much more time consuming: 24 hrs additional work as compared to no additional work with the Xen-Blanket.

Migrating a VM image from our private setup to Amazon EC2 is relatively straightforward given the Xen-Blanket. No image modifications are required, so the process begins with bundling, or preparing the image to upload for use in EC2. The image was compressed with `gzip`, split into 5 GB chunks for Amazon's Simple Storage Service (S3), and uploaded. Then, we started an EC2 instance running the Xen-Blanket, retrieved the disk image from S3, concatenated the pieces of the file, and unzipped the image. The VM itself was created using standard Xen tools.

Without the Xen-Blanket, there currently exists a EC2-specific process to create an Amazon Machine Image (AMI) from an existing Xen disk image, roughly matching the bundle, upload, and launch steps. Before that, two modifications were required to our VM image. First, we had to modify the image to contain the kernel because no compatible kernel was offered by EC2.[13] This task was complicated by the fact that our private Xen setup did not have the correct tools

[12]Linux, Apache, MySQL, and PHP

[13]Before July 2010, Amazon EC2 only allowed a limited selection of a few standard kernels and initial ramdisks for use outside the image. After July 2010, Amazon EC2 began to support kernels stored within the image.

to boot the kernel within the image. Second, we had to shrink our 40 GB disk image to fit within the 10 GB image limit on EC2. This involved manually examining the disk image in order to locate, copy, and remove a large portion of the application data, then resizing the VM's filesystem and image. After the modifications were complete, we used an AMI tool called `ec2-bundle-image` to split the VM image into pieces and then compressed, split and uploaded the relocated data to S3. We then started an EC2 instance with our new AMI, configured it to mount a disk, and reintegrated the user data from S3 into the filesystem.

It should be noted that subsequent launches of the migrated VMs do not require all of the steps outlined above. However, if a modified or updated version of the VM is released, the entire process must be redone. Even worse, we expect migrating to other clouds to be similarly arduous and provider-specific, if possible at all. In contrast, using the Xen-Blanket, the migration process will always be the same, and can be reduced to a simple remote copy operation.

4.4.2 Live Multi-Cloud Migration

Live migration typically relies on memory tracing: a hypervisor-level technique. Such techniques are not available across clouds.[14] The Xen-Blanket enables immediate implementation of live migration across cloud providers. We have experimented with live migration between an enterprise cloud and Amazon EC2. We note that since live migration between two clouds is not currently possible without provider support or the Xen-Blanket, we do not have a comparison

[14]While some providers expose an interface for users to use live migration within their own cloud, as with other provider-centric approaches, standardization may take years.

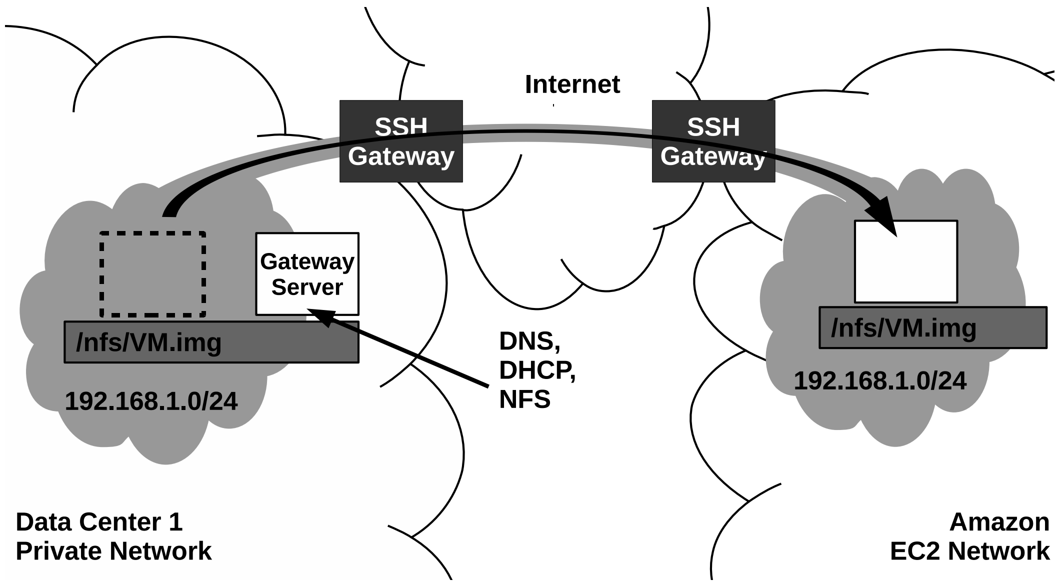


Figure 4.14: Xen-Blanket instances are connected with a layer-2 tunnel, while a gateway server VM provides DNS, DHCP and NFS to the virtual network, eliminating the communication and storage barriers to multi-cloud live migration.

point to present.

Beyond the ability to implement hypervisor-level features like memory tracing, there are two key challenges to implement live multi-cloud migration on the Xen-Blanket. First, Xen-Blanket instances in different clouds are in different IP subnets, causing communication issues before and after Blanket guest migrations. Second, Xen-Blanket instances in different clouds do not share network attached storage, which is often assumed for live VM migration.

To address the networking issues, each Xen-Blanket instance runs a virtual switch in Domain 0 to which the virtual network interfaces belonging to Blanket guest VMs are attached. A layer-2 tunnel connects the virtual switches across the Internet. The result is that VMs on either of the two Xen-Blanket instances appear to be sharing a private LAN. A few basic network services are useful to introduce onto the virtual network. A gateway server VM can be run with two virtual network interfaces: one attached to the virtual switch and the virtual network; the other attached to the externally visible interface of the Xen-Blanket instance. The gateway server VM, shown in Figure 4.14, runs `dnsmasq`[15] as a lightweight DHCP and DNS server.

Once VMs on the Xen-Blanket can communicate, the storage issues can be addressed with a network file system, such as NFS [133]. NFS is useful for live VM migration because it avoids the need to transfer the entire disk image of the VM at once during migration. In our setup, the gateway server VM also runs an NFS server. The NFS server exports files onto the virtual network and is mounted by the Domain 0 of each Xen-Blanket instance. Both Xen-Blanket instances mount the NFS share at the same location. Therefore, during VM migration, the VM root filesystem image can always be located at the same filesystem location, regardless of the physical machine.

With Xen-Blanket VMs able to communicate, maintain their network addresses, and access storage within either cloud, live VM migration proceeds by following the typical procedure in the Blanket hypervisor. However, while we have successfully live-migrated a VM from an enterprise cloud to Amazon EC2 and back, this is simply a proof-of-concept. It is clearly inefficient to rely on a NFS disk image potentially residing on another cloud instead of a local disk. Moreover, the layer-2 tunnel only connects two machines. We discuss a more sophisticated networking solution in which the cloud user implements the control logic in Chapter 5. Wide-area live migration techniques [46], can also be implemented and evaluated on the Xen-Blanket.

[15]http://www.thekelleys.org.uk/dnsmasq/doc.html

4.5 Summary

Current IaaS clouds lack the homogeneity required for users to easily deploy services across multiple providers. This prevents a user from deploying superclouds. We explored an alternative to standardization, or provider-centric homogenization, in which cloud users have the ability to homogenize the cloud themselves. In this chapter, we presented the Xen-Blanket, a system that enables user-centric homogenization of existing cloud infrastructures.

The Xen-Blanket leverages a second-layer Xen hypervisor—completely controlled by the user—that utilizes a set of provider-specific Blanket drivers to execute on top of existing clouds without requiring any modifications to the provider. Blanket drivers have been developed for both Xen and KVM based systems, and achieve high performance: network and disk throughput remain within 12% of paravirtualized drivers in a single-level paravirtualized guest. The Xen-Blanket is currently running on Amazon EC2, an enterprise cloud, and private servers at Cornell University. The Xen-Blanket project website is located at `http://xcloud.cs.cornell.edu/`, and the code for the Xen-Blanket is publicly available at `http://code.google.com/p/xen-blanket/`.

However, the homogeneous VM interface provided by the Xen-Blanket is not sufficient to build superclouds. Initial VM migration experiments, were successful at moving VM images between the three different sites with no modifications to the images, but exposed the inability of existing clouds to support network configuration. In the next chapter, we use the extensibility enabled by the Xen-Blanket to address network issues in the clouds.

Similarly, we have only scratched the surface in terms of the efficient use of cloud resources. We have exploited the user-centric nature of the Xen-Blanket to oversubscribe resources and save money on EC2, achieving a cost of 47% of the price per hour of small instances for 40 CPU-intensive VMs, despite the inherent overheads of nested virtualization. In Chapter 6, we look at oversubscription of memory for even higher efficiency.

CHAPTER 5

TOWARDS USER CONTROL OF CLOUD NETWORKS: VIRTUALWIRE

In order to build superclouds, existing clouds must be extended to support complex networks, such as those encountered in enterprise deployments. In this chapter, we leverage cloud extensibility and the Xen-Blanket to enable the cloud user to implement the network control logic necessary to support enterprise networks in existing third-party clouds.

Increasingly, cloud providers are interested in exposing virtual network abstractions to users, in an attempt to reduce or eliminate the manual, application-specific network reconfiguration often required for migrating enterprise workloads. Existing network virtualization systems (including VL2 [75], NetLord [118], and Nicira [14]) provide users with a private, virtual network address space, but limit the users' control over the virtual network. More specifically, the *control logic*, responsible for implementing flow policies in the virtual network, is implemented by the provider, as shown in Figure 5.1(a). Users can only interact with the control logic using well defined, high-level APIs.

In this chapter, we argue that implementing the control logic in the provider creates an unnecessary burden, both for the user and provider of a virtual network. Users must adapt network management and configuration techniques or tools to work with the features and APIs exposed by the cloud provider. Therefore, the migration of enterprise deployments and their complex networks and flow policies still requires non-trivial reconfiguration efforts. Furthermore, the provider shoulders the responsibility of implementing and managing a virtual network and its broad spectrum of features (e.g., addressing, routing, protocol support, flow policies) that may differ for each enterprise workload.

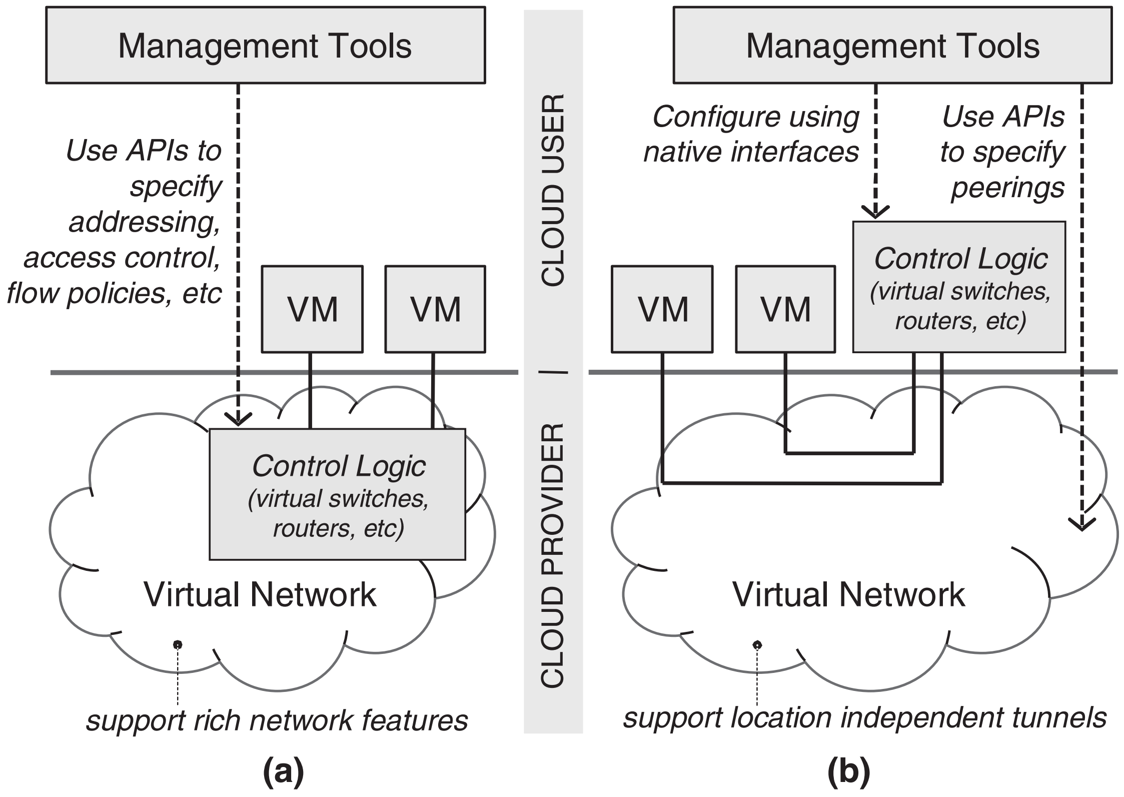

Management Tools
Use APIs to specify addressing, access control, flow policies, etc
VM
VM
Control Logic (virtual switches, routers, etc)
Virtual Network
support rich network features
(a)
CLOUD USER
CLOUD PROVIDER
Management Tools
Configure using native interfaces
Use APIs to specify peerings
VM
VM
Control Logic (virtual switches, routers, etc)
Virtual Network
support location independent tunnels
(b)

Figure 5.1: Virtual network control logic implemented by a) the provider or b) the user

To radically simplify the migration process, we present VirtualWire. VirtualWire is a system that provides a virtual network abstraction in which the user—not the provider—implements the control logic of the virtual network, as depicted in Figure 5.1(b). Users directly run their own virtualized equivalent of network devices (such as switches, routers, middleboxes), and configure them using low-level device interfaces, just like configuring a physical network. The API with the underlying system is simple. It consists of specifying the peerings between different virtual network interfaces of virtual network components in a process that mimics the act of plugging networking cables into network interface cards. The physical locations of all network components and the details of the underlying physical network are hidden from the user; a migrating enterprise deployment appears to maintain its network configuration without any changes to the way the network is managed or configured. Furthermore, consistent management of the network—virtual or physical—enables an incremental strategy for migration: even small parts of the network can be virtualized and run on third-party infrastructure.

With this simple interface, the cloud provider's task of managing the virtual network is reduced to ensuring efficient delivery of network traffic across paired virtual interfaces (as specified by the user), independent of their physical location. VirtualWire achieves this by managing *connectors* at the hypervisor level. Connectors consist of two *endpoints*, each associated with a virtual network interface as the bottom-half in a split-device model. All packets from one endpoint are tunneled to exactly one other endpoint using VXLAN's [105] wire format for encapsulating data link layer (L2) Ethernet frames in network layer (L3) IP packets.[1] When components are migrated, the affected connectors independently update their state to ensure traffic continues to be tunneled correctly between endpoints.

Inherent to VirtualWire is an assumption that virtualized software equivalents of physical network components exist and can be configured using the tools and techniques used on the physical components (discussed in Section 5.1.1). The performance implications of this assumption and techniques to incrementally enhance mission-critical parts of the network to use more efficient network designs and distributed component implementations are discussed in Section 5.1.5.

To date, we have implemented VirtualWire in Xen [41] to transparently inter-

[1]VXLAN is an emerging encapsulation standard towards enabling virtual L2 network segments to extend across layer 3 networks.

cept packets from virtual network interfaces and tunnel them across the physical network. We have also migrated deployments that rely on VLANs, IP multicast, and encoded flow policies through transparent middleboxes, to Amazon EC2 (a third-party cloud) by leveraging an open-source nested virtualization approach [155]. We have migrated a 3-tier application configured to use multiple firewall middleboxes onto EC2, achieving performance within 9% of a native EC2 deployment. We have even migrated deployments using not-yet-ubiquitous OpenFlow-enabled [110] switches. To demonstrate the consistency of the virtual network regardless of physical location, live VM migration is enabled by VirtualWire across one or more third-party clouds, not only for servers but also for network components. In particular, we have performed live VM migration from our local environment to Amazon EC2—with no changes to VMs or the network topology—incurring downtime as low as 1.4 s.

In this chapter, we make the following contributions:

- a new virtual network abstraction that places the control logic of the virtual network under user control to maintain complex network flow policies,
- a practical strategy for incremental migration to the cloud or upgrade of the enterprise network,
- an application of nested virtualization to extend the user's virtual network across one or more third-party clouds, and
- a demonstration of the preservation of the virtual network throughout live VM migrations of virtual servers and network components between our local setup and Amazon EC2.

The rest of this chapter is organized as follows. Section 5.1 and Section 5.2 describe the design and implementation of VirtualWire, respectively. Section 5.3 evaluates VirtualWire, including live VM migration between clouds; Section 5.4 concludes.

5.1 VirtualWire

This section describes the high-level design of VirtualWire, and highlights how providing a virtual network abstraction in which users can implement the control logic of their virtual network results in low management overhead and enables incremental migration in a real-world enterprise migration scenario. We also discuss the evolution of a VirtualWire virtual network to achieve high performance and limitations of this approach.

A virtual network in VirtualWire has a split architecture by design, illustrated in Figure 5.2. The cloud user is responsible for configuring the *user layer*, made up of *virtual network components*, while using a simple API to attach *connectors* between the virtual network interfaces on components. Connecters are optimized network tunnels that maintain the virtual network topology regardless of where virtual network components and servers are located; the provider implements the connector abstraction in the *provider layer*.

5.1.1 User Layer: Network Components

In VirtualWire, users run and configure isolated software modules called *virtual network components*, which contain configurable software implementations

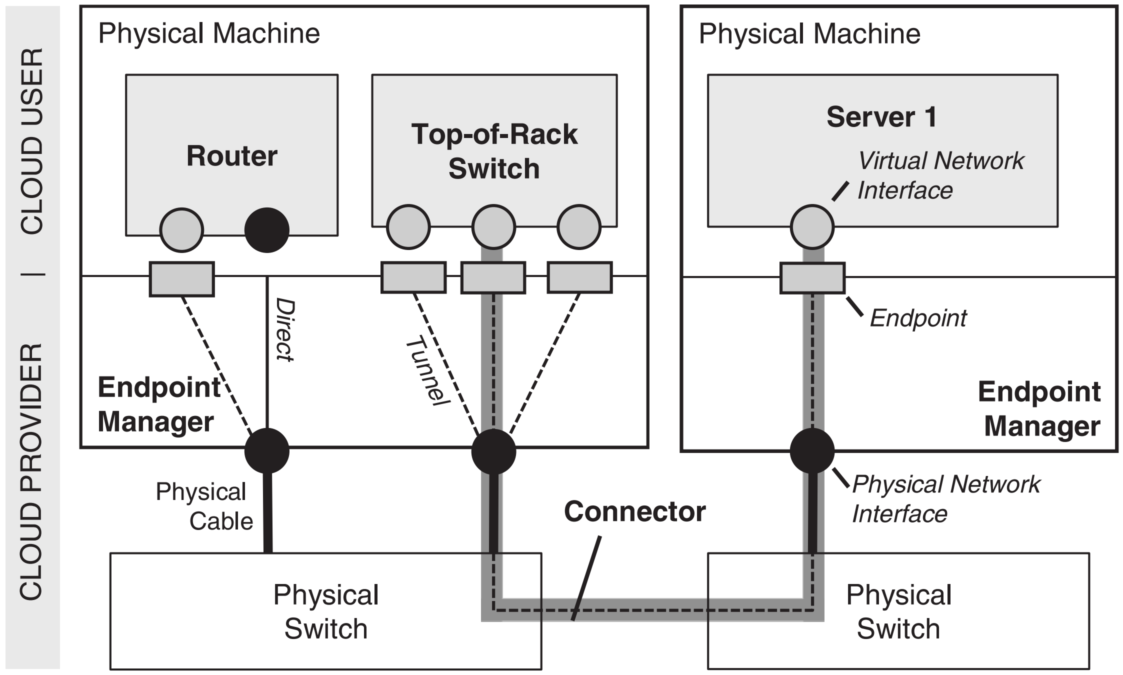


Figure 5.2: VirtualWire

of switches, routers, and middleboxes. Many software implementations of network components exist (e.g., Open vSwitch [126], Click router [97], XORP [82]), but it is most important for enterprise migration efforts that the software components can be configured identically to physical components. As computing infrastructure becomes ever more virtualized, hardware vendors have an incentive to release such virtual network components. For example, Cisco has released the Nexus 1000V series of production virtual switches [56] that can be managed identically to physical switches, using the Cisco command-line interface (CLI) [21], Simple Network Management Protocol (SNMP) [83], and tools like the Encapsulated Remote Switched Port Analyzer (ERSPAN) [4] and NetFlow [10]. Although not production, NetSim [18] contains software implementations of 42 routers and 6 switches to train Cisco network operators. Similarly the Olive JUNOS [5] implementation for training on Juniper devices runs on

FreeBSD.

Virtual network components can execute on—and migrate between—a variety of physical hosts. Regardless of what host a virtual network component is moved to, it will continue to perform according to its local configuration, which matches the configuration of the physical network component it has replaced. Packets are sent and received through one or more *virtual network interfaces*, which are analogous to physical ports on a device. In most cases, users specify that a virtual network interface of one component is connected (using a connector; see Section 5.1.2) to an interface of another component, allowing the components to interact on the virtual network.

5.1.2 Provider Layer: Connectors

The provider links virtual network interfaces on virtual network components together as specified by the user with *connectors*. Connectors have two *endpoints*, each bound to a virtual network interface. The binding between endpoint and virtual network interface is configured by an *endpoint manager* residing on the physical machine. This binding does not change, even if a network component is migrated to another physical machine. On migration, the local configuration of endpoints is updated to ensure the virtual network topology is maintained (see Section 5.1.3).

Encapsulation. Connectors are layer-2-in-layer-3 network tunnels. A layer 2 packet sent on a virtual network interface enters its associated endpoint, which encapsulates the full packet (with the entire MAC header, including VLAN tags)

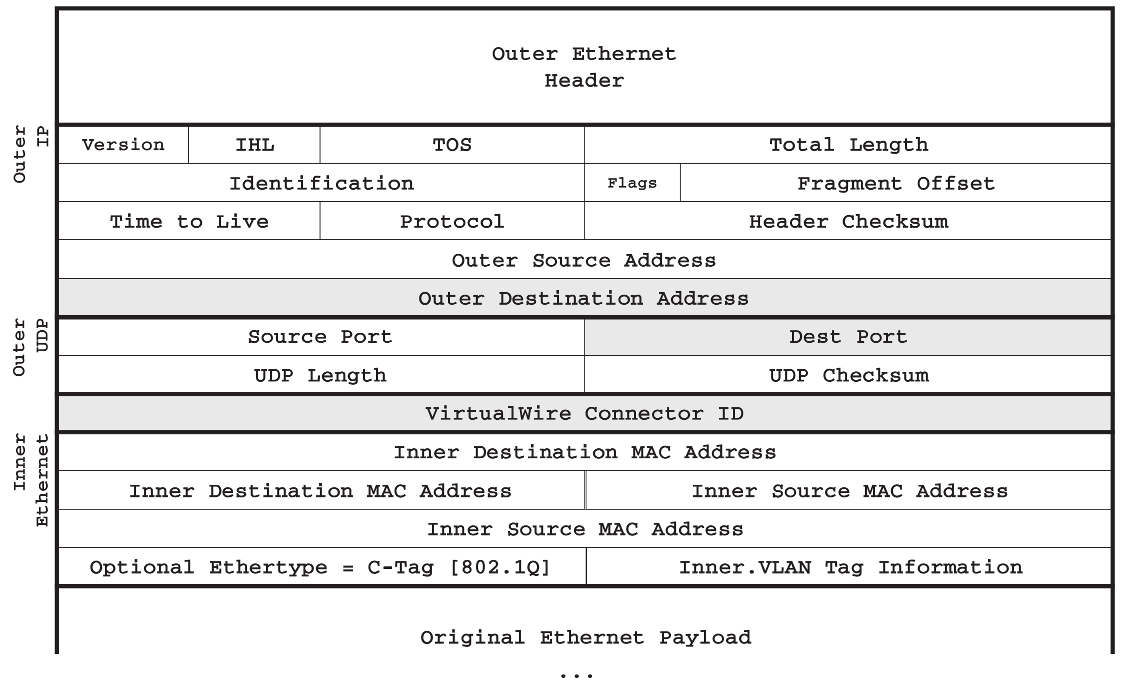


Figure 5.3: VirtualWire encapsulation

in a UDP packet; the 44 byte header is shown in Figure 5.3. The encapsulation used by endpoints is similar to the wire format in VXLAN [105]; however, unlike VXLAN, the header encodes a 32-bit connector ID instead of a network segment (e.g., a VXLAN Network Identifier (VNI) [105] in VXLAN terminology). The target IP address and port number correspond to the physical network address of the target endpoint manager. Upon receipt of a packet, the endpoint manager strips the outer headers, examines the connector ID, and forwards the packet to the target endpoint. Figure 5.4 details the path of a packet in VirtualWire. In order to maintain the network topology, every endpoint sends packets to exactly one other endpoint. If an endpoint is migrated, the migration process ensures the relevant endpoint configuration is updated to address encapsulated packets to the correct endpoint manager (further discussed in Section 5.1.3).

The extra 44 bytes added to each packet introduces a risk of fragmentation. To address this, the endpoint supports path maximum transmission unit (MTU) discovery, similar to Network Virtualization using Generic Routing Encapsulation (NVGRE) [137], such that a server can deduce the reduced MTU and avoid fragmentation of packets after encapsulation. For example, on a standard Ethernet network, the 1500 byte MTU is reported as 1456 bytes; on a network with jumbo frames enabled, the 9000 byte MTU is reported as 8956 bytes.

Since encapsulated packets are unreliable, it is possible that they may get dropped by the network for a number of reasons, including downtime during migration of a component or endpoint. This is akin to the unreliability of Ethernet, in which packets are delivered in best-effort fashion. If greater reliability is desired, a reliable transport protocol may be used to implement a connector.

Optimizations. Connectors between virtual network components running on the same host are automatically collapsed by the endpoint manager. In particular, the endpoint manager configures the endpoints to route packets directly to the co-located endpoint, rather than encapsulating self-addressed packets (see Figure 5.4). If, at a later time, one of the virtual network components is migrated to a different physical host, the endpoint manager re-enables encapsulation.

Extenders. Similarly, connectors also automatically extend across cloud networks and firewalls. Instead of addressing encapsulated packets directly to the recipient endpoint manager, packets are rerouted to an *extender*. An extender is a server (or set of servers) that both acts as an endpoint manager and maintains a permanent virtual private network (VPN) tunnel to another cloud. As shown in Figure 5.4, an extender may be local (co-located with an endpoint). Pack-

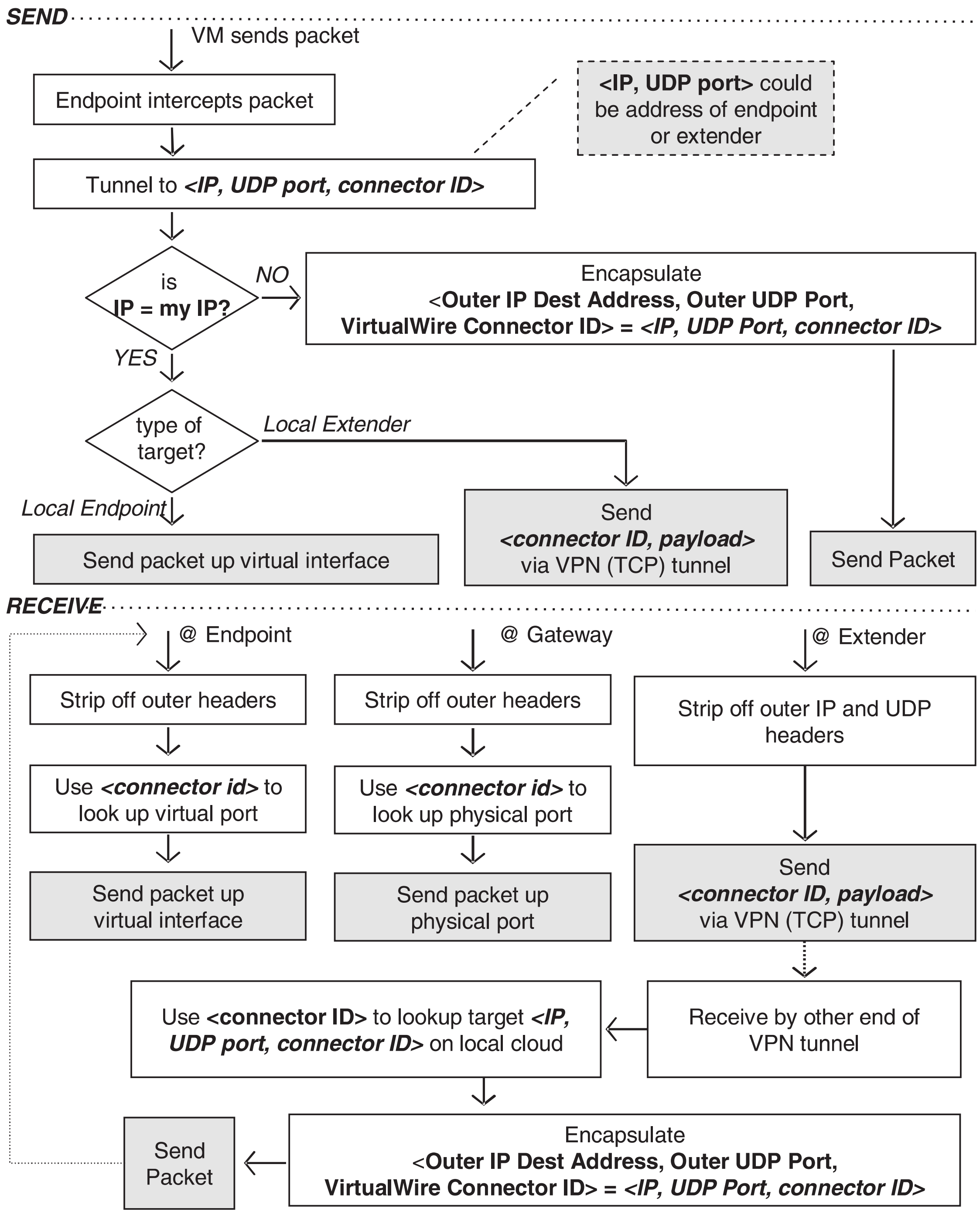

SEND
VM sends packet
Endpoint intercepts packet
<IP, UDP port> could be address of endpoint or extender
Tunnel to <IP, UDP port, connector ID>
is IP = my IP?
NO
Encapsulate <Outer IP Dest Address, Outer UDP Port, VirtualWire Connector ID> = <IP, UDP Port, connector ID>
YES
type of target?
Local Extender
Local Endpoint
Send packet up virtual interface
Send <connector ID, payload> via VPN (TCP) tunnel
Send Packet
RECEIVE
@ Endpoint
@ Gateway
@ Extender
Strip off outer headers
Strip off outer headers
Strip off outer IP and UDP headers
Use <connector id> to look up virtual port
Use <connector id> to look up physical port
Send packet up virtual interface
Send packet up physical port
Send <connector ID, payload> via VPN (TCP) tunnel
Use <connector ID> to lookup target <IP, UDP port, connector ID> on local cloud
Receive by other end of VPN tunnel
Send Packet
Encapsulate <Outer IP Dest Address, Outer UDP Port, VirtualWire Connector ID> = <IP, UDP Port, connector ID>

Figure 5.4: Path of a packet in VirtualWire

ets arriving at an extender are automatically sent across the VPN tunnel, where they enter a new endpoint that encapsulates the packet and sends it to the target endpoint manager. From the point of view of the network components, the extender is invisible: the two network components remain logically connected. If the virtual network components migrate to the same cloud, the extender will no longer be used.

5.1.3 Connector Management

VirtualWire exposes a simple interface made up of two operations, `connect` and `disconnect`, which create and destroy connectors between the virtual network interfaces on virtual servers and network components. Similarly, VirtualWire exposes a `connect` and `disconnect` for extenders. These interfaces augment the typical VM creation and destruction interfaces exposed by existing cloud managers (e.g., OpenStack [30], Eucalyptus [68], vSphere [25]). No additional configuration database is required by the cloud manager to keep track of connectors and endpoints; management of endpoint and connectors is intrinsically distributed.

For example, on migration, each host system is responsible for ensuring that endpoints are updated so that the logical network topology is maintained. Updating an endpoint consists of modifying the destination or source addresses used for encapsulation. Affected endpoints include both endpoints associated with every connector attached to the migrating component. Further, migration may necessitate the use of extenders where they were previously unnecessary or obviate their use in situations where they were being utilized. Connectors are automatically attached/detached to/from existing extenders if necessary.

5.1.4 Incremental Migration

The key to incremental migration of an enterprise application is the ability to move a subset of the servers and network components into VirtualWire, while allowing them to interact with the remainder of the (still-physical) servers and network components. To achieve this, a physical network appliance called a *VirtualWire gateway* can be used to bridge the virtual and physical networks. A VirtualWire gateway is a middlebox in the physical enterprise network that also hosts VirtualWire endpoints and communicates with other servers hosting VirtualWire components. Figure 5.5 shows a physical deployment partially migrated into VirtualWire, where a subset of the servers and their corresponding Top-of-Rack (ToR) switch are migrated. Traffic entering the gateway on a particular physical port flows into an associated VirtualWire endpoint hosted by the gateway. From the point of view of the physical and virtual network components on either side of the gateway, the gateway is transparent: traffic sent down a network cable from one component will arrive at the next component on the same interface as if both were still in the physical network. There can be many gateways in the remaining physical network; this is dependent on the network topology and order in which components are migrated.

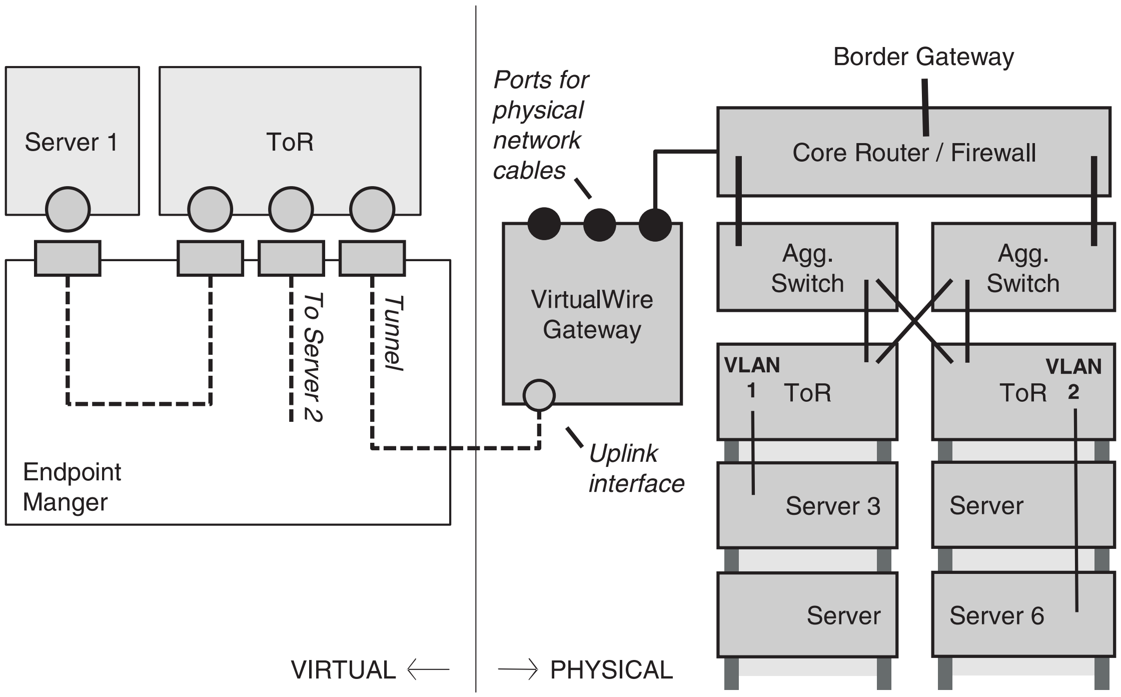


Figure 5.5: VirtualWire gateway

5.1.5 Discussion

Physical Interface Limitations. Although a virtual network component can support an identical number of virtual network interfaces as its physical counterpart, the physical host running the virtual component may have far fewer physical network interfaces than virtual, directly impacting performance. For example, a physical switch may have 32 ports, all capable of achieving 1 Gbps. When the switch is moved to VirtualWire, the virtual switch may be hosted on a physical machine with a single 1 Gbps interface. The virtual switch will still have 32 ports, but is now only capable of achieving 1 Gbps—in aggregate—due to the limitations of the physical interface. There are three approaches to alleviating this performance bottleneck: co-locating network components, implementing distributed components, and evolving performance-critical parts of the network.

Co-location. Co-location of virtual network components onto a single network host causes connectors to automatically eliminate encapsulation. Similarly, co-located components do not have a throughput limit imposed by the physical network link. At a larger scale, co-location of network components onto the same cloud can also significantly improve performance; cross datacenter links are not only lower performance, but incur additional monetary charges. Poor placement of severs and components leads to many expensive crossings, known as *traffic trombones*. The migration processes should, in general, optimize the placement of connected components; however, such placement optimizations are not discussed in this dissertation.

Distributed Components. During the incremental migration of an enterprise deployment, it may be difficult to efficiently place network components and servers to maximize co-location if some components are heavily shared. For example, consider a deployment containing a central switch acting as a bottleneck on the path between a large number of servers. Replacing the switch with a distributed virtual switch (e.g., VMware vSphere Distributed Switch (DVS) [144]) can allow data to flow directly to the next connected component in the virtual network, bypassing the bottleneck. In VirtualWire, distributed components can coexist with non-distributed components (e.g., an external gateway), maintaining the logical network topology.

Evolving the VirtualWire Network. Beyond co-location or incrementally reimplementing network components with distributed versions, multiple network components can be collapsed into one. For instance, if a number of switches in the original deployment were logically operating as a single big switch (e.g., Juniper Network's Virtual Chassis), only one (possibly distributed) component is required in the virtual network. Furthermore, in VirtualWire, subsections of the network which are well understood can be collapsed, while more complex subsets can remain functionally intact without modification.

5.2 Implementation

We have implemented VirtualWire on an enterprise cloud, infrastructure at Cornell, and Amazon EC2, where we do not have control of the underlying infrastructure. We leverage virtualization as an effective starting point to implement virtual network components, with connectors at the hypervisor level (Figure 5.7). We further leverage *nested* virtualization [43, 155] to transparently implement VirtualWire across different clouds.

5.2.1 Nested Virtualization Background

As described in Chapter 3, cloud extensibility allows extensions to be implemented across a mixture of public and private clouds. We deploy VirtualWire on existing clouds by leveraging the Xen-Blanket. As described in Chapter 4, the Xen-Blanket consists of a modified Xen [41] hypervisor that runs—without special support—on top of a variety of cloud providers. We install the Xen-Blanket across two types of clouds: private, where we had access to the underlying physical infrastructure, and public (specifically Amazon EC2), where we did

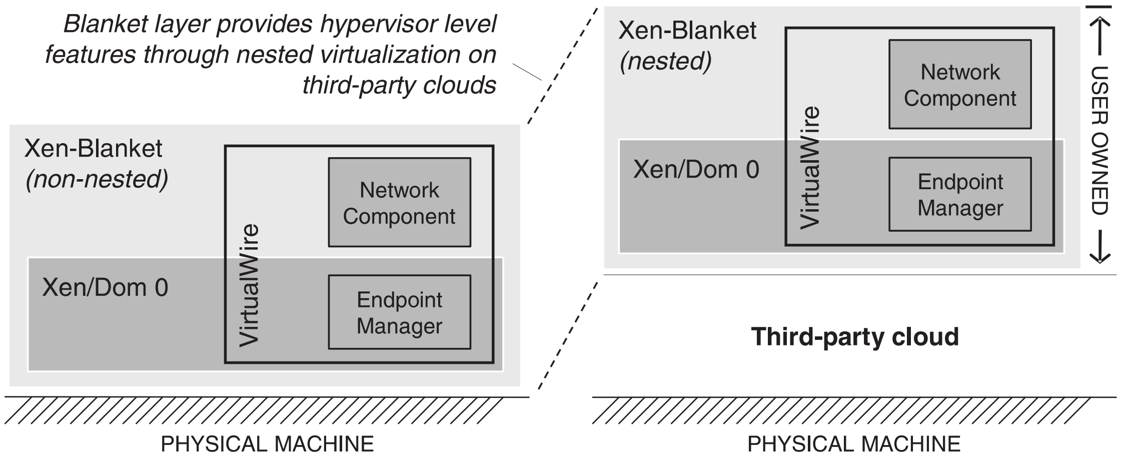


Figure 5.6: Nested Virtualization

not have access to the physical infrastructure or hypervisor. This configuration is depicted in Figure 5.6. In the private setting, we run the Xen-Blanket hypervisor directly on the hardware. In the public setting, we run the Xen-Blanket in nested mode. As a result, a configurable virtualization environment, called a *Blanket layer*, is created across a number of different physical environments in which we implement VirtualWire.

Guest VMs on the Blanket layer are assigned virtual network interfaces that use paravirtualized Xen split (front- and back-end) network drivers. Each guest virtual network interface is a front-end with a corresponding back-end in the control domain (Domain 0) of the Blanket layer. All guest VM packets pass through the back-end interface.

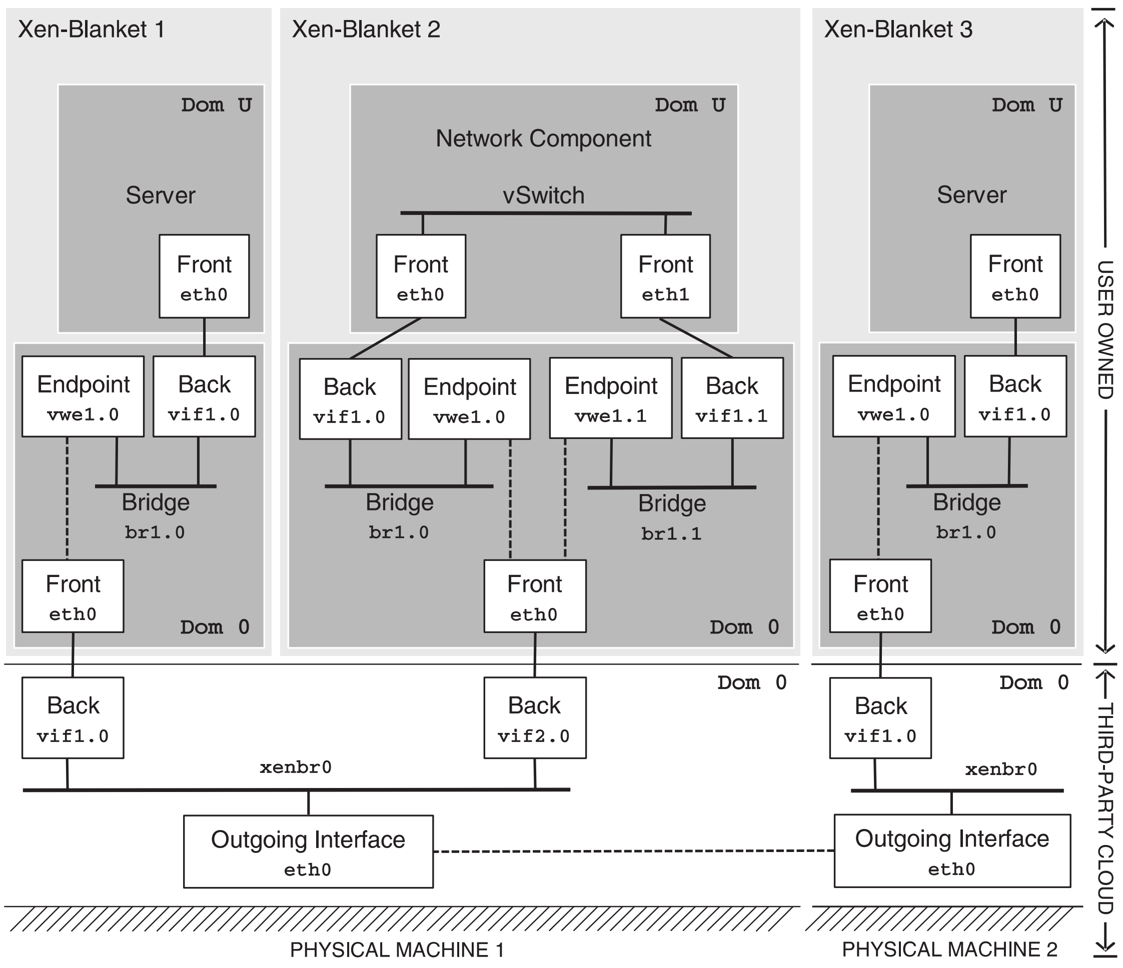


Figure 5.7: Implementation

5.2.2 Network Components

We implement network components in Xen-Blanket guest VMs. For example, Figure 5.7 shows two servers connected to a switch component. The switch component is implemented by running the standard Linux software bridge in a Xen-Blanket guest VM configured with two virtual network interfaces. To run a server or component VM on any Xen-Blanket hypervisor on any cloud, we place virtual disk images containing the root filesystem of VMs in a globally accessible NFS share.[2] Virtualization enables live migration of VMs between physical servers with virtually no downtime; as shown in Section 5.3.3, VirtualWire extends live migration across third-party clouds.

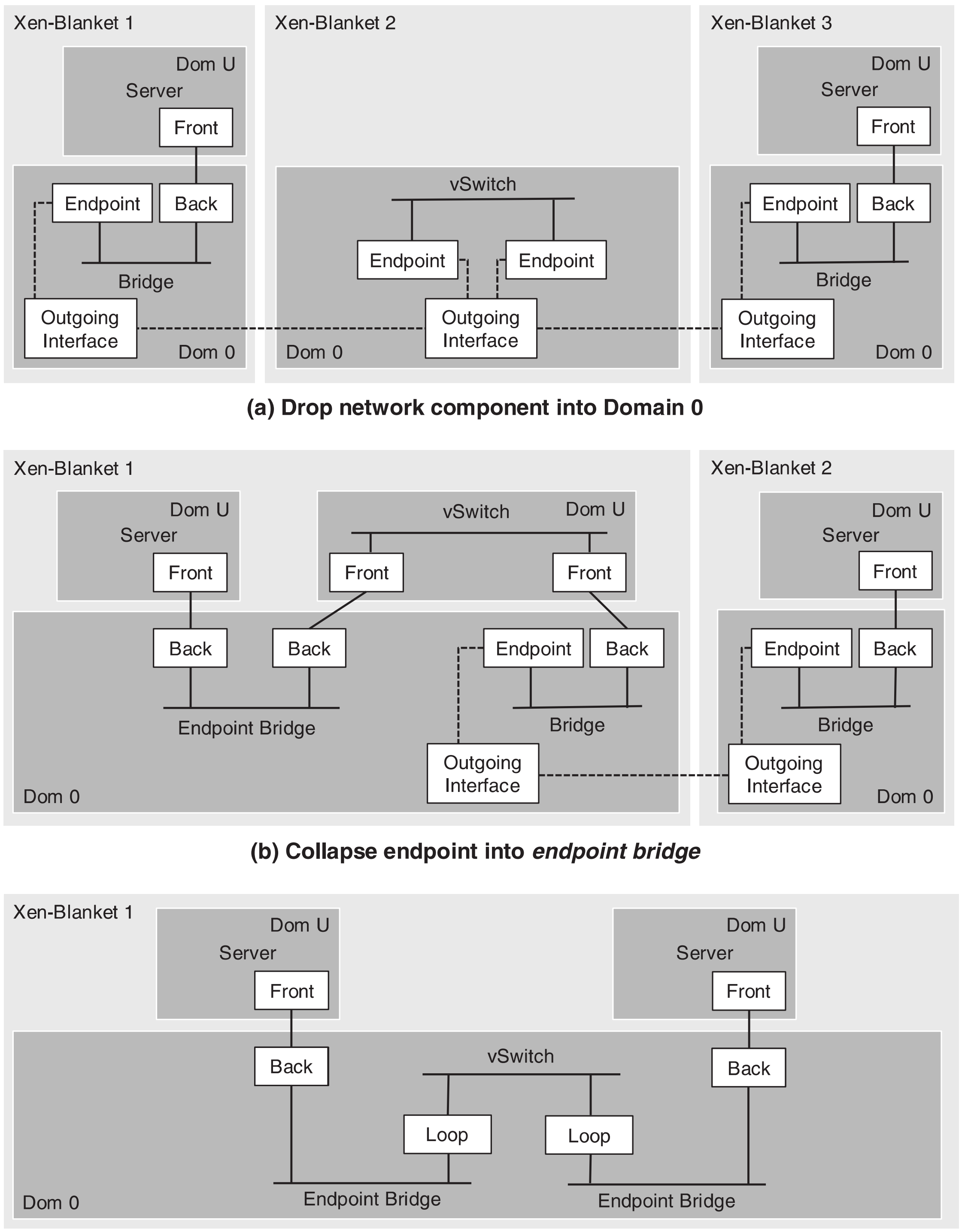
We relax the isolation of the VM for some network components, like the Linux software bridge. Instead, we implement the switch in Domain 0, as shown in Figure 5.8(a), and avoid two additional packet copies required to transfer a packet into and out of a guest VM. Because the switch is a simple component, migration is achieved by remotely reconfiguring Linux bridges with the `brctl` command, which is used to set up, maintain, and inspect the Ethernet bridge configuration in the Linux kernel [50]. However, complex network components cannot be as easily implemented or migrated in Domain 0.

5.2.3 Connectors

Connector endpoints are bound to back-end guest interfaces using a software bridge in the Xen-Blanket Domain 0, as shown in Figure 5.7. The back-end interface of a migrating VM will attach to a similar bridge, and the VM migration process is modified to ensure that the local endpoint will be moved and attached to the new bridge upon the completion of the VM migration operation.

Endpoint tunneling is performed via the endpoint manager kernel module. We implemented tunneling in a Domain 0 kernel module to reduce the number of context switches required when compared to a userspace implementation, such as `vtun` [47], a popular Linux tunneling solution, which uses the network tap, tunnel (TAP/TUN) [100] mechanism. The TAP/TUN mechanism allows

[2]Providing an efficient mechanism to access VM disk images from any cloud is a subject of future work.



(a) Drop network component into Domain 0

(b) Collapse endpoint into *endpoint bridge*

(c) Collapse endpoints and drop network components info Domain 0

Figure 5.8: Implementation optimizations

```
/* create a new endpoint N on instance A that receives
   packets on Adr_A and sends packets to Adr_B */
echo "<N> <Adr_A> <Adr_B>" > /proc/vw/create
/* return current endpoint config */
cat /proc/vw/<N>
/* update endpoint send config (e.g. other end migrated) */
echo "<Adr_C>" > /proc/vw/<N>
/* destroy the endpoint (e.g. this end migrated) */
echo "<N>" > /proc/vw/destroy
```

Figure 5.9: The proc interface to control VirtualWire endpoints.

packets from the kernel to be routed to userspace for processing. However, a kernel implementation is especially important on the Xen-Blanket because nested virtualization magnifies the cost of context switches [155]. Inside the kernel module, each endpoint has an associated socket and thread to listen for UDP packets received on the external interface, but destined for the endpoint. Outgoing packets are intercepted and encapsulated in UDP packets as described in Section 5.1.2, then sent through Domain 0's external interface.

If both endpoints of a connector are co-located on the same Xen-Blanket instance, the endpoint manager instructs the endpoints to forgo encapsulation. Instead, the back-end virtual interfaces from the network components are connected through a software bridge in Domain 0, called an *endpoint bridge*. The configuration after collapsing one endpoint is depicted in Figure 5.8(b). Endpoints are collapsed even when a virtual network component is implemented in Domain 0. Figure 5.8(c) shows the configuration if all three VMs are co-located and the switch is implemented in Domain 0.

5.2.4 Management and Configuration

We implemented an interface in the `/proc` filesystem in the endpoint manager kernel module, shown in Figure 5.9, through which endpoints are controlled. This interface provides a convenient mechanism to create and destroy them as needed. It is also used by the VM migration process to update endpoints.

We modified the Xen live VM migration process [57] to include endpoint migration. Usually, during VM migration, a VM container is created at the destination host. Then, memory is copied in several rounds before the VM is stopped, copied, and resumed at the destination. In VirtualWire, a migrating VM has a number of endpoints directly connected to its back-end interfaces. When a new VM container is created at the destination host, endpoints are created on the destination host, each of which are configured to send encapsulated packets to their sister endpoint. After the VM is stopped and copied to the destination, the configuration of every sister endpoint is updated to send encapsulated packets to the new endpoints; the original endpoints are deleted.

Unlike live VM migration within a traditional subnet, a migrating VM does not end up behind a different switch port from the perspective of the logical network. Thus, no unsolicited Address Resolution Protocol (ARP) messages need to be generated. The VM migration process is identical even when VMs migrate across subnets.

5.2.5 Gateway

We implement a VirtualWire Gateway with a physical aggregation switch attached to a general purpose physical server. The aggregation switch provides the large number of physical ports required to replace port-heavy physical network components. The server hosts a software endpoint implementation, identical to that on any other server, that uses an uplink to the network to communicate with other VirutalWire components.

5.3 Evaluation

In this section, we evaluate VirtualWire, first evaluating the performance of a 3-tier enterprise application migrated to Amazon EC2. Then, we demonstrate independence of the network from physical location by performing live migration of VMs and network components from our local environment to Amazon EC2. Finally, we describe microbenchmarks and the performance of VirtualWire.

5.3.1 Experimental Setup

To evaluate VirtualWire, we use resources at our local institution as well as from Amazon EC2.

Local Testbed. In our local environment, we use up to two physical machines connected by a 1 Gbps network, each containing two six-core 2.93 GHz processors, 24 GB of memory, and four 1 TB disks. Each machine runs Xen with 2

VCPUs and 4 GB of memory dedicated to Domain 0. On this underlying system, we use HVM instances configured with 22 VCPUs and 12 GB of memory to run Xen-Blanket and VirtualWire.

Amazon EC2. In Amazon EC2, we use up to six Cluster Compute Quadruple Extra Large instances, each with 23 GB memory, 33.5 EC2 Compute Units, 1690 GB of local instance storage, a 64-bit platform, and a 10 Gigabit Ethernet between instances.

Both environments use the Xen-Blanket patches for nested virtualization. The Xen-Blanket Domain 0 is configured with 8 VCPUs and 4 GB of memory. All guests (DomUs) are paravirtualized instances configured with 4 VCPUs and 2 GB of memory, and can run either on a single layer of virtualization in the local environment or nested within a Xen-Blanket instance. An NFS server running at our local setup provides VM disk images; VMs on Amazon EC2 access the NFS server through a VPN tunnel. For most of the experiments, we use `netperf` to generate 1400 byte packets in UDP_STREAM and UDP_RR modes for throughput and latency measurements, respectively.

5.3.2 Migrating an Enterprise Application

In this subsection, we use VirtualWire to facilitate the migration of an application to the cloud and quantify the performance implications. As an application, we run the RUBiS [66] benchmark with three VMs representing the Web tier, application server tier, and database tier, respectively. To better represent the

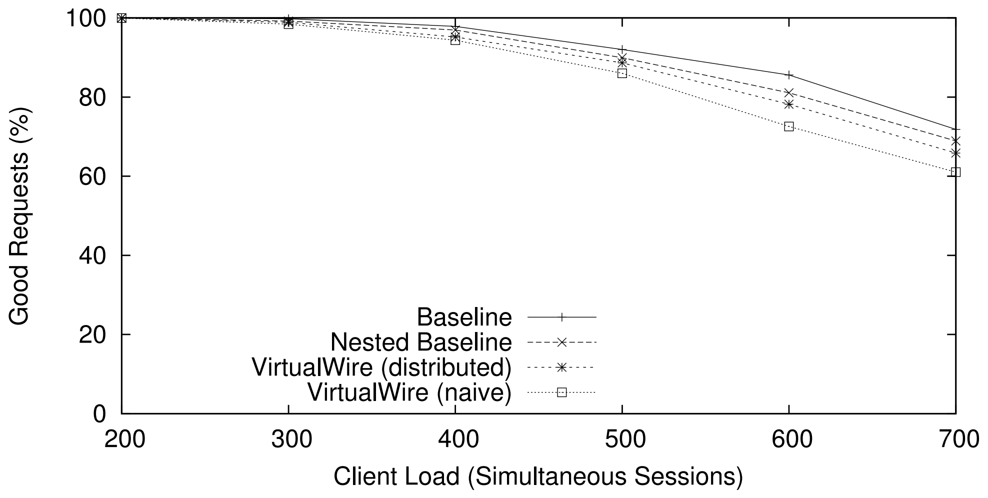


Figure 5.10: Performance of migrated RUBiS application

complexity that can arise in enterprise applications, we add two VMs running software firewalls (`iptables` [12]) in between each tier. Like many real applications, the VMs each have a hardcoded configuration—route table entries and IP addresses—that complicates their migration to the cloud using traditional methods.

Figure 5.10 shows the number of "good requests" (latency within 2000 ms) using the RUBiS application under various client load (simultaneous sessions). We examine four scenarios. The first two scenarios require manual modifications to the network configuration of each VM, but represent best-case scenarios in terms of application performance. The *baseline* is obtained by running each VM directly in an Amazon EC2 medium instance. The *nested baseline* runs an identically provisioned VM inside a Xen-Blanket instance, and therefore represents a best-case given nested virtualization overheads.

The other two scenarios are configurations of VirtualWire and *require no mod-*

ifications to the guest VMs. The first scenario, *VirtualWire naïve*, introduces a sixth VM that runs a virtual switch. Connectors are configured between each of the five VMs and the switch VM. The second scenario, *VirtualWire distributed*, replaces the switch VM with a distributed switch component as discussed in Section 5.1.5 and further evaluated in Section 5.3.6. In this configuration, VirtualWire achieves within 4% of the nested baseline and 9% of the baseline. User control over the cloud network enables enterprise workloads with customized network configurations, such as the RUBiS workload evaluated in this subsection, to run with good performance without requiring modifications.

5.3.3 Live Migration

In this subsection, we evaluate the ability of components and servers in VirtualWire to undergo live VM migration while maintaining the virtual network topology both within a single cloud and between clouds. Because we are unaware of other systems that enable cross-provider live migration, we do not compare the performance of migration in VirtualWire to other systems. For these experiments, we migrate a VM that is continuously receiving `netperf` UDP throughput benchmark traffic from another (initially co-located) VM. The network topology includes a virtual switch, also co-located, between the two VMs and implemented in Domain 0 for performance. For each experiment, we report the average (and standard deviation) application downtime and the total time the migration operation took to complete from 6 identical runs. Application downtime is calculated by examining periodic output from the `netperf` benchmark.[3]

[3]The frequency of the periodic output is set to 0.1 s, so we cannot measure downtime less than 0.1 s.

	Downtime	Duration
Non-nested	0.7 [0.4]	19.86 [0.2]
Nested	1.0 [0.5]	20.04 [0.1]
Nested w/ endpoints	1.3 [0.5]	20.13 [0.1]

Table 5.1: Mean [w/standard deviation] downtime (s) and total duration (s) for local live VM migration

	Downtime	Duration
Local to Local	1.3 [0.5]	20.13 [0.1]
EC2 to EC2	1.9 [0.3]	10.69 [0.6]
Local to EC2	2.8 [1.2]	162.25 [150.0]

Table 5.2: Mean [w/standard deviation] downtime (s) and total duration (s) for live VM migration

The performance time of live migration of a VM (receiving `netperf` UDP traffic) between two machines in our local setup is shown in Table 5.1. The VM and its `netperf` partner VM were both run on a single layer of virtualization (*non-nested*) or nested setup (*nested* and *nested w/ endpoints*) before and after the migration. To isolate the nesting overhead from endpoint migration overhead, VirtualWire connectors are not used between the two VMs in the *non-nested* and *nested* experiments; instead, the physical network is bridged. We find a 43% increase in downtime and an 0.9% increase in total duration due to nesting. The added task of migrating VirtualWire endpoints introduces an additional 43% increase in downtime (30% over nested), but a negligible increase in total duration.

Table 5.2 quantifies the performance of live migration across clouds while maintaining the virtual network topology using a VirtualWire connector. We compare the performance of single-cloud live migration within our local nested

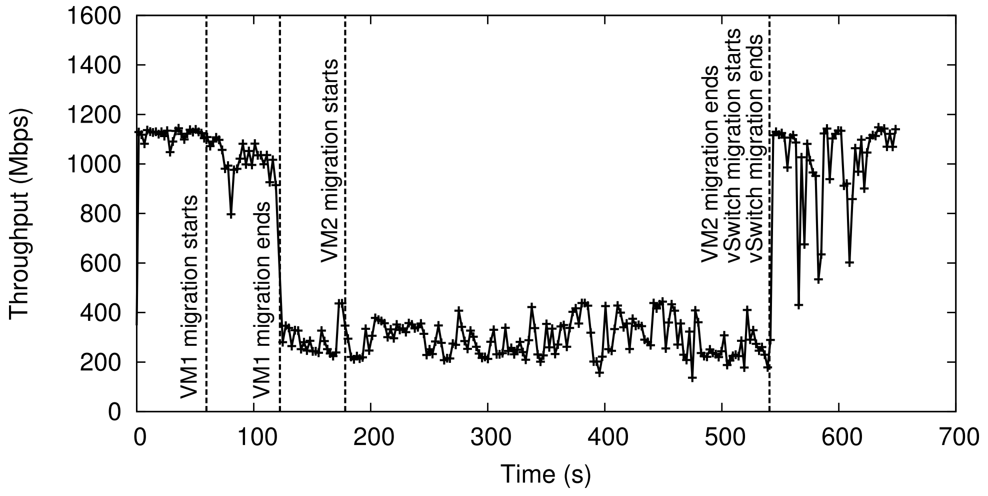


Figure 5.11: Throughput over time between two VMs migrating to Amazon EC2

setup (*Local to Local*) and within Amazon EC2 (*EC2 to EC2*) to multi-cloud migration between the two (*Local to EC2*). Within one cloud, local or EC2, the latency between the instances is within 0.3 ms, whereas across clouds it is about 10 ms. VPN overhead limits throughput across clouds to approximately 230 Mbps. The 10 Gbps network between our EC2 instances leads to significantly reduced total migration time when compared to local; however, the downtime was comparable. Live migration of a VM (receiving `netperf` UDP traffic) between our local nested setup and Amazon EC2 has a downtime of 2.8 s and a total duration of 162.25 s on average, but variation is high: the duration ranged from 48 s to 8 min. For an idle VM, the performance of the network between machines has little effect: the downtime during live migration between two local machines and from local to EC2 is 1.4 s on average.

We ran two more experiments,[4] shown in Figure 5.11 and Figure 5.12, to

[4]We could not measure both the throughput and latency from a single experiment using

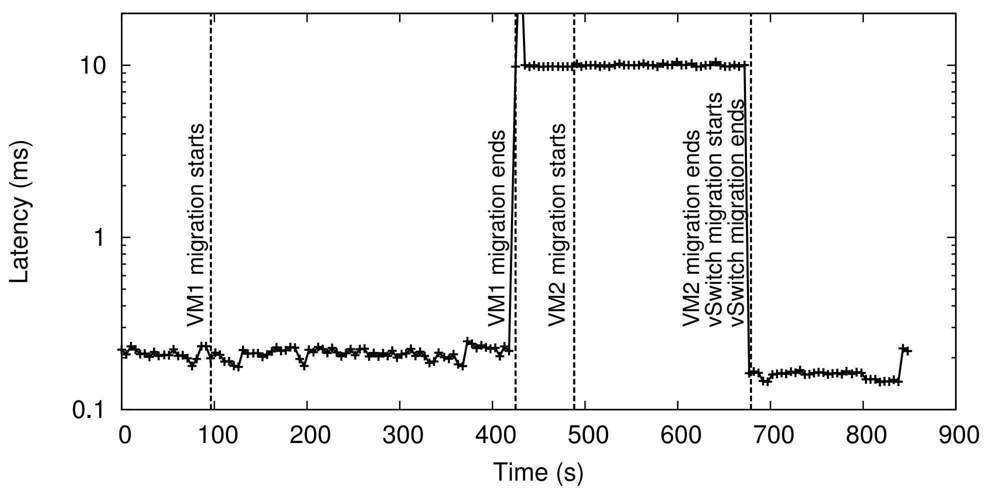


Figure 5.12: Latency over time between two VMs migrating to Amazon EC2.

identify the throughput and latency over time for the test deployment as the recipient VM and then the switch and the sender VM were live migrated to Amazon EC2, respectively. Migration of the switch did not affect throughput because its Domain 0 implementation was trivial to migrate. This experiment demonstrates that connectivity and network topology are maintained, independent of physical location. However, as expected, significant degradation in the throughput and latency occurs when not all components are co-located on the same cloud.

5.3.4 Microbenchmarks

Nesting Overhead. We isolate the overhead VirtualWire incurs due to Xen-Blanket nested virtualization using a `netperf` process in the Domain 0 of one

`netperf`.

	Latency	Agg. CPU Util.
Single Dom0	63 μs	61%
Single DomU	86 μs	88%
Nested Dom0	96 μs	97%
Nested DomU	210 μs	196%

Table 5.3: Network latency comparison in nested and non-nested virtualization settings

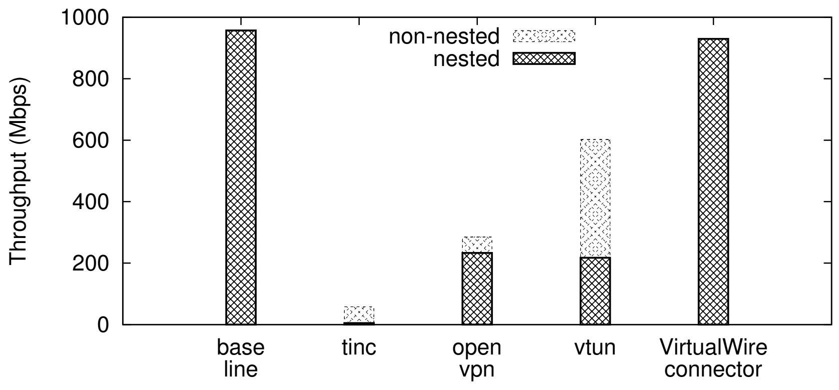


Figure 5.13: Tunnel throughput comparison

physical machine across the 1 Gbps link in the local setup. We identify the throughput, latency and CPU utilization when network packets are received by a single-layer Domain 0 (`single dom0`), inside a guest running on the first layer (`single domU`), on the Xen-Blanket Domain 0 (`nested dom0`), or inside a guest running on the Xen-Blanket (`nested domU`). Throughput measurements UDP throughput is maintained at line speed in all cases, while TCP throughput decreases by 6.7% for `nested domU`. However, as shown in Table 5.3, the latency increases sharply with a corresponding increase in CPU utilization due to extra packet copies and context switching.

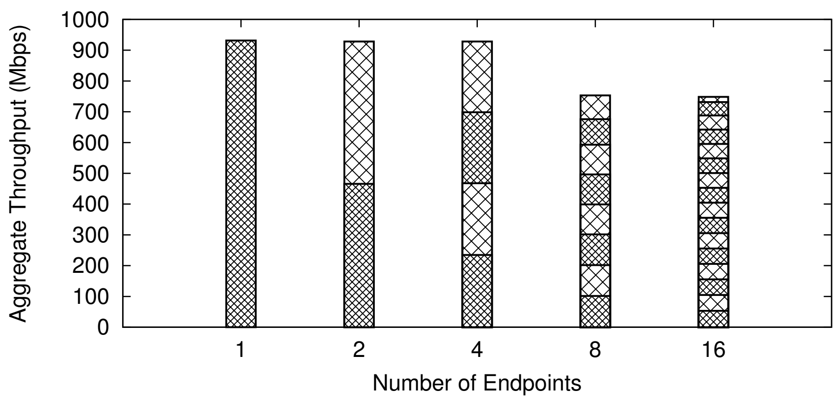


Figure 5.14: Scalability of VirtualWire endpoints in a single nested Domain 0

Connectors. VirtualWire connectors are implemented in a kernel module and therefore avoid the high penalty for context switches experienced in the Xen-Blanket. Figure 5.13 compares the performance of connectors with various software tunnel packages. `tinc` [20], OpenVPN [124], and `vtun` [47] are open-source VPN solutions that create tunnels between Linux hosts. On our local—both nested and non-nested—1 Gbps setup, we measure the UDP performance of a VM sending data to another VM through the tunnel. The baseline is a measurement of the direct (non-tunneled) link between the VMs. The kernel-module approach of VirtualWire connectors pays off: while connectors increase throughput by a factor of 1.55 over the popular open-source tunneling software `vtun` in a non-nested scenario, it achieves a factor of 3.28 improvement in a nested environment.

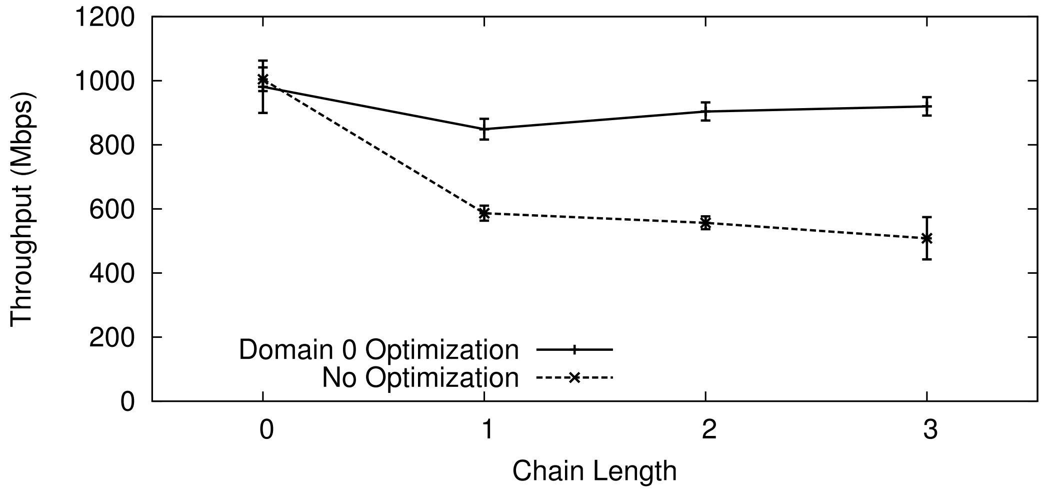
Endpoint Scalability. In VirtualWire, a network component with many ports, such as a switch or router corresponds to a DomU with many virtual network interfaces and many endpoints. All endpoints terminate at the same Domain 0

	Connector w/o Extender		Connector w/ Extender	
	Throughput	Avg. Latency	Throughput	Avg. Latency
Local to Local	929.52 Mbps	0.24 ms	233.00 Mbps	0.30 ms
EC2 to EC2	1012.80 Mbps	0.27 ms	230.83 Mbps	0.31 ms
Local to EC2	n/a	n/a	239.96 Mbps	10 ms

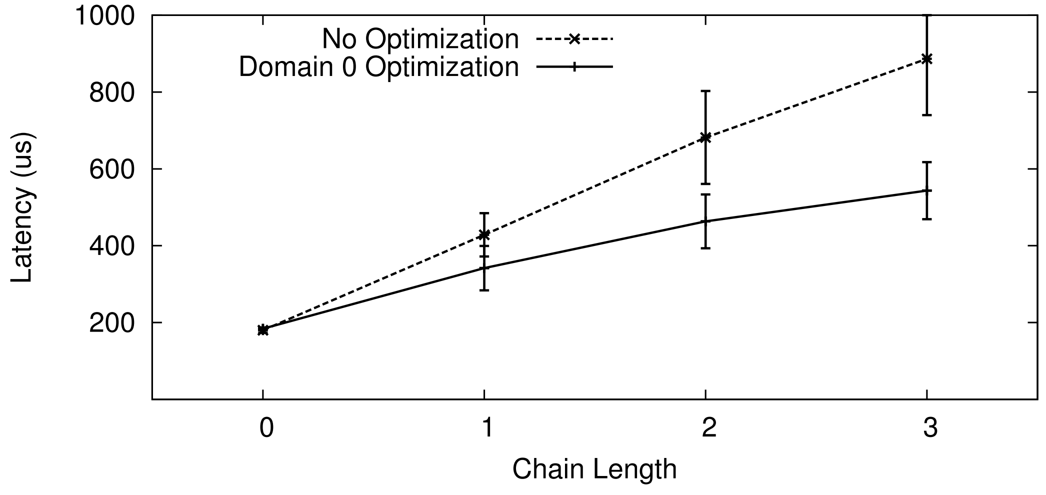
Table 5.4: Performance of connectors with and without extenders

and are connected to the DomU. Figure 5.14 shows scalability of endpoints in a single nested Domain 0. We measure endpoint scalability by increasing the number of interfaces with endpoints in the DomU and run several simultaneous `netperf` UDP throughput tests over each connector from another host. The 1 Gbps bandwidth becomes split between the individual connectors, but in aggregate remains high for at least 16 endpoints.

Extenders. VirtualWire uses OpenVPN [124] to create extenders so that connectors can span clouds. Table 5.4 shows the performance (measured with `netperf`) between instances on our local nested setup or EC2 with and without extenders. As previously discussed (Figure 5.13), OpenVPN introduces significant overhead; when used as an extender, throughput and average latency suffered by up to 25%. Across clouds, where extenders must be used, we also see higher latency. This result underscores the importance of co-locating heavily communicating VMs on a single cloud, rather than across clouds when possible.



(a) Throughput



(b) Latency

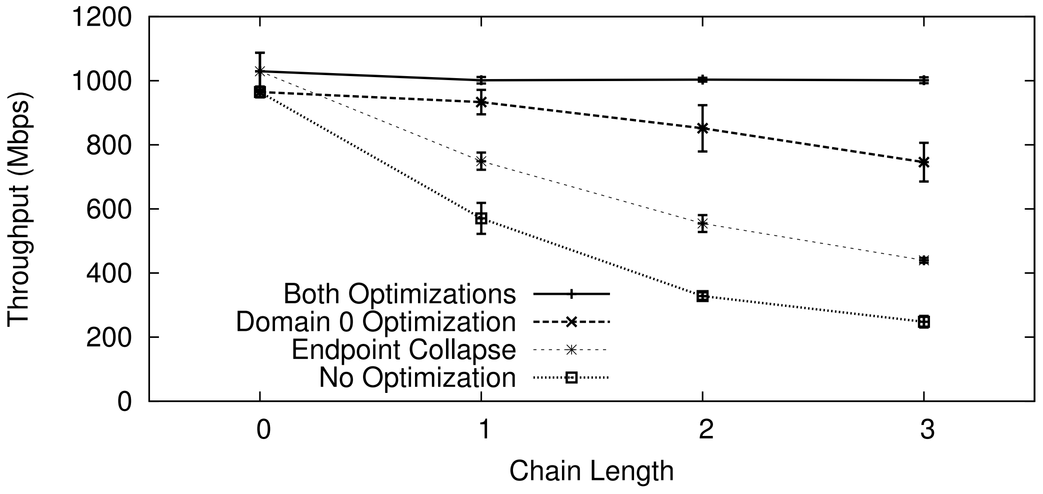
Figure 5.15: Effects of component chaining on throughput and latency

5.3.5 Component Chain Length

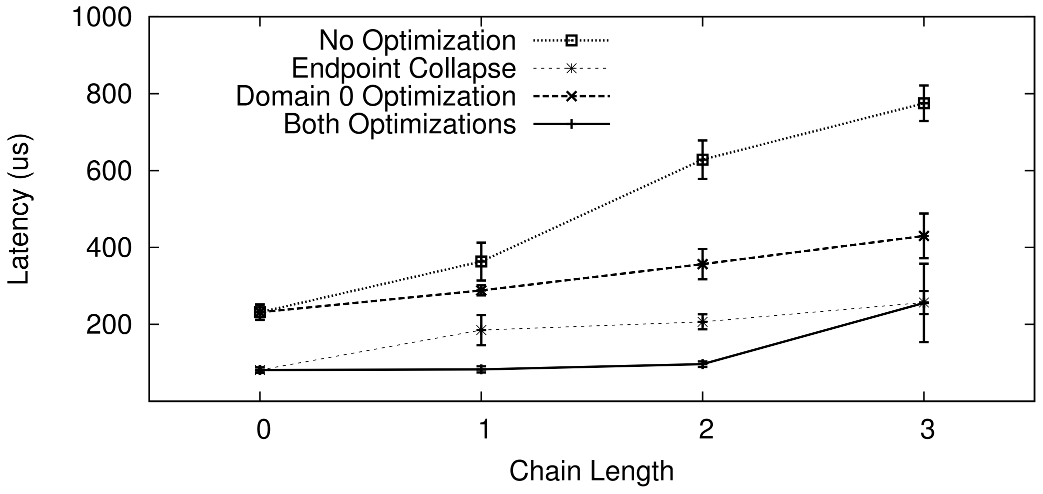
We examine the effects of chain-length within a VirtualWire network topology on throughput and latency with and without optimizations. We restrict our experiment to a single physical machine in the local environment to eliminate network congestion and jitter from the results. We vary the number of switches in the VirtualWire topology between the `netperf` endpoints. The switches are configured in a chain: each switch has two ports (thus, no cross traffic), while the server VMs have a single interface.

We first examine the case in which components are on different instances. We run five Xen-Blanket instances on a single physical machine, with 4 GB of memory and 4 VCPUs each, each hosting one VM. Figure 5.15(a) and Figure 5.15(b) show that, as expected, the throughput slightly decreases and the latency linearly increases as the chain length increases. By dropping all of the switch VMs into the Domain 0 of their respective Xen-Blanket instances, we found throughput was maintained with at least three switches on the path while latency was reduced from approximately 237 μs to 119 μs per switch.

In the case where components are co-located in a single instance, endpoints are collapsed and encapsulation is unnecessary. To evaluate this optimization, we run a single Xen-Blanket instance with 14 GB of memory and 22 VCPUs so that it can support up to five guests and Domain 0. Figure 5.16(a) and Figure 5.16(b) show the throughput and latency between the two servers with up to three switch VMs interposing on the packets in four configurations: either encapsulating or collapsing endpoints, each with DomU or Domain 0 switches. By avoiding encapsulation, endpoint collapse reduces latency by about 135 μs per switch. Combined with Domain 0 switches, good performance is maintained for all evaluated chain lengths. These results suggest that beyond co-locating heavily communicating components on the same cloud, VirtualWire derives significant benefit from co-locating components on the same instance.



(a) Throughput



(b) Latency

Figure 5.16: Effects of endpoint collapse and Domain 0 optimization in co-located component chains

5.3.6 Distributed Components

In this subsection, we demonstrate how distributed components can overcome performance limitations of centralized virtual network components. We run

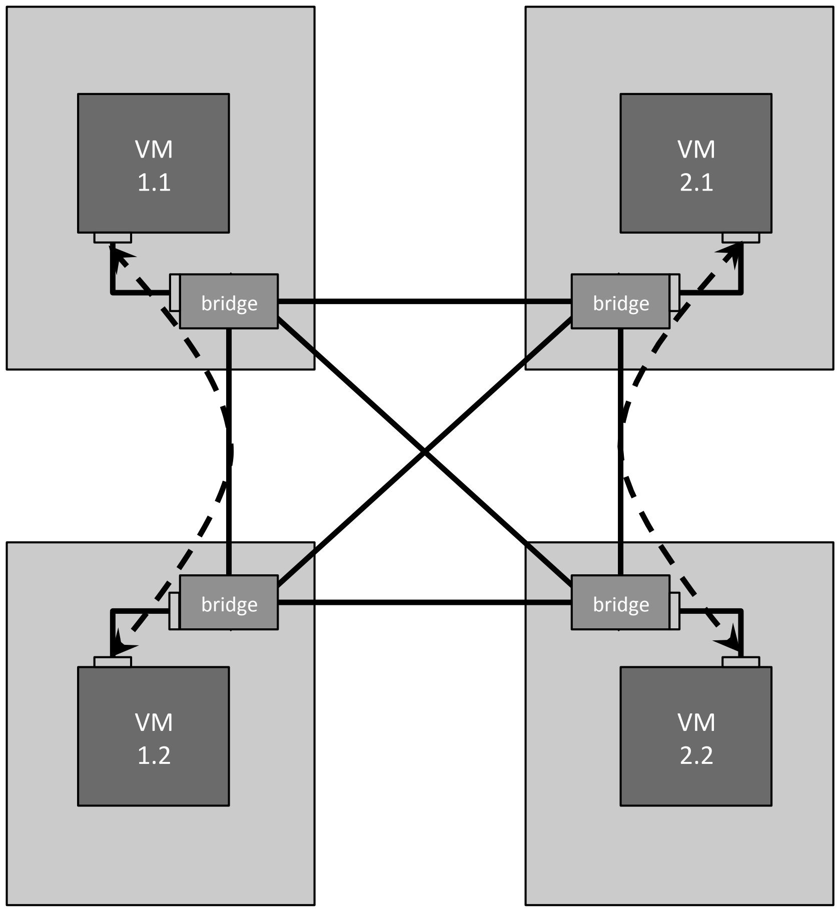


Figure 5.17: Proof-of-concept distributed virtual switch

`netperf` between four server VMs on four EC2 Xen-Blanket instances. All VMs are connected to a virtual switch component running in Domain 0 on a fifth EC2 Xen-Blanket instance. Traversing the switch, each pair of VMs can achieve throughput of 1.39 Gbps on average. If two flows are active at the same time, throughput is split between the two flows, dropping them to 720.5 Mbps each.

We demonstrate that a proof-of-concept distributed virtual switch (DVS) can overcome this performance limitation. The DVS, shown in Figure 5.17, is constructed by running a Linux bridge on each of the four EC2 Xen-Blanket instances and manually linking them (with connectors) and configuring them to act as a single switch. The configuration was done with `ebtables` [6], a tool that enables transparent filtering of network traffic passing through a Linux bridge. Using the DVS and avoiding the centralized component as a bottleneck, each pair of VMs achieve throughput of 1.77 Gbps on average regardless of whether one or two flows are active. By eliminating centralized components in performance critical parts of the network, an enterprise workload can run efficiently, while maintaining the control logic for the cloud network.

5.4 Summary

At the time of writing this dissertation, clouds lack the support for complex enterprise networks, which are essential for constructing superclouds. In this chapter, we exploited cloud extensibility to provide support for enterprise networks in existing third party clouds.

We have presented VirtualWire, a virtual networking system that facilitates the migration of enterprise deployments to the cloud by enabling cloud users to implement the control logic of their virtual network. We have migrated a 3-tier application and its complex network topology to EC2, unmodified, maintaining within 9% of the performance of a native deployment. With VirtualWire, Servers and network components can live migrate to EC2, experiencing as low as 1.4 s of downtime. Although virtualized equivalents of software components affect network performance, through a combination of virtual network component placement, incremental introduction of distributed network components, and adoption of ideas from state-of-the-art network virtualization architectures, VirtualWire is an efficient and practical first step towards the goal of enterprise migration to the cloud.

VirtualWire is a crucial step towards superclouds. By using cloud extensibility, in particular, with the Xen-Blanket and VirtualWire, superclouds can support enterprise applications and their networks. In the next chapter, we will describe how to efficiently utilize memory in superclouds.

CHAPTER 6

TOWARDS EFFICIENT CLOUD RESOURCE UTILIZATION: OVERDRIVER

Cloud extensibility affords a cloud user the opportunity to implement new techniques to achieve high utilization of resources in superclouds. Low resource utilization in enterprise deployments is due to the classical model of overprovisioning, in which each VM is allocated enough resources to support relatively rare peak load conditions. This problem is not unique to superclouds, but also affects cloud providers.

Oversubscription of resources can increase utilization, however, without complete knowledge of all future VM resource usage, one or more VMs will likely experience *overload*. Overload is a situation in which the resource requirements of a VM exceed its resource allocation. While overload can happen with respect to any resource machine, we focus on memory overload. The availability of memory contributes to limits on VM density (the number of VMs per machine) and consolidation and as such, is an attractive resource for oversubscription. In addition, recent pricing data for different configurations in Amazon's Elastic Compute Cloud (EC2) indicate that memory is twice as expensive as EC2 Compute Units.[1] However, memory is typically not oversubscribed in practice as much as other resources, like CPU, because memory overload is particularly

[1]We use Amazon's pricing data from November 2010 as input parameters for a series of linear equations of the form $p_m \times m_i + p_c \times c_i + p_s \times s_i = price_i$, where m_i, c_i, s_i, and $price_i$ are pricing data for configuration i for memory, EC2 Compute Units, storage, and hourly cost, respectively. Also, p_m, p_c, and p_s are the unknown unit cost of memory, EC2 Compute Units, and storage, respectively. Approximate solutions for the above equations consistently show that memory is twice as expensive as EC2 Compute Units. Particularly, the average hourly unit cost for memory is 0.019 cents/GB. This is in contrast with an average hourly unit cost of 0.008 cents/EC2 Compute Unit and 0.0002 cents/GB of storage.

devastating to application performance. Memory overload can be characterized by one or more VMs swapping its memory pages out to disk, resulting in severely degraded performance. Whereas overload on the CPU or disk result in the hardware operating at full speed with contention introducing some performance loss, memory overload includes large overheads, sometimes to the point of thrashing, in which no progress can be made.

Unless oversubscribed clouds (or superclouds) can manage memory overload, memory will be a bottleneck that limits the VM density that can be achieved. In this chapter we investigate techniques to oversubscribe memory while mitigating VM performance degradation due to memory overload.

As described in Section 2.1.3, there are two types of memory overload encountered in a cloud deployment caused by oversubscription. Through analysis of data center logs from well-provisioned enterprise data centers, we conclude that there is ample opportunity for memory oversubscription to be employed: only 28% of machines experience any overload whatsoever, an average of 1.76 servers experience overload at the same time, and 71% of overloads last at most only long enough for one measurement period (Section 2.1.3). Experimenting with higher degrees of oversubscription on a Web server under a realistic client load, we find, while the likelihood of overload can increase to 16% for a reasonably oversubscribed VM, the duration of overload varies. While overload occasionally consists of long, sustained periods of thrashing to the disk, this is not the common case: 88.1% of overloads are less than 2 minutes long (Section 2.1.3). The fact that memory overload in an oversubscribed environment is a continuum, rather than entirely sustained or transient, suggests that different types of overload may be best addressed with a different mitigation strategy/technique.

In this chapter, we identify tradeoffs between memory overload mitigation strategies. Any overload mitigation strategy will have an effect on application performance and will introduce some overhead on the cloud itself. Existing migration-based strategies [37, 94, 138, 158] address memory overload by reconfiguring the VM to resource mapping such that every VM has adequate memory and no VMs are overloaded. VM migration is a heavyweight process, best suited to handle predictable or sustained overloads. The overload continuum points to a class of transient overloads that are not covered by migration. Instead, we propose a new application of network memory [27,36,62,71,86,106,123] to manage overload, called *cooperative swap*. Cooperative swap sends swap pages from overloaded VMs to memory servers across the network, instead of to the relatively poor performing disk. Unlike migration, cooperative swap is a lightweight process, best suited to handle unpredictable or transient overloads. Each technique, cooperative swap or migration, carries different costs, and addresses a different section of the overload continuum, but neither technique can manage all types of overload. Both techniques can be used in superclouds due to cloud extensibility.

Finally, we present *Overdriver*, a system that adaptively chooses between VM migration and cooperative swap to manage a full continuum of sustained and transient overloads. Overdriver uses a threshold-based mechanism that actively monitors the duration of overload in order to decide when to initiate VM migration. The thresholds are adjusted based on VM-specific probability overload profiles, which Overdriver learns dynamically. Overdriver's adaptation reduces potential application performance degradation, while ensuring the chance of unnecessary migration operations remains low.

For the mitigation techniques to work, excess space is required in the cloud, whether it is as a target for migration or a page repository for cooperative swap. Overdriver aggregates the VM-specific probability overload profiles over a large number of VMs in order to provide insight into the amount and distribution of excess space in the cloud. We have implemented Overdriver and evaluated it when compared to either technique on its own to show that Overdriver successfully takes advantage of the overload continuum, mitigating all overloads within 8% of well-provisioned performance. Furthermore, under reasonable oversubscription ratios, where transient overload constitutes the vast majority of overloads, Overdriver requires 83% of the excess space and generates 46% of the network traffic of a migration-only approach.

To summarize, this chapter makes three main contributions:

- We observe the overload continuum: memory overloads encountered in enterprise deployments include, and will likely continue to include, both transient and sustained bursts, although an overwhelming majority will be transient.
- We show there are tradeoffs between memory overload mitigation strategies that are impacted by the overload continuum, and propose a new application of network memory, called cooperative swap, to address transient overloads.
- We design, implement and evaluate Overdriver, a system that adapts to handle the entire memory overload continuum.

The rest of this chapter is organized as follows. Section 6.1 describes the tradeoffs between mitigation techniques under different types of memory overload. Section 6.2 describes the design and implementation of Overdriver and

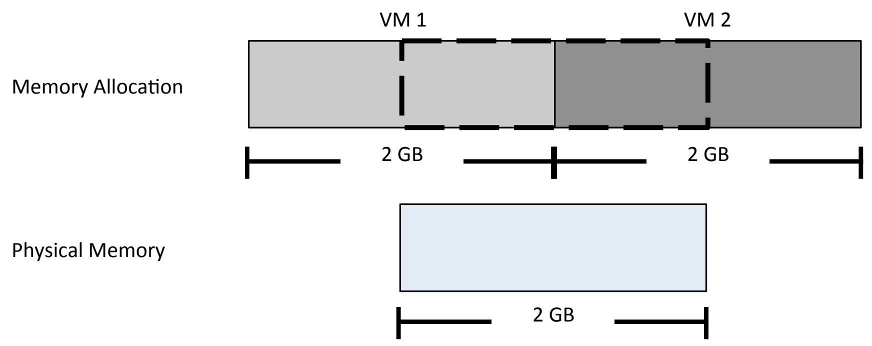


Figure 6.1: Memory oversubscription

how it adaptively mitigates the damage of all types of memory overload. Section 6.3 quantifies Overdriver's effects on the application and the cloud, comparing it to systems that do not adapt to the tradeoffs caused by the overload continuum. Finally, Section 6.4 summarizes the paper.

6.1 Overload Mitigation

As described in Section 2.1.3, a VM is *overloaded* if the amount of memory allocated to the VM is insufficient to support the working set of the application component within the VM. We focus on mitigating overload due to oversubscription, in the case where the cloud dedicates less memory to a VM than it requested. Figure 6.1 shows an oversubscribed machine. The machine has 2 GB of physical memory and has assigned two VMs with 2 GB of memory allocated to each VM. This machine is oversubscribed—each VM only has 1 GB of physical memory backing its allotted 2 GB memory. If each VM only requires 1 GB of memory most of the time, this memory allocation is sufficient and performance will not degrade due to oversubscription. However, if resource utilization of one or both VMs increases such that they require more than 2 GB of memory in aggregate, one or both will experience memory overload, and performance will be significantly degraded.

In this section, we discuss strategies to mitigate overload. When considering an overload mitigation strategy, the cost of the strategy can be measured in two dimensions: the effect to the application that is experiencing overload and the effect to the supporting cloud infrastructure caused by overhead intrinsic to the strategy.

Application effects refer to the performance of the application that is experiencing overload. Ideally, a mitigation strategy would sustain application response time, throughput, or other performance metrics throughout the overload, so that the VM is unaware that it even took place. Cloud effects include the overhead or contention introduced by the overload and the mitigation strategy. Ideally, the resources used during the overload are no more than what would have been actively used if no oversubscription was in effect. In this section, we discuss the application and cloud effects of two different mitigation techniques: VM migration and network memory.

6.1.1 VM Migration

Existing techniques to mitigate overload consist largely of VM migration techniques that address overload by reconfiguring the VM to machine mapping such that every VM has adequate memory and no VMs are overloaded [37, 94, 138,158] (i.e. the VM memory allocation is increased, possibly up to the amount originally requested by the cloud user). VM migration is a relatively heavy-weight solution: it incurs delays before it goes into effect, and has a high, fixed impact to the network infrastructure, regardless of the transience of the overload. For these reasons, migration strategies are usually designed to be proactive. Trending resource utilization, predicting overload, and placement strategies to minimize future overloads are key components of a migration strategy. Despite these challenges, VM migration strategies are popular because once migrations complete and hotspots are eliminated, all application components will have adequate memory to return to the performance they would have enjoyed without oversubscription.

Live migration boasts very low downtime: as low as 60ms for the migrating VM [57]. However, in the best case, the time until completion of migration is dependent on the speed at which the entire memory footprint of the migrating VM can be sent over the network. In many settings, further migration delays are likely to arise from the complexity of migration decisions. In addition to computing VM placements for resource allocation, migration decisions may require analysis of new and old network patterns, hardware compatibility lists, licensing constraints, security policies and zoning issues in the cloud. Even worse, a single enterprise workload is typically made up of an elaborate VM deployment architecture, containing load balancers, worker replicas, and database backends, that may experience correlated load spikes [136]. A migration decision, in such case, has to consider the whole application ecosystem, rather than individual VMs. This complexity can be reason enough to require sign off by a human operator. Finally, in the worst case, the infrastructure required for efficient VM migration may not be available, including a shared storage infrastructure, such as a SAN, and a networking infrastructure that is migration aware.

The effect that a migration strategy has on the cloud is mostly measured in terms of network impact. Typically, VM migration involves sending the entire memory footprint of the VM or more in the case of live migration (See Appendix B). This cost is fixed, regardless of the characteristics of the overload that may be occurring[2]. A fixed cost is not necessarily a bad thing, especially when considering long, sustained overloads, in which the fixed cost acts as an upper bound to the network overhead. Migration also requires a high, fixed amount of resources to be available at the target. The target must have enough resources to support a full VM, including CPU and enough memory to support the desired allocation for the migrating VM.

Coupled with the unbeatable performance of local memory available to a VM after a migration strategy completes, the fixed cost to the cloud makes migration an attractive strategy to handle both predictable load increases, as well as sustained overloads.

6.1.2 Network Memory

Two important results from Section 2.1.3 highlight a gap in the solution space that existing migration-based solutions do not address. First, overload follows unpredictable patterns. This indicates that, unless oversubscription policies are extremely conservative, reactive strategies are necessary. Second, transient

[2]Live VM migration will send more than the few hundred megabytes to tens of gigabytes of data comprising the VM's memory footprint because it must re-transmit the written working set in iterations. However, the network cost of live migration strategies is still relatively fixed because live migration strategies impose a limit on the number of iterations.

	Tput (MB/s)		Latency (μs)	
	Read	Write	Read	Write
Network Mem	118	43.3	119 ± 51	25.45 ± .04
Local Disk	54.56	4.66	451 ± 95	24.85 ± .05

Table 6.1: Network memory vs local disk performance.

overload is, and will likely continue to be, the most common type of overload. As described in Section 6.1.1, migration, when used reactively, has high delays, and high network overhead for the relatively short lived transient overloads. There is an opportunity to consider reactive strategies that focus on transient overloads.

Network memory is known to perform much faster than disk [27, 36, 62, 71, 86, 106, 123], especially on fast networks, and has been applied to nearly every level of a system. We propose and investigate *cooperative swap*, an application of network memory as an overload mitigation solution in which VM swap pages are written to and read from page repositories across the network to supplement the memory allocation of an overloaded VM. Cooperative swap is entirely reactive, and begins to mitigate overload when the very first swap page is written out by the overloaded VM[3]. Its impact on the network is dependent on the duration of overload. However, using cooperative swap does not match the performance of local memory over the long term.

Table 6.1 shows the relative throughput of disk I/O vs. network memory. These numbers were computed from running a sequential disk dump (using the standard Unix tool `dd`) between the system (`/dev/zero` or `/dev/null`) and a hard disk (`/dev/sda1`) versus a Linux network block device (`/dev/nbd0`)

[3]We do not use cooperative swap unless the VM has a reduced memory allocation due to oversubscription. Otherwise, overload is the responsibility of the VM.

attached to a ramdisk across the network. The network connecting physical machines is 1 Gbps so can sustain a maximum rate of 125 MB/s. Both are block devices and so should be affected by Linux equally in terms of overhead[4]. Each result is the average of 20 runs of writing or reading 1 GB for the throughput test and one 4 KB page for the latency test. Our setup consists of a server with a Xeon 5120 1.86 GHz dual core processor and a 133 GB SCSI disk with 7200 rpm. We find that network memory is significantly faster than disk. Using cooperative swap, short bursts of paging complete faster, allowing it to maintain application performance through transient bursts of overload. However, cooperative swap does not restore local memory and so cannot restore application performance to where a cloud user would expect if oversubscription was not taking place.

Cooperative swap affects the cloud by sending memory pages back and forth across the network. The amount of memory read or written from the network is dependent on the length of the overload. This means that cooperative swap is relatively cheap in terms of network overhead for transient overloads, but could generate unbounded amounts of network traffic for sustained overloads. The amount of pages that must be available from the remote page repositories is also dependent on the duration of overload, however, this number is bounded by the size of the VM's originally requested memory size.

Reactivity, proportional network overhead to overload duration, and long-term performance issues make cooperative swap an attractive solution for unpredictable overloads and transient overloads, filling the gap in the solution space left by existing migration strategies.

[4]Care was taken to eliminate caching effects as much as possible for the latency tests, dropping the caches and writing 500 MB to the device between each latency test.

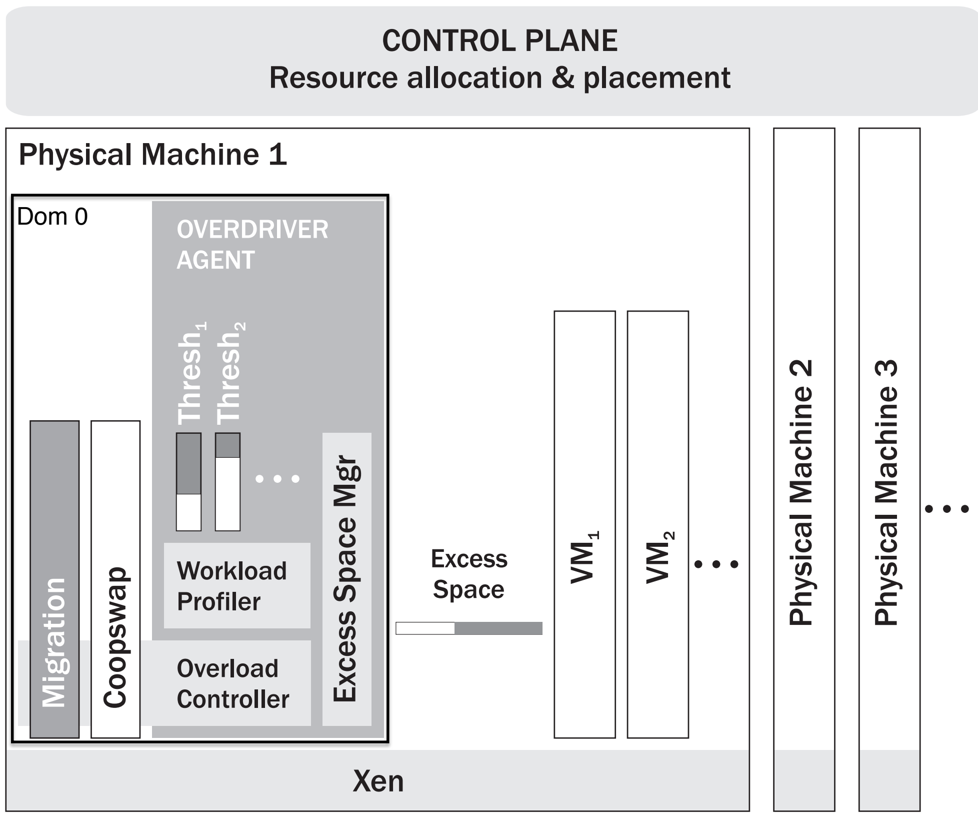


Figure 6.2: The memory resources in the cloud are split into space for VMs and excess space to mitigate overload, of which there are two types: cooperative swap page repositories, and future migration targets. Resource allocation and placement of all resources is performed by the control plane.

6.2 Overdriver

As seen in Section 6.1, the overload continuum leads to tradeoffs between mitigation strategies. We design a system that exploits these tradeoffs to manage overload called *Overdriver*. Overdriver adaptively decides to use cooperative swap for transient overloads and migration for sustained overloads.

Figure 6.2 shows the high-level components in Overdriver. Each physical machine (or Xen-Blanket instance) in the cloud (or supercloud) runs a hypervisor and supports some number of guest VMs. Each physical machine also runs an *Overdriver agent* in its control domain (Domain 0) that monitors the memory overload behavior of the local guest VMs by measuring paging rate. Within the Overdriver agent, the *workload profiler* locally learns a memory overload probability profile for each VM similar to that in Figure 2.3, which it then uses to set adaptive thresholds on the length at which the overload is classified as sustained. Using these thresholds, the *overload controller* decides which mitigation strategy to employ for a given overload. A transient overload, defined as an overload whose duration has not yet passed the threshold, is mitigated by redirecting the overloaded VM's swap pages to cooperative swap page repositories, rather than to the local disk. If the transient overload becomes sustained, characterized when the duration of the overload exceeds the threshold, the overload controller initiates a migration operation, wherein one or more VMs are migrated to perform an increase of the memory allocation of the overloaded VM.

For either mitigation strategy to be useful, there must be some *excess space* in the cloud. The *excess space manager* on each physical machine (or Xen-Blanket instance) is responsible for dedicating some of the local excess space to act as a target for migration, and some to act as a page repository for cooperative swap from overloaded VMs throughout the cloud. The actual allocation and placement of VMs and excess space, including page repositories, throughout the cloud is performed by the *control plane*, which is similar to the one in [158]. The control plane may run a proactive migration algorithm to avoid hotspots [158] or to consolidate VMs [94].

6.2.1 Deciding to Migrate VMs

Overdriver uses a threshold on the duration of overload to determine when to classify an overload as sustained and to employ migration. Choosing an appropriate threshold is difficult, and a good choice depends on the overload characteristics of the application VM. At one extreme, a very low threshold approaches a migration-based overload mitigation strategy, with good application performance but high cost to the cloud itself. On the other hand, setting the threshold to be too large approaches a network memory-based strategy, with lower application performance but also lower cost to the cloud. Intuitively, a good choice for the threshold would be high enough that a vast majority of the overloads do not require migration while being low enough that performance does not suffer too much. A profile of the application VMs, including the probabilities of each duration of overload, can help determine a reasonable value for the threshold.

In reality, an application VM profile that describes the probability of overload having a particular duration is not available. However, rather than selecting a single threshold for all VMs, the overload manager starts with a fixed threshold, then relies on the workload profiler to learn the probabilities of various duration overloads to give a basis for reducing the threshold.

There are two challenges in learning a probability profile. First, in the case of sustained overload where migration is employed, the overloaded VM will be granted additional resources that will fundamentally change its overload behavior. In particular, the probability profile of the application VM will change, requiring the learning process to start anew. Second, the learned probability profile takes some time to converge. For example, we attempted to learn the probability profile for a VM allocated 640 MB from the SPECweb2009 experiments in Section 2.1.3. For this scenario, at least 25 sustained overloads must be witnessed before the learned profile becomes reasonable. Since the local profile is useless after a migration takes place, the 25 sustained overloads must be endured using only cooperative swap in order to learn a complete profile.

Despite these challenges, focusing only on the different durations of transient overloads, the workload profiler learns enough to reduce the threshold without resorting to excessive migration. The workload profiler maintains a list of buckets for each VM, corresponding to possible transient overload durations. As the paging rates of a VM are monitored, the profiler maintains a count for the number of times an overload of a specified duration is encountered in the appropriate bucket. Once the number of measurements exceeds a base amount, we begin to estimate a tighter bound on the migration threshold by computing the distance from the mean in which transient overload is unlikely to occur. For example, as a heuristic, we assume the distribution of transient overloads is normal, then compute $\mu + 3\sigma$ to be a new threshold. If the new threshold is lower than the original threshold, we adopt the tighter bound to reduce the time until migration is triggered for sustained overload.

6.2.2 Capacity Planning

Despite the limitations of the learned probability profiles, they can be sent to the control plane, where they can be aggregated over a large number of VMs over time. This can give insight into the quantity of excess space in the cloud needed to handle overload and how to partition it between space for migration and space for cooperative swap. The control plane, in return, informs each excess space manager how to subdivide its resources.

To demonstrate how probability profiles inform the subdivision of excess space, consider a VM running the webserver component of the SPECweb2009 application described in Section 2.1.3, when allocated only 640 MB of memory. According to its probability profile (Figure 2.3) this application VM has a 16% chance of experiencing overload where 96% of the overloads are less than 1 minute in duration. In order for migration to handle sustained overload, we assume that there must be enough excess capacity to have room to migrate a full VM. Similarly, in order for cooperative swap to handle transient overload, we conservatively assume that the sum of the overloaded VM's current memory allocation and the excess space that is required for cooperative swap is equal to the non-oversubscribed allocation, regardless of how short the burst is.

Assuming the overload characteristics of all VMs are independent[5], if p is the probability of the most likely VM to have overload, we can compute a bound on the probability that at most k VMs will experience simultaneous overload:

$$P\{\#\text{ overloaded VMs} \leq k\} = \sum_{i=0}^{k} \binom{n}{i} \cdot p^{i} \cdot (1-p)^{n-i},$$

where n is the number of VMs in the cloud. For example, consider an oversubscribed cloud supporting 150 VMs, all running the SPECweb2009 configuration described above. Even if each VM is allocated 640 MB, rather than the 1024 MB they would have requested, we would expect—with probability 0.97—that no more than 3 VMs experience simultaneous sustained overload and no more than 31 VMs experience simultaneous transient overload. This, along with our assumptions about how much excess space is required to satisfy a single overload

[5]VMs that make up a single application can experience correlated overload; however the enterprise data in Section 2.1.3 indicates that overload is not highly correlated across all VMs in a well provisioned data center.

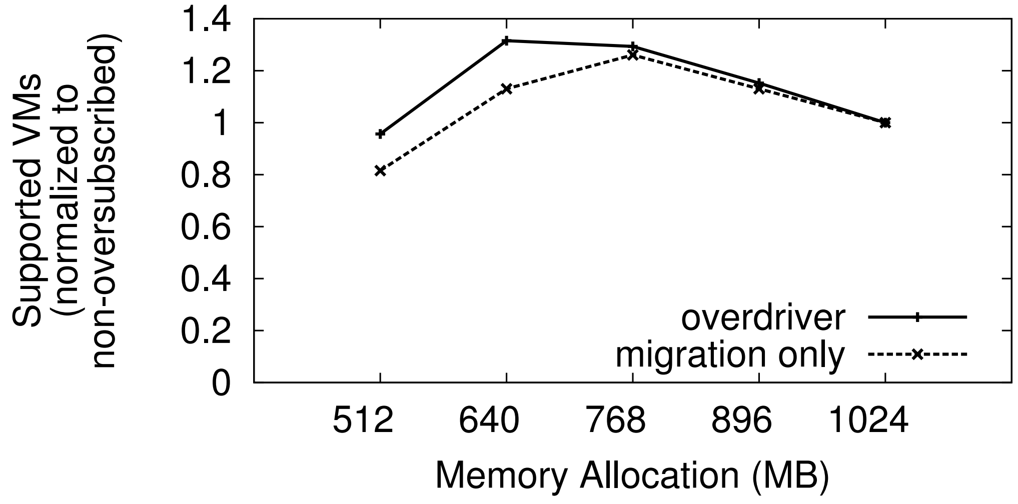


Figure 6.3: Overdriver allows more VMs to be supported by balancing the amount of excess space needed for sustained and transient overloads.

of either type, allows us to compute—with high probability—the amount of each type of excess space needed to handle all overloads in the entire cloud.

Recognizing the overload continuum exists, and selecting the correct amount of each type of excess space can allow an increase in the number of VMs supported in the cloud. Assuming that migration and cooperative swap can handle overload longer and shorter than one minute respectively while preserving application performance, we numerically compute the total number of VMs that can be run in the cloud under different oversubscription levels. We fix the total amount of memory in the cloud as a constant just under 100 GB. For each memory allocation, we input the likelihood of overload and the likelihood that the overload is transient or sustained from Figure 2.3. The analysis iteratively increases the number of VMs and calculates the amount of excess needed to handle the expected number of simultaneous overloads, in order to find the maximum number of VMs that can be safely supported.

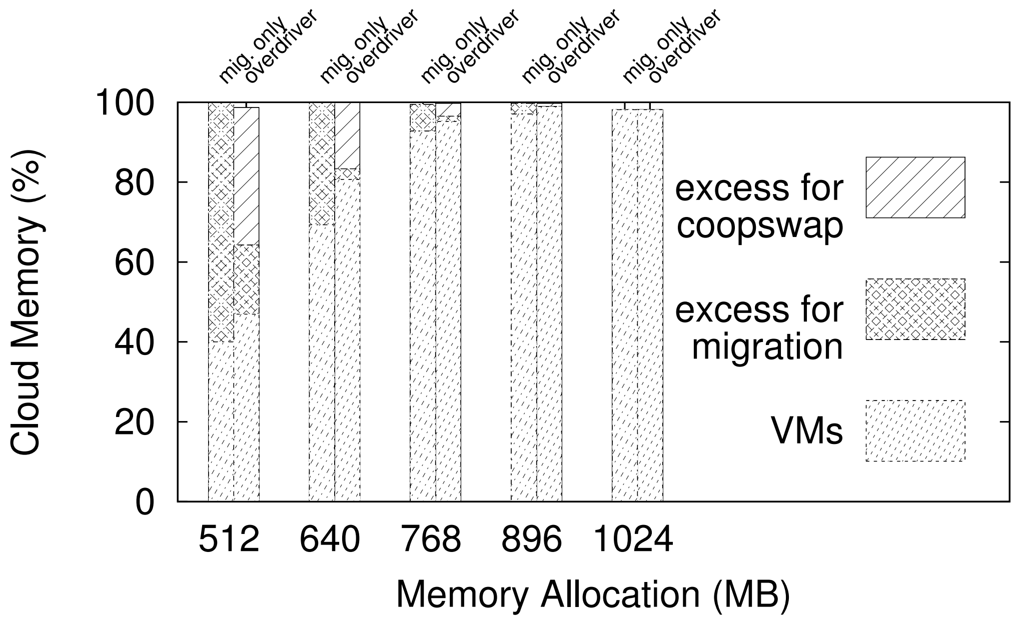


Figure 6.4: The breakdown of cloud resources in terms of space for VMs, excess space for migrations, and excess space for cooperative swap.

Figure 6.3 shows the number of VMs that can be supported with high probability using Overdriver or a migration-only strategy. As memory becomes more oversubscribed, more VMs can be run, but more excess space must be maintained to endure overload. Since cooperative swap consumes space proportional to the overload duration, it allows more room for VMs than an approach that uses migration to address all types of overload, including transient overload. To make this point clear, Figure 6.4 shows the breakdown of memory in the cloud, with Overdriver saving 12% more of the cloud resources for VMs in the 640 MB case. If the physical machines (or Xen-Blanket instances) are oversubscribed too much, corresponding to VM allocations of only 512 MB in this case, the excess space required to handle sustained overload through migration begins to dominate, reducing the available memory to support additional VMs.

6.2.3 Discussion

The maintenance of excess space is a key issue when designing a system to handle overload in the cloud. Where the various types of excess space are placed throughout the cloud, when and how excess space is reclaimed after an overload, and how to combat fragmentation within excess space, are all important questions to consider.

Each mitigation strategy imposes different constraints on how its portion of excess space can be placed throughout the cloud. Excess space saved for cooperative swap in the form of page repositories has very few constraints on placement: there is no requirement that swap pages are stored on the same physical machine (or Xen-Blanket instance); nor do the page repositories need many other resources, such as CPU. On the other hand, excess space saved for migration has more constraints: there must be enough resources to support a full VM co-located on a single physical machine (or Xen-Blanket instance), including memory and CPU. Another mitigation strategy that we have not discussed is memory ballooning [145], or modifying the memory allocation on the local machine. Excess space saved for ballooning has very limiting constraints, namely that it must reside on the same physical machine (or Xen-Blanket instance) as the VM that is experiencing overload. Resource allocation and placement algorithms must adhere to each of these various constraints.

Overload occurs in a continuum of durations, but when it finally subsides, the excess space that was being used for the overload must be reclaimed. The reclaiming process can be proactive, in which excess space is pre-allocated before overload occurs, or reactive, in which resources are dynamically carved out of the cloud on an as-needed basis. Regardless, policies for reclaiming resources are tightly integrated into VM placement and resource allocation, located in the control plane.

Reclaiming memory resources for future migration operations may require squeezing the memory allocations of VMs which may or may not have experienced an overload, and care must be taken to ensure that the reclaiming process does not trigger more overloads in a cascading effect. Reclaiming space in a cooperative swap page repository, on the other hand, can be straightforward—if the VM reads a swap page after overload subsides, that swap page can most likely be deleted from network memory. Otherwise, if the swap page has not been read, the swap page will remain outside of the VM. However, the swap page can be copied to local disk and removed from network memory at any time,[6] although the performance of a future read of the page will suffer.

Resource reclaiming may also need to consider the fragmentation that occurs within excess space. For example, after a migration completes, some resources are allocated to the overloaded VM. The amount of resources awarded need not be the original requested amount nor must they include all of the available resources on the physical machine (or Xen-Blanket instance). More likely, there will be some resources remaining that are insufficient to host a new VM, but not needed by any of the running VMs. Filling the unused resources with cooperative swap page repositories is one option to combat fragmentation, but ultimately, some consolidation process involving migration will be necessary, once again tightly integrated with VM placement and resource allocation.

[6]Depending on the fault tolerance policy, a swap page may already be on local disk (see Section 6.2.4).

6.2.4 Implementation

We have implemented Overdriver to run on Xen and the Xen-Blanket. We leverage Xen's built-in support for live migration and implement cooperative swap clients and page repositories from scratch. An Overdriver agent, written in Python and running in Domain 0, locally monitors the paging rate of the VMs on its physical machine (or Xen-Blanket instance) to detect and react to overload. Probability profiles are maintained in a straightforward manner, which updates the migration threshold. Page repositories implementing excess space for cooperative swap exist as C programs, executed in Xen's Domain 0, that pin memory to ensure that pages written and read do not encounter additional delays. While much of the implementation is straightforward, given space constraints, we focus on some interesting implementation considerations for the overload controller and the cooperative swap subsystem.

Overload Controller

As described in Section 6.2.1, the overload controller within the agent initiates a migration operation after overload has been sustained past an adaptive threshold. However, the implementation must be able to identify overload from normal behavior. Based on our observations of VMs that are using the paging system, a very low paging rate tends to be innocuous, done occasionally by the operating system even if there is no visible performance degradation. While any paging may be a good indication of future overload, Overdriver uses a paging rate threshold to determine whether overload is occurring.

Furthermore, the implementation must differentiate between a series of transient overloads and a sustained overload which has some oscillatory behavior. From our experiments inducing sustained overload, we observe oscillations that can last for almost a minute. In order to correctly classify periods of oscillation as sustained overload, Overdriver includes a sliding window. A configurable number of time intervals within the window must have experienced overload for Overdriver to identify sustained overload.

Cooperative Swap

A key factor in the performance of cooperative swap is where the client is implemented. In order to remain guest-agnostic, we only consider implementations within the VMM or the control domain (Domain 0) of the hypervisor. We have experimented with two different architectures. In the first, we leverage the Xen block tap drivers [149] (`blktap`) as an easy way to implement cooperative swap clients in user-space of Domain 0. When a VM begins to swap, Xen forwards the page requests into userspace, where the block tap device can either read or write the pages to the network or the disk. We noticed that swapping to disk, using a `blktap` driver in Domain 0 for the disk, was significantly outperforming cooperative swap. The reason for this unexpected result was that pages being written by the user-space disk driver were being passed into the kernel of Domain 0, where they would enter the Linux buffer cache, and wait to be written asynchronously. Asynchrony is especially well suited to cooperative swap, because, unlike writes to file systems or databases, the pages written out have no value in the case of a client failure. Furthermore, the buffer cache may be able to service some reads. In order to take advantage of the buffer cache, in addition to reducing context switch overhead, we decided to implement cooperative swap in the kernel underneath the buffer cache, similar to a network block device (`nbd`).

Regardless of the location of the cooperative swap client, the implementation must provide some additional functionality to reading and writing pages to and from page repositories across the network. We highlight two interesting aspects of the implementation. First, cooperative swap clients must be able to locate pages, which may or may not be on the same page repository, even after a VM has migrated. In-memory state consists mainly of a per-VM hash table, indexed by the sector number of the virtual swap device. Each entry includes the address of the page repository the page is stored at, a capability to access the page, and the location of the page on disk. Both the data structure and the swap pages on disk must be migrated with a VM. This is currently done during the stop-and-copy portion of live migration, although a pre-copy or post-copy live migration technique could be implemented for both the data structure and disk pages. Second, it is important to ensure that failure of a remote machine in the cloud does not cause failure of a VM that may have stored a swap page there. Fortunately, there are several accepted mechanisms for reliability in a system that uses network memory. By far the simplest is to treat the page repositories across the network like a write-through cache [62, 71]. Since every page is available on the disk as well as to the network, the dependency on other machines is eliminated, and garbage collection policies are simplified. Reads enjoy the speed of network memory, while writes can be done efficiently through asynchrony. Alternative approaches exist, such as using full replication of swap pages or a RAID-like mirroring or parity scheme [106,117], but they add considerable complexity to failure recovery as well as garbage collection.

6.3 Evaluation

We have argued that in order to enable high resource utilization using aggressive memory oversubscription, it is necessary to *react* to overload, both transient or sustained. In this section, we quantify the tradeoffs described in Section 6.1 in terms of the impact of each strategy on application performance, and in terms of overhead to the cloud, most notably network and excess space overhead. We also show that Overdriver successfully navigates this tradeoff, maintaining application throughput to within 8% of a non-oversubscribed system, while using at most 150 MB of excess space for transient overloads.

At a high level, we want to answer the following questions:

- Does Overdriver maintain application performance despite memory overloads?
- Does Overdriver generate low overhead for the cloud despite memory overloads?

The answers to these questions clearly depend on the application, its traffic, and the level of oversubscription employed. Ideally, we would like to deploy Overdriver in a production cloud and experiment with varying oversubscription levels to answer these questions. Unfortunately, we do not have access to a production cloud, so we use three servers, each with a Xeon 5120 1.86 GHz dual core processor, 32 GB of memory, and a 133 GB SCSI disk with 7200 rpm. For the workload, we once again experiment with SPECweb2009. Instead of simulating a realistic client workload as described in Section 2.1.3, we run a steady base load of clients and inject bursts of various duration in order to evaluate the performance of Overdriver in the face of different types of memory overload.

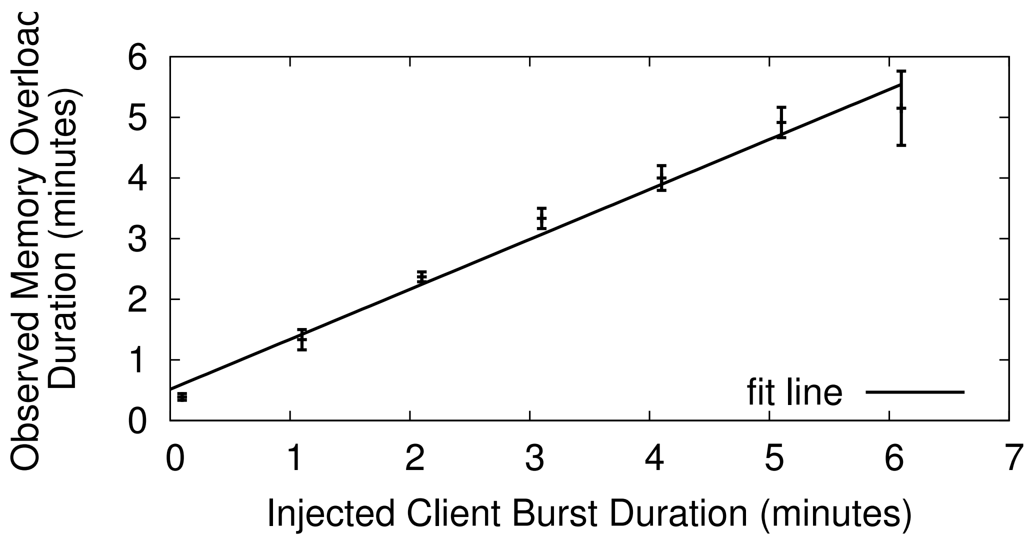
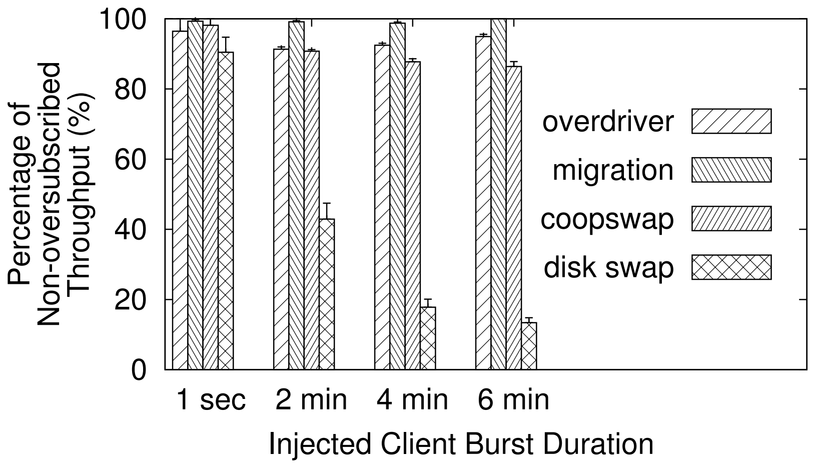


Figure 6.5: Observed memory overload duration roughly matches the duration of the injected client burst.

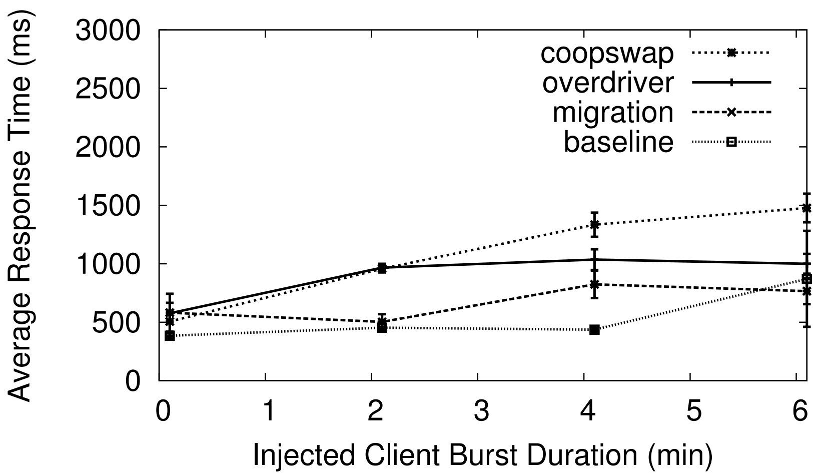
To be more precise, each SPECweb2009 experiment consists of a steady client load of 250 simultaneous sessions being imposed upon a web server VM. The VM, which initially requested 1024 MB, has only been allocated 512 MB of memory because of oversubscription, achieved by inflating the VM's memory balloon. The experiment lasts for 10 minutes. After waiting for 1 minute at the steady state, a client burst is injected, increasing the number of simultaneous sessions by 350 for a total of 600 sessions during bursts. The burst is sustained for a configurable amount of time before the simultaneous sessions return to 250 for the remainder of the experiment. A longer client burst roughly corresponds to a longer memory overload, as shown in Figure 6.5. We perform experiments on a native cloud (e.g. not on superclouds) to best isolate performance during overload. The physical machines used in the experiments have 4 GB of memory each, while the memory allocation of Domain 0 is set at 700 MB.

Through the experiments we compare how Overdriver handles overload (identified as `overdriver` in each graph) to several other techniques. First, as a best-case, we measure the performance of the application, had its VM been well-provisioned. Then, the option of simply swapping pages out to disk (called `disk swap`) is provided as a worst case for application performance, and a baseline for cloud overhead. In between these two extremes we run a solely co-operative swap approach (`coopswap`) and a solely migration-based approach (`migration`). The migration approach has a number of factors, discussed earlier, that make it difficult to compare against. Resource allocation and placement are central to migration strategies, as is some sort of migration trigger. To avoid these issues, we compare against an idealized migration scenario involving a different VM co-located on the same physical machine as the overloaded VM. On the first sign of overload, this other VM, which has a memory allocation of 1024 MB, is migrated to another physical machine, releasing its memory for use by the overloaded VM. In reality, VM allocations can be much larger than 1024 MB, resulting in longer delays before migration can complete and more impact to the network. So, in some sense, the migration strategy we compare against is a best-case scenario in terms of application impact and cloud overhead.

Throughout the experiments, we configure Overdriver to monitor the VM every 10 seconds, and consider time intervals where the paging rate is above 200 operations per second as periods of overload. The threshold used to trigger migration is initially set at 120 seconds, with a sliding window parameter requiring 8 out of the 12 measurements to be overloads (i.e. 120 seconds comes from twelve 10 second monitoring periods and requires at least 80 seconds out of a window of 120 seconds to trigger a migration).



(a) Throughput



(b) Average response time

Figure 6.6: The effects of each mitigation technique on the SPECweb2009 application. Overdriver maintains application throughput within 8% of that of a non-oversubscribed VM and an average response time under 1 second for any length overload.

6.3.1 Application Effects

Figure 6.6 shows the performance degradation experienced by the application under various overload mitigation strategies in terms of lost throughput and increase in average response time. First, we examine the aggregate throughput over the 10 minute experiment. The experiment was run on a well-provisioned VM to get a baseline for how many connections should have been handled if no overload occurred, which was between 23 and 38 thousand connections, depending on the length of the injected client burst. Figure 6.6(a) shows the percentage of that ideal throughput for each strategy. The first thing to notice is that the throughput while using disk swap drops off dramatically, whereas degradation is much more graceful using cooperative swap. Migration, on the other hand, completes with nearly full performance, except for very small spikes resulting in reductions in throughput performance for short periods of time, until migration completes and benefits of migration can be seen. A less aggressive solely migration-based strategy would degrade with disk swap until migration was triggered. Overdriver, on the other hand, begins to degrade gracefully along with cooperative swap, but then improves for longer overload as it switches to rely on migration. A similar pattern can be seen with application response time, shown in Figure 6.6(b). While longer overload periods cause cooperative swap to increase the average response time, Overdriver levels off when migration is used. Disk swap is not shown on Figure 6.6(b) because it performs significantly worse than other strategies; the average response time varies between 2.5 s and 16 s, depending on the overload duration. Overall, Overdriver achieves a throughput within 8% of a well-provisioned, non-oversubscribed VM, and an average response time under 1 second.

In terms of application throughput, we see that Overdriver degrades gracefully to a point, at which overload is sustained and warrants migration. The cost of Overdriver to the application, while fairly low, is higher than a very aggressive migration strategy, because Overdriver pays for the time spent deciding

Threshold(s)	Tput(%)	Response Time (ms)
100	94.97	946
120	92.47	1249
150	93.76	1291
170	84.52	1189
200	84.85	1344

Table 6.2: As the threshold for migration increases, more performance loss is experienced. This table shows the lowest percentage of throughput achieved, and highest average response time, out of all durations of overload.

whether the spike will be sustained. Cooperative swap drastically improves performance while making this decision. As discussed earlier, especially given the prevalence of transient overload, the number of migration operations required is also drastically reduced, affecting both the amount of excess space required, and ultimately the number of VMs that can be supported in the cloud.

The choice of threshold has an effect on the application performance, because a higher threshold translates into an increased reliance on cooperative swap. Table 6.2 shows application performance experienced by Overdriver as the migration threshold due to sustained overload varies. As the threshold increases above 120 seconds performance degrades. This performance degradation gives an idea of the performance that can be gained by adaptively learning a tighter threshold.

6.3.2 Cloud Effects

The most immediately quantifiable impact to the cloud of the overload mitigation techniques we have described is in terms of the amount of traffic induced on the network, and in terms of the amount of excess space required for migration

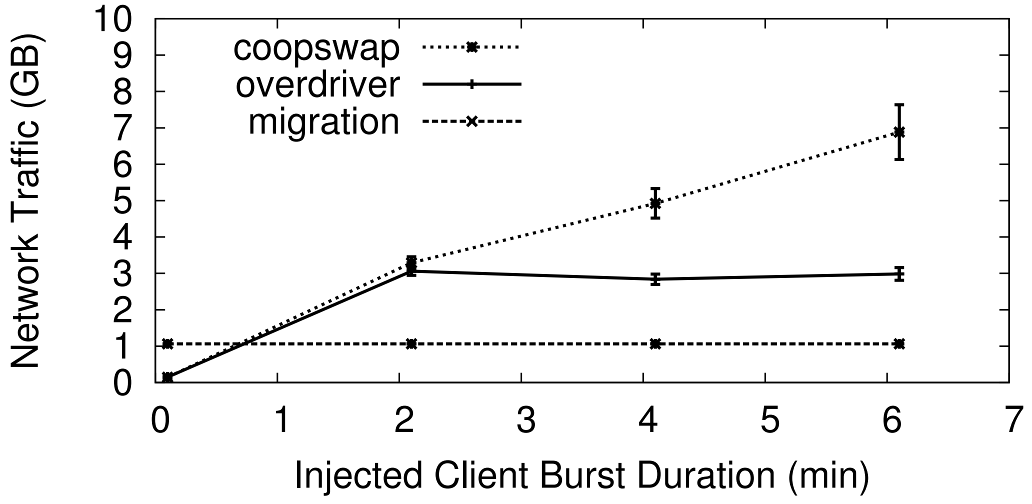


Figure 6.7: Network traffic induced by various length load spikes using cooperative swap versus disk swap.

or cooperative swap. We show that Overdriver can handle transient overloads with a fraction of the cost of a purely migration-based solution. However, we also show that Overdriver incurs additional costs for sustained overloads.

Figure 6.7 shows the amount of traffic sent and received during the experiments described above (Section 6.3.1) for various length client bursts. The number of pages written to the cooperative swap page repositories increases fairly linearly as the length of the overload increases. However, cooperative swap also reads from the page repositories, which results in a further increase in network traffic. Migration, on the other hand, exerts a fixed constant, which is almost entirely written, regardless of the duration of overload. The fact that the value on the graph is fixed at 1 GB is because that is the memory allocation of the VM that is being migrated in this experiment. If a larger, 2 GB VM was migrated, we would see the migration line at 2 GB instead. Overdriver follows the cooperative swap line until the duration of overload exceeds its threshold, and it resorts

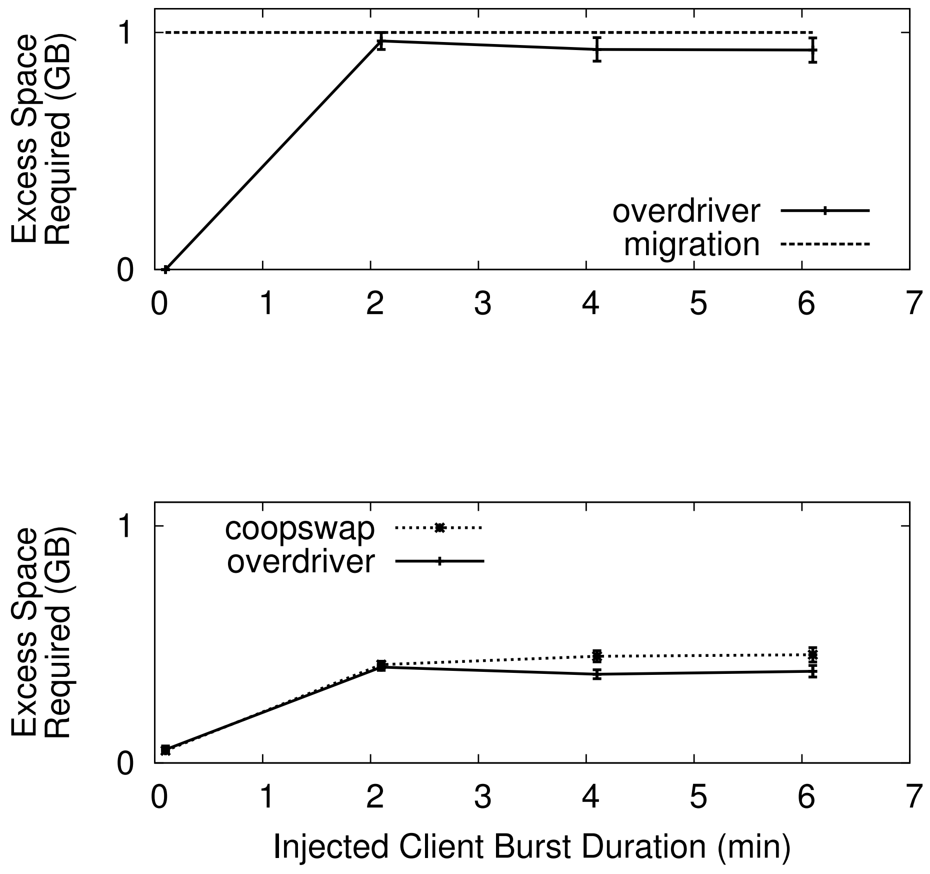


Figure 6.8: Amount of excess space required for migration and cooperative swap using Overdriver.

to migration. This causes a sharp increase by the memory allocation of the VM (1 GB in this case) in the amount of data written to the network by Overdriver for sustained overload. In other words, to get the benefit for transient overloads, Overdriver pays a cost for sustained overloads that is proportional to the amount of time it used cooperative swap.

The amount of traffic over the network does not necessarily show the amount of excess space that is required for cooperative swap. For example,

Threshold(s)	Traffic(GB)	Space(MB)
100	2.1	424
120	3.6	549
150	4.4	554
170	5.6	568
200	6.1	590

Table 6.3: For sustained overloads, as the threshold for migration increases, more overhead is accumulated while waiting for migration to happen. This table shows the maximum overhead for cooperative swap in terms of traffic generated and excess space required for a sustained overload.

some pages may be written, read, then overwritten, without consuming more space at the page repository. Figure 6.8 quantifies the amount of excess space required for migration and cooperative swap. Even though cooperative swap may generate a virtually unbounded amount of network traffic for sustained overload, the amount of space that is required remains reasonable. In particular, the amount of space required does not increase above the amount of memory that was withheld because of oversubscription. For transient overloads, Overdriver must only use the modest amount of excess space for cooperative swap. For sustained overloads, however, Overdriver uses both.

Similarly to application performance, cloud impact is also affected by the threshold that Overdriver uses. Table 6.3 quantifies the increases in overhead for a sustained overload that can be reduced by adaptively shrinking the threshold.

Finally, while Figures 6.7 and 6.8 quantify a tradeoff, and show how Overdriver navigates the tradeoff for a single overload, they only provide a glimpse into the tradeoffs that would appear if done at a cloud scale. In particular, transient overloads are dominant, and so the modest savings that Overdriver achieves in this graph must be multiplied by the number of transient over-

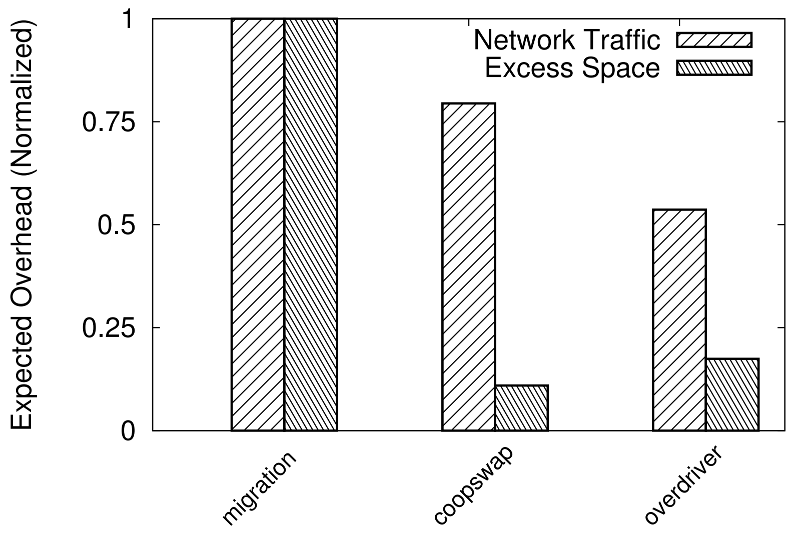


Figure 6.9: The expected overhead given the dominance of transient overloads

loads, which should far outweigh the overhead incurred for the relatively rare sustained overloads. Figure 6.9 shows that, using the probability profile for a SPECweb2009 VM allocated 640 MB discussed in Section 2.1.3, Overdriver is expected to save 46% network bandwidth and requires 83% less excess space than migration only. Furthermore, as discussed in Section 6.2, the differences in terms of how excess space is being used has far-reaching implications in terms of additional costs related to maintaining excess space.

6.4 Summary

Oversubscription is one avenue towards maximizing the utilization of resources in superclouds. As interest grows in oversubscribing resources in the cloud, effective overload management is needed to ensure that application performance remains comparable to performance on a non-oversubscribed cloud. Through analysis of traces from an enterprise data center and using controlled experiments with a realistic Web server workload, we confirmed that overload appears as a spectrum containing both transient and sustained overloads. More importantly, the vast majority of overload is transient, lasting for 2 minutes or less.

The existence of the overload continuum creates tradeoffs, in which various mitigation techniques are better suited to some overloads than others. The most popular approach, VM migration-based strategies, are well suited to sustained overload, because of the eventual performance that can be achieved. Cooperative swap applies to transient overloads, where it can maintain reasonable performance at a low cost.

We have presented Overdriver, a system that handles all durations of memory overload. Overdriver adapts its mitigation strategy to balance the tradeoffs between migration and cooperative swap. Overdriver also maintains application VM workload profiles which it uses to adjust its migration threshold and to estimate how much excess space is required in the cloud to safely manage overload. We show, through experimentation, that Overdriver has a reasonably small impact on application performance, completing within 8% of the connections that a well-provisioned VM can complete under the continuum of overload, while requiring (under reasonable oversubscription ratios) only 15% of the excess space that a migration-based solution would need.

Cloud extensibility enables each cloud user to decide which oversubscription techniques are right for their enterprise workloads and provides the control to implement them. For example, Overdriver and the functionality it relies on (e.g. VM migration and cooperative swap) can run on Amazon EC2 by leveraging the Xen-Blanket. Overdriver shows that safe oversubscription is possible, opening the door for new systems that effectively manage excess space in the cloud for the purpose of handling overload, ultimately leading towards a new class of highly efficient, oversubscribed cloud deployments. The Overdriver project webpage is available at: `http://overdriver.cs.cornell.edu`.

CHAPTER 7

RELATED WORK

In this dissertation, we explored a new user-centric cloud computing model called superclouds. We investigated an abstraction—cloud extensibility—that enables cloud users to obtain unprecedented level of control over third-party cloud resources. Cloud extensibility is influenced by related work on extensible operating systems and nested virtualization. Leveraging cloud extensibility, we researched cloud interoperability, or how a cloud user can create a uniform environment across cloud providers. While there are numerous other efforts towards cloud interoperability, existing approaches do not enable the level of user control of the Xen-Blanket. We researched how a cloud user can implement the control logic of a cloud network. Related work that investigates virtual networking abstractions does not offer the level of control of VirtualWire. Finally, we researched how a cloud user can efficiently utilize cloud resources through memory oversubscription. Related work on handling memory overload typically utilizes either live VM migration or network memory but does not address transient and sustained overloads differently. In this chapter, we discuss work related to each of our contributions, the techniques they build on, and alternate or complementary approaches.

7.1 Cloud Extensibility

We have described cloud extensibility as an abstraction that enables superclouds. Extensible clouds are influenced by prior work on extensible operating systems. For example, SPIN [45] allows extensions to be downloaded into the kernel safely using language features, while Exokernels [67] advocate hardware to be exposed to a library OS controlled by the user.

To avoid a provider-centric approach, like standardization, we focus on nested virtualization as a deployable, user-centric approach to cloud extensibility. Graf and Roedel [74] and the Turtles Project [43] are pioneers of enabling nested virtualization with one or more levels of full virtualization, on AMD and Intel hardware, respectively. Berghmans [44] describes the performance of several nested virtualization environments. CloudVisor [161] explores nested virtualization in a cloud context, but for security, where the provider controls both layers.

7.2 Cloud Interoperability

We have explored how to enable a cloud user to homogenize, or provide a uniform environment across, third-party cloud providers. Our approach is embodied in the Xen-Blanket. Other techniques exist to deploy applications across multiple clouds, but none afford the user the flexibility or level of control of user-centric approach embodied in the Xen-Blanket. There are also a number of existing systems which exhibit the type of control needed for enterprise workloads but cannot be deployed on third-party clouds. These systems may be able to be implemented in a supercloud.

Multi-Cloud Deployments

Using tools from Rightscale [58], a user can create ServerTemplates, which can be deployed on a variety of clouds and utilize unique features of clouds without sacrificing portability. However, users are unable to homogenize the underlying clouds, particularly hypervisor-level services.

Middleware, such as IBM's Altocumulus [109] system homogenizes both IaaS clouds like Amazon EC2 and Platform as a Service (PaaS) clouds like Google App Engine into a PaaS abstraction across multiple clouds. However, without control at the IaaS (hypervisor) level, the amount of customization possible by the cloud user is fundamentally limited.

The *fos* [151] operating system, deployed on EC2 and potentially deployable across a wide variety of heterogeneous clouds, exposes a single system image instead of a VM interface. However, users must learn to program their applications for fos; the familiar VM interface and legacy applications contained within must be abandoned.

Eucalyptus [68] and AppScale [54] are open-source cloud computing systems that can enable private infrastructures to share an API with Amazon EC2 and Google App Engine respectively. However, the user cannot implement their own multi-cloud hypervisor-level feature. OpenStack [30] is another open-source implementation of an IaaS cloud, with the same limitation.

The RESERVOIR project [131] is a multi-cloud agenda in which two or more independent cloud providers create a *federated cloud*. A provider-centric approach is assumed; standardization is necessary before federation can extend beyond the testbed. With the Xen-Blanket, such an agenda could be applied across existing public clouds.

Controlling Clouds

OpenCirrus [51] is an initiative that aims to enable cloud targeted system level research—deploying a user-centric cloud—by allowing access to bare hardware, as in Emulab [153], in a number of dedicated data centers. However, OpenCirrus is not aimed at applying this ability to existing cloud infrastructures.

Cloud operating systems such as VMWare's vSphere [142] allow the administrator of a private cloud to utilize a pool of physical resources, while providing features like automatic resource allocation, automated failover, or zero-downtime maintenance. These features, which are examples of hypervisor-level services, cannot easily be integrated with current public cloud offerings.

It may be possible to implement these systems, along with Eucalyptus [68] and OpenStack [30], in a supercloud. This challenge are a subject of future research, as discussed in Chapter 8.

7.3 User Control of Cloud Networks

We have explored how to enable a cloud user to manage the network control logic in a cloud. Our approach is embodied in VirtualWire. New data center network architectures are emerging that increase the flexibility of cloud networks. Network virtualization promises an important step towards superclouds by decoupling the network from physical infrastructure. Like VirtualWire, a number of network virtualization approaches exist, however, VirtualWire is unique in that it places the control logic of the virtual network in the domain of the user, and that it has been applied in superclouds.

Enterprise Cloud Migration

Similar to VirtualWire, Virtual Private Networks (VPNs) are being applied to enable enterprise deployments to migrate or extend into third party clouds. For example, Amazon Virtual Private Cloud (VPC) [1] allows users to pick their own IP addresses within a subnet connected to a private network via a VPN, but does not support layer 2 protocols. vCider [22] and VPN-Cubed [24] support layer 2 protocols in the cloud and even provide some control over the network topology, but require configuration in the guest operating systems. CloudSwitch [3] operates in an isolation layer that avoids guest operating system configuration, but does not facilitate the implementation of flow policies in the cloud. VirtualWire allows a user to specify a virtual network topology that can be maintained throughout migration, regardless of the physical location of components. VirtualWire can benefit from applying the VM migration optimizations across VPNs described in CloudNet [157].

Data Center Network Architecture

VirtualWire allows users to implement the control logic for their virtual networks. At the provider level, a number of network architectures have been proposed to maximize flexibility and efficiency [35, 40, 76, 79, 80, 95, 119]. Two network virtualization systems that achieve scale and performance are VL2 [75] and NetLord [118]. OpenFlow [110] and, more broadly, software defined networks (SDN) [78, 98, 126] can produce efficient virtual networks [14]. A user in VirtualWire can develop and test new applications of SDN, like Mininet [102], or incrementally upgrade the control logic in their virtual network to benefit from them. VirtualWire is the only system that enables users of third-party clouds to implement the network control logic.

Software Network Components

VirtualWire relies on the availability of software network component implementations. Open vSwitch [126] is a production quality, fully featured OpenFlow-enabled software switch; Cisco's 1000V [56] and NetSim [18] share interfaces with physical Cisco network devices. Software routers are known for extensibility [82, 97], and, with recent developments in server hardware, can achieve very high performance [64]. VROOM [148] describes how to migrate software routers for network management. Unlike VirtualWire, virtual routers in VROOM are not running on general purpose servers or third-party clouds, but on high-end routers with virtualization support.

Standards

VirtualWire performs encapsulation and adopts the Virtual eXtensible Local Area Network (VXLAN) [105] wire format. VXLAN [105] and Network Virtualization using Generic Routing Encapsulation (NVGRE) [137], are emerging to use encapsulation to extend virtual segments even over layer 3 networks. Other standards, such as Locator/ID Separation Protocol (LISP) [129], have been proposed to separate device identification and network routing.

7.4 Efficient Cloud Resource Utilization

We have explored how to enable efficient cloud resource utilization through memory oversubscription and handling memory overload. Our approach is embodied in Overdriver. Oversubscription is common, as are VM migration and network memory, the two techniques used by Overdriver to handle memory overload.

Memory Oversubscription

The ability to oversubscribe memory is common in modern virtual machine monitors. VMWare ESX server [145] was the first to describe memory balloon drivers which provided a mechanism to remove pages from a guest OS by installing a guest-specific driver. Xen [41] also contains a balloon driver. Balloon sizing can be done automatically with an idle memory tax [145] or as a more sophisticated driver inside the guest OS [85]. While ballooning allows changes to the allocation of machine pages, memory usage can also be reduced through page sharing techniques: mapping a single page in a copy-on-write fashion to multiple VMs. This technique was first described in VMWare ESX server [145] and has subsequently been extended to similarity detection and compression in Difference Engine [81]. Memory Buddies [159] is a migration-based strategy that tries to place VMs to optimize for page sharing. Satori [116] is examining new mechanisms to detect sharing opportunities from within the guest OS. These systems are great mechanisms for oversubscribing memory; however, a robust system to reactively handle and mitigate overload, like Overdriver, is necessary to maintain performance.

Migration

VM migration has become very popular and is touted as a solution free of residual dependency problems that have plagued process migration systems for years [115]. Stop-and-copy VM migration appeared in Internet suspend/resume [99]. Compression has been noted as a technique to improve the performance of migration, especially if the image contains lots of zeroes [134]. However, these techniques all impose significant downtimes for VMs. Live migration techniques, on the other hand, allow VMs to be migrated with minimal downtime. Push-based live migration techniques are the most popular; implementations include VMWare's VMotion [122], Xen [57], and KVM [96]. Pull-based live migration [85] has been evaluated and a similar mechanism underlies the fast cloning technique in SnowFlock [101].

We do not discuss addressing memory overload by spawning VMs. If a hosted service is structured to be trivially parallelizable, such that a newly initialized VM can handle new requests and share the load, spawning may be another viable mechanism to alleviate memory overload. Work has been done to maintain a synchronized hot spare [61] and to speed up cloning delays [101,128].

There are many migration-based approaches that try to achieve various placement objectives, but they do not discuss maintaining excess space to handle overload. Khanna et al. [94], use heuristics to try to consolidate VMs on the fewest physical machines, while other approaches Entropy [84] and Van et al. [141] aim for an optimal consolidation with the fewest migrations. Sandpiper [158] uses migration to alleviate hotspots in consolidated data centers. In an effort to eliminate needless migrations, Andreolini et al. [37] use a trend analysis instead of triggering migration with a utilization threshold. Stage and

Setzer [138] advocate long-term migration plans involving migrations of varying priority in order to avoid network link saturation.

Network Memory

Overdriver uses cooperative swap to address the paging bottleneck of overloaded VMs. Accessing remote memory on machines across a fast network has been recognized to perform better than disk for some time [36], and this concept has been applied to almost every level of a system. `memcached` [27] leverages network memory at the application level. Cooperative caching [62] gains an extra cache level in a file system from remote client memory. The Global Memory System [71] uses a remote memory cache deep in the virtual memory subsystem of the OS, naturally incorporating all memory usage including file systems and paging. Nswap [123] and the reliable remote paging system [106] focus specifically on sending swap pages across the network. Cellular Disco [73] is a hypervisor that uses network memory to borrow memory between fault-containment units called cells. Most similar to cooperative swap, MemX [86] implements swapping to the network for a guest VM from within the hypervisor. MemX is focused on extremely large working sets that do not fit in a physical machine's memory, whereas cooperative swap is designed to react quickly to overload bursts, many of which are transient. Other techniques to increase the performance of paging include the use of SSDs. However, addressing the paging bottleneck is not enough to handle the entire overload continuum, particularly sustained overloads.

CHAPTER 8

FUTURE WORK

Superclouds embody a new environment in which enterprise workloads can be deployed on one or more clouds. In this dissertation, we have investigated three aspects of a supercloud. First, we researched how to enable a cloud user to homogenize, or provide a uniform environment, across clouds, as a step towards cloud interoperability. Second, we researched moving the control logic of the network to be managed by the cloud user. Finally, we researched efficient cloud resource utilization through memory oversubscription.

We also designed, implemented and evaluated an instance of each of the three approaches, resulting in three systems: the Xen-Blanket, VirtualWire, and Overdriver. Together these systems form an instance of a supercloud. In the future, we plan to adopt entire cloud stacks, such as Eucalyptus [68] or OpenStack [30], to enhance our instance of a supercloud.

In addition, there are many interesting research directions that build on superclouds. In this chapter, we focus on three key directions. First, superclouds have an opportunity to more completely abstract away physical infrastructure such that workloads on superclouds may not be aware of what underlying cloud provider they are running on. In other words, superclouds can present a seamless multi-cloud environment. Second, superclouds can be customized to fit an enterprise workload to an extent that existing cloud providers cannot attempt. Finally, there are interesting research challenges in supporting more guests on superclouds and extending superclouds across more heterogeneous cloud providers. In the remainder of this chapter, we briefly describe these future directions.

8.1 Seamless Multi-Cloud

Superclouds have the potential to significantly abstract away the existence of multiple cloud providers. We have begun to show how superclouds can enable interesting functionality that spans clouds, such as cross-provider live migration. New services and functionality offered by superclouds spanning networking, storage, and high availability can further ensure a seamless, high performance cloud environment that spans multiple clouds.

8.1.1 Upgrading the Virtual Network

VirtualWire enables a cloud user to implement a virtual network including the network component configurations that encode the control logic in the network. However, the virtual network may contain network components that are tied to a single cloud provider. In particular, a virtual switch, router, or middlebox that is difficult to re-implement in a distributed architecture may cause traffic between two virtual servers on one cloud to cross to another cloud simply to traverse the middlebox. We are investigating techniques to split an arbitrary middlebox VM between clouds, with each piece processing traffic local to its cloud.

8.1.2 Storage

Our work towards superclouds reduces vendor lock-in by offering a unified, homogeneous cloud environment to cloud users. However, storage remains tied to a single cloud provider. Accessing the highly available, virtually infinite storage offered by one cloud provider from another cloud introduces latency and incurs a financial penalty for data transfer into and out of each cloud. The lock-in risks caused by cloud storage have been addressed in systems such as RACS [32], in which RAID-like techniques stripe data across cloud providers. Superclouds provide an opportunity to co-locate data with VMs that are likely to access the data, while maintaining independence from any one cloud provider.

8.1.3 High Avavailability

While applications can be designed for high availability across clouds (and facilitated by services such as those provided by Rightscale [58]), arbitrary VMs fundamentally lack support for high availability. Even if a cloud provider implemented a high availability VM service, it will be limited to a single cloud provider; the VM cannot withstand the failure of an entire cloud provider. Superclouds span multiple clouds and therefore can implement high availability strategies such as Remus [61] between cloud providers. Challenges include minimizing cross-site traffic, adapting Remus-style protocols for WAN environments, and designing provider-independent gateway points into the cloud deployment. Some challenges have been addresses in PipeCloud [156] and SecondSite [130], but they do not enable cloud users on third-party clouds to utilize them.

8.2 Custom Superclouds

Superclouds fundamentally change the incentive to design highly customized cloud infrastructures. Whereas a cloud provider designs cloud services and interfaces to support the maximum possible range of clients, a cloud user can design superclouds to efficiently support exactly the functionality that best suits one or more workloads. This section describes several directions for research into custom superclouds, by increasing the paravirtualization interface or introducing new abstractions.

8.2.1 Super-Paravirtualization

IaaS cloud platforms typically manage virtual machines as a "black box", in order to support a range of isolated customers. However, treating virtual machines as black boxes results in inefficiencies in almost every aspect of the cloud infrastructure. For example, live VM migration imparts unnecessary overhead to the network because in current black box approaches, the hypervisor cannot differentiate between memory pages containing important system state and those comprising ephemeral caches. Furthermore, resource allocation and oversubscription policies and systems like Overdriver must guess the memory requirements of each VM leading to poor utilization or poor performance.

Hypervisors that use paravirtualization, including the Xen-Blanket, do not treat VMs strictly as black boxes. Instead, guest OSs can directly communicate with the hypervisors. Paravirtualization is typically used to simplify the implementation or improve the performance of tasks that require hypervisor intervention. For example, the guest OS asks the hypervisor for assistance when performing I/O. We propose substantially expanding the interface between the OS and hypervisor in order to both improve resource efficiency and offer new services.

For example, the hypervisor can be extended to manage a single buffer cache used to store disk pages from all VMs on a host, allowing it to remove redundancy if VMs have similar disk state. While a typical OS greedily consumes all available memory for caching, our approach allows the hypervisor to constrain the amount of memory given to each VM, or to grant additional buffer cache space from unused RAM. The result is more efficient use of system memory and added control for the hypervisor. Our approach should also substantially reduce VM migration time since ephemeral memory pages used for caching can be easily skipped by the migration algorithm. In addition, we believe that making operating systems aware that they are being migrated will offer new optimization opportunities.

8.2.2 Memory Cloud

Overprovisioning of memory occurs because of difficulties in predicting the memory usage of VMs. Truly elastic resources, such as storage, do not require prediction: an abstraction of infinite storage is presented, while a VM pays for what it actually uses. We are interested in pursuing the idea of truly elastic memory or a memory cloud, in which elastic memory behaves like elastic storage. With a memory cloud, VMs see a virtually unlimited amount of memory and pay for only what they use. The underlying hypervisor-level software must maintain the illusion of infinite memory without VMs perceiving that some of the memory may not be local. Techniques like those explored in Overdriver will be essential, and emerging architectures with extremely fast inter-node communication, such as Blue Gene [88], may facilitate the implementation of an efficient memory cloud abstraction.

8.3 Blanket Layer Virtualization

In this dissertation, we have described how cloud extensibility can be leveraged for cloud interoperability. In particular, a Blanket layer can homogenize clouds and enable superclouds. This section describes future work that augments the Blanket layer to support a wider selection of guests and encompass more existing clouds.

8.3.1 Unmodified Guests on the Blanket Layer

The Xen-Blanket inherits the limitations of paravirtualization. In particular, guest operating systems must be modified to run on a paravirtualized environment. While standard Linux distributions contain support for paravirtualization, other popular operating systems, such as Microsoft Windows, do not. Without support from the underlying hypervisor (e.g., the Turtles Project [43]), x86 hardware extensions for virtualization cannot be used. Instead of paravirtualization, binary translation (e.g., VMware [140]) can be used for the Blanket layer. As such, unmodified operating systems can run on the Blanket layer. The performance of binary translation in the Blanket layer, however, will likely suffer, especially for device I/O.

8.3.2 Blanket Layers in Paravirtualized Clouds

The Xen-Blanket assumes a fully virtualized (HVM) guest container in which to run the Blanket layer. Even as clouds increasingly offer HVM guests for performance and compatibility, many clouds, including Amazon EC2, continue to offer a wide selection of paravirtualized instances. As future work, we plan to investigate running a version of the Xen-Blanket to run on a paravirtualized interface, while continuing to export a homogeneous interface to guests. Paravirtualizing the Blanket layer is technically feasible, but may encounter performance issues in the memory subsystem where hardware features such as Extended Page Tables (EPT) cannot be used and is another subject of future research.

Further, we believe it is possible to support an unmodified guest using binary translation on a paravirtualized instance. The hypervisor in the Blanket layer is a modified binary translating hypervisor. The hypervisor uses paravirtualized hypercalls to perform common tasks, such as memory management. In the meantime, the instructions used by the unmodified guest as being dynamically translated to communicate with the Blanket layer and the underlying hypervisor. By supporting a Blanket layer across all existing clouds, we increase the applicability of superclouds.

8.4 Summary

Superclouds enable an environment within which novel systems and hypervisor level experimentation can be performed. Superclouds enable new, seamless multi-cloud applications, that could lead to new deployment paradigms for enterprise workloads. Superclouds also enable new, highly customized cloud environments that could lead to better performance and better resource utilization for certain mission-critical applications. As Superclouds span increasing numbers of cloud providers, they have the potential for impact across the entire industry.

CHAPTER 9

CONCLUSION

At the time of writing this dissertation, cloud computing is provider-centric; cloud users do not have adequate control over the dictated format or available services in the cloud. Furthermore, providers are not interoperable. Migrating an enterprise workload to the cloud requires re-engineering effort every time the workload is moved to a new cloud provider.

We have explored a fundamentally user-centric approach to cloud computing. In particular, we have proposed and investigated superclouds: a new, user-centric model for cloud computing. Leveraging a new abstraction called cloud extensibility, the cloud user—not the provider—maintains control over the cloud, from VM format to services. Furthermore, cloud extensibility can be applied to homogenize existing clouds for cloud interoperability. Control over the cloud network can be exerted by cloud users rather than being mandated by cloud providers. Cloud resources can be used efficiently through custom oversubscription policies. Together this dissertation describes important steps towards superclouds.

To validate our user-centric approach, we have described the design, implementation and evaluation of three systems that together are steps towards an instance of a supercloud. The Xen-Blanket is an instance of cloud extensibility, enabling cloud interoperability. VirtualWire is an instance of user control of cloud networks. Overdriver is an instance of handling overload while employing memory oversubscription to a achieve efficient utilization of cloud resources.

A supercloud is not bound to any provider or physical resources and therefore decouples the task of managing physical infrastructure from the service abstraction in cloud computing. This decoupling is an important step toward the further commoditization of computational resources. Additionally, superclouds facilitate research into novel systems and hypervisor level experimentation. Superclouds enable seamless multi-cloud applications and highly customized cloud environments. As the market for customized, cloud-agnostic services grows, superclouds will affect not just cloud users, but the entire cloud ecosystem.

APPENDIX A
BACKGROUND ON VIRTUALIZATION

Virtualization is defined as the separation of a resource or request for a service from the underlying physical delivery of that service [143]. Applied to servers, virtualization is the process implemented by a virtual machine monitor (VMM), or hypervisor, of exporting a virtual hardware interface that reflects the underlying machine architecture [145]. In other words, a hypervisor runs virtual machines (VMs). A VM is defined as an efficient replica of a computer system in software, complete with all the processor instructions and system resources (i.e., memory and I/O devices) [72]. Cloud providers use virtualization to implement an Infrastructure as a Service (IaaS) cloud abstraction. In this appendix, we give a brief primer to virtualization.

While the concept of virtualization can be applied to any hardware instruction set architecture (e.g., x86, ARM, PowerPC), the techniques used to virtualize the hardware differ depending on the architecture. We focus on virtualization techniques for the x86 because it is used by existing cloud providers [29,31]. As described further in Section A.2, the x86 presented some unique challenges to virtualization, requiring the development of software virtualization techniques. Despite the emergence of extensions to the x86 hardware that facilitate virtualization [103, 160], software techniques remain popular. In Section A.3, we discuss two software virtualization techniques for the x86: *binary translation* [140] and *paravirtualization* [41].

A.1 A Historical Perspective

Virtualization was pioneered in the late 1960s by IBM with the Control Program/Cambridge Monitor System CP-40/CMS [60], a precursor to the popular VM/370 [60]. One of the initial uses of virtualization was time-sharing of a single machine for multiple users; using virtualization, each user could run an existing single-user operating system. Virtualization was therefore contemporary to other time-sharing multi-user approaches of the 1960s such as the MULTICS [59] operating system. Virtualization also found use as a mechanism to test and develop systems, for example, an instance of the IBM VM hypervisor could run on the IBM VM hypervisor. The successors of the VM/370 are still used on mainframes in 2012.

As personal computers (PCs) became popular for server workloads, virtualization remained only on mainframes. The reason for its absence on PCs is because PCs used hardware architectures, like the x86, that did not easily lend themselves to virtualization. It was not until over 30 years later that new virtualization techniques were developed in software to overcome the virtualization obstacles inherent in the x86 architecture. Subsequently, virtualization experienced a renaissance, becoming widely popular on the x86 for utility/cloud computing. Eventually, by 2006, 40 years after the first hypervisors were explored by IBM, x86 hardware vendors introduced hardware extensions to support virtualization. The history of virtualization, linked with utility/cloud computing, is summarized in Table A.1.

Virtualization	Date	Utility/Cloud computing
	1965	Several efforts at MIT, including the MAC system [69], aim for computer power to be analogous to the electrical distribution system.
IBM pioneers full system virtualization with the CP-40 [60].	1967	
IBM releases the VM/370 [60].	1972	
	⋮	
The first product to virtualize the x86 is released in VMware's Virtual Platform for x86 [147].	1999	
VMware supports Windows in its GSX x86 virtualization product [23].	2001	HP introduces the Utility Data Center [70].
	2002	Sun announces the N1 [17].
Xen is released to the open-source community, using paravirtualization to virtualize the x86 [41]	2003	
AMD and Intel provide hardware support for virtualizing the x86 [103,160]	2006	Amazon Web Services is launched, in particular, the Elastic Compute Cloud (EC2) [42].
	2008	Eucalyptus [68] provides the first open-source software for building private clouds.

Table A.1: A subset of the important events in the history of virtualization and utility computing. Both were pioneered in the 1960s for mainframes, and both have experienced renewed interest in the 2000s, resulting in current clouds.

A.2 Virtualization Basics and x86 Issues

In 1974, Popek and Goldberg formally defined the requirements of system software to be a hypervisor [127]. A hypervisor must faithfully reproduce the semantics of physical hardware (except for timing), execute a majority of instructions directly on hardware, and the hypervisor must maintain complete control over physical resources [127]. The classical design for a hypervisor is known as *trap-and-emulate*. In a trap-and-emulate system, privileged instructions executed by the VM cause the processor to *trap*, or transfer control to the hypervisor, which *emulates* the instruction on the virtual CPU (VCPU). Until recently, trap-and-emulate was not possible on the x86 architecture.

A.2.1 Trap-and-emulate

Processor architectures contain privileged and unprivileged instructions. Privileged instructions are typically used by operating systems. For example, operating systems manage memory through page table assignments or mask processor interrupts from peripheral devices. If a privileged instruction is issued while the processor is executing unprivileged code (e.g. user-space code), the processor generates a *trap*, which transfers control to the operating system. The operating system then decides how to handle the trap or terminates the application.

The basic strategy of trap-and-emulate virtualization is to *de-privilege* the virtual machine—including its operating system— and rely on the trap mechanism to intercept attempts to use privileged instructions. In this way, the hypervisor ultimately maintains control of physical resources at all times. During execution of the VM, the operating system may, for example, attempt to modify the page tables in order to context switch between its user processes. In a trap-and-emulate system, this instruction would trap to the hypervisor, which would emulate the instruction by modifying a page table structure maintained by the hypervisor, instead of the hardware page tables. The hypervisor performs this trick for every privileged instruction, throughout the execution of the VM.

A.2.2 x86 Issues

Unfortunately, the x86 instruction set architecture traditionally contained some privileged instructions, which, when executed in unprivileged mode, did not trap. Instead, some instructions simply have different semantics depending on the privilege context of the executing program. For example, the "pop flags" (`popf`) instruction, when executed in privileged context may change the state in the processor that contains the result of a branch instruction (e.g., the zero flag (ZF)) or the state in the processor that contains enables and disables interrupts on the processor (e.g., the interrupt flag IF). In unprivileged context, attempts to modify the interrupt flag are suppressed; no trap is generated. Therefore, the x86 architecture cannot be used in a strictly trap-and-emulate system. Instead, sophisticated software techniques, including binary translation and paravirtualization, were developed to virtualize the x86 instruction set architecture, further discussed in Section A.3. A concise description of the issues with trap-and-emulate in the x86 architecture appears in [33], in which hardware x86 virtualization techniques are compared with software techniques to virtualize the x86.

By 2006, x86 hardware vendors introduced hardware extensions for virtualization [103, 160], which contained a privilege level called *non-root mode*. In this architecture, there are no privileged instructions that do not trap.[1] Therefore, the

[1]Root mode and non-root mode are a different privilege scheme than the standard x86 priv-

x86 architecture now supports the design of a classic trap-and-emulate hypervisor. In Xen, a popular VMM, the use of hardware extensions for virtualization is referred to as hardware-assisted virtualization (HVM). Despite the emergence of these hardware extensions, software techniques for virtualization like binary translation and paravirtualization are still used by many cloud providers on old hardware or by nested virtualization solutions like the Xen-Blanket (Chapter 4).

A.3 Software x86 Virtualization Techniques

In order to surmount the inability of x86 to support the trap-and-emulate technique for virtualization, new software-based x86 virtualization techniques emerged at the turn of the 21st century. The leading two techniques are binary translation and paravirtualization.

A.3.1 Binary Translation

One approach to handling x86 instructions in a VM that do not trap is to modify the binary code of the VM such that it transfers control into the hypervisor, or performs an action on behalf of the hypervisor. This technique, and variants of it, are known as *binary translation*. A similar dynamic translation concept is used in Just-in-Time (JIT) compilers, where Java bytecode is dynamically converted to native x86 during runtime. VMware [140] is a virtualization company that produces extremely popular virtualization platforms based on binary translation.

ilege levels, which are still present in both modes in processors with virtualization extensions.

Binary translation allows most instructions to run natively on the processor, with no interpretation. Therefore binary translation achieves high performance. It can also support unmodified guest operating systems. However, the lack of a popular open-source binary translation hypervisor has allowed other approaches (such as paravirtualization) to become popular.

A.3.2 Paravirtualization

Another approach to handling x86 instructions in a VM that do not trap is to modify the code of the guest OS. Problem instruction sequences can be replaced by *hypercall*s, or system calls into the hypervisor. Xen [41] is an open-source hypervisor that uses paravirtualization to virtualize the x86.

Paravirtualization boasts high performance, especially for device I/O, where large amounts of data can be transferred to the hypervisor (and subsequently the I/O device) in a single call, instead of byte by byte, as is the case in many emulated device strategies. However, paravirtualization suffers from the requirement that the OS within the VM must be modified. As an example, there is no support for Windows because it has not been modified to run on Xen.[2] Despite this limitation, paravirtualization continues to be used by many cloud providers including Amazon EC2.

[2]Paravirtualized device drivers have been developed for Windows, but binary translation or hardware assisted virtualization are required to support the (unmodified) OS.

A.3.3 Tradeoffs

The main advantage of binary translation over paravirtualization is that guests can run unmodified. In particular, guests running operating systems that have not been paravirtualized, such as Microsoft Windows, are supported with binary translation. On the other hand, the main advantage of paravirtualization over binary translation is performance. Performance results between VMware Workstation [140] (which uses binary translation) and Xen [41] (which uses paravirtualization) have demonstrated the performance advantages. However, quantitative performance comparisons with more robust VMware products, such as ESX server [145], are forbidden due to due to the terms of the product's End User License Agreement [41].

APPENDIX B
BACKGROUND ON XEN

Xen [41] is a popular open source virtual machine monitor (VMM). It has been embraced by cloud providers such as Amazon EC2 and Rackspace. Xen is also frequently used in academic research, due to its availability and open source community. Xen originally used paravirtualization to virtualize the x86 instruction set architecture (ISA), but also supports unmodified guests, using the x86 hardware extensions [103, 160].

B.1 Architecture

A Xen-based cloud consists of three parts, depicted in Figure B.1 First, the Xen *hypervisor proper* is a thin software layer that multiplexes the bare hardware. Second, the control domain, called *Domain 0 (or Dom 0)*, interacts with the hypervisor proper in order to implement most hypervisor-level functionality, including the creation and destruction of guest VMs and VM live migration. Finally, guest VMs, each of which are referred to as a *Domain U (or Dom U)*, run on top of the hypervisor. Guest VMs on Xen can be either paravirtualized or run unmodified in hardware-assisted virtualization (HVM) mode. Typically, the cloud provider manages Domain 0 and the hypervisor proper, while the cloud user controls the Dom Us. The hypervisor proper and Domain 0 make up the Xen VMM and their combination is often referred as "the Xen hypervisor". Throughout this dissertation, we use the term *hypervisor proper* to refer to the thin layer running on bare metal and the terms *hypervisor* or *VMM* to refer to the hypervisor proper and Domain 0.

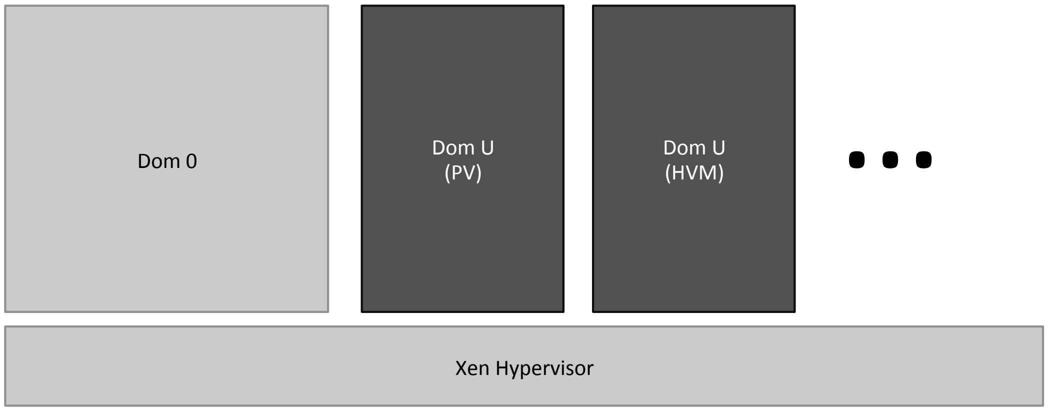


Figure B.1: In a Xen-based cloud, Domain 0 and the hypervisor proper (light gray) are typically managed by the provider, whereas guest VMs, called Dom U's (dark gray) are controlled by the user.

Like an operating system, the hypervisor proper multiplexes physical hardware. Instead of running and switching between processes, the hypervisor proper runs and switches between VMs. However, the mechanisms in the hypervisor proper are identical to an operating system like Linux. For example, the hypervisor proper interacts with the Advanced Programmable Interrupt Controller (APIC) to register timer interrupts and implement scheduling policies. If the physical system contains multiple CPU cores, the APIC may also be used to send interrupts from one processor to another, called Inter-Processor Interrupts (IPIs). For example, if a processor is changing context to run a different VM, it may generate IPIs to all other processors so that they can invalidate any cached page table entries.

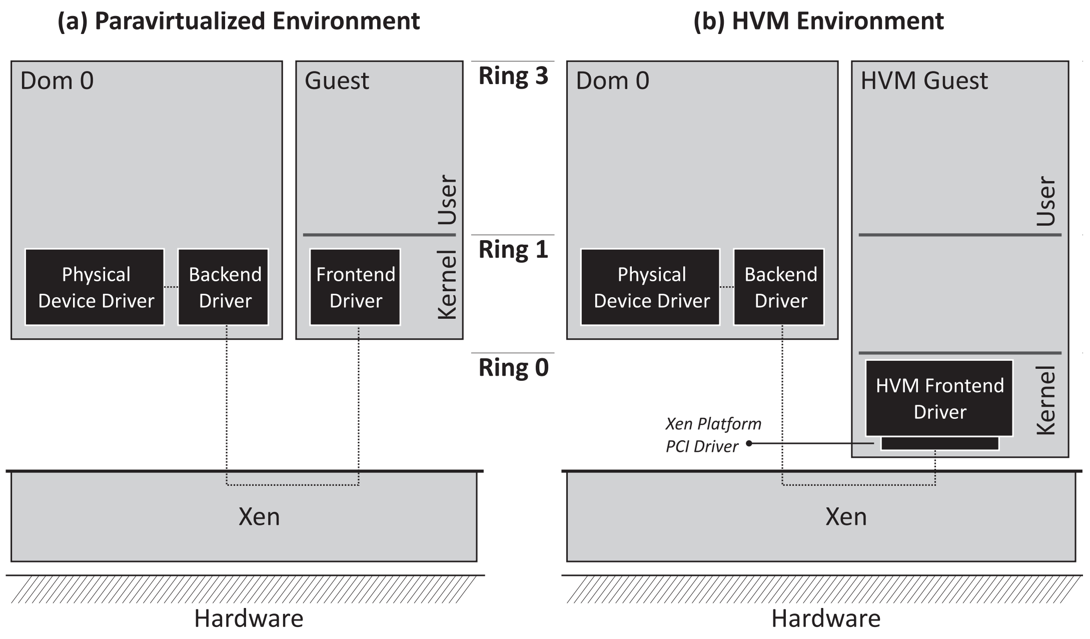


Figure B.2: Xen uses a split driver model to provide paravirtualized device I/O to guests.

B.2 Device I/O

Domain 0 maintains control of I/O devices, such as the network and disk. The operating system running in Domain 0 runs standard device drivers to operate the physical hardware. Domain 0 also acts as an intermediary for device access from guest VMs, using a *split driver* architecture.

In Xen's split driver architecture, shown in Figure B.2 each device driver is split into a *front-end* and a *back-end*. The guest VM runs the front-end driver. The back-end driver is run in Domain 0. Between the two is a paravirtualized device interface. Communication between the front-end and back-end driver is accomplished through shared memory ring buffers and an event mechanism provided by Xen. Both the guest VM and Domain 0 communicate with Xen to set up these communication channels.

A guest VM using hardware assisted mode (HVM) in Xen can either use an emulated or a paravirtualized device interface with Domain 0. Emulated devices do not utilize the split-driver model. Instead a standard device driver is used in the guest, and Xen handles traps that occur due to privileged instructions related to device access (See Appendix A for details on traps). The paravirtualized device interface for HVM guests, on the other hand, does utilize the split driver model (Figure B.2). However, it requires the guest to run additional modules to set up the machinery (e.g., shared memory ring buffers and event mechanism) to enable communication between the front and back-end. In particular, Xen exposes a *Xen platform Peripheral Component Interconnect (PCI) device* to HVM guests, which acts as a familiar environment wherein shared memory pages are used to communicate with Xen and an interrupt request (IRQ) line is used to deliver events from Xen. So, in addition to a front-end driver for each type of device (e.g. network, disk), an HVM Xen guest also contains a Xen platform PCI device driver. The front-end drivers and the Xen platform PCI driver are the only Xen-aware modules in an HVM guest. This contrasts with a paravirtualized guest, in which many subsystems in the modified OS (e.g., the memory subsystem) are aware of Xen.

B.3 Live Migration

Live migration [57] is one of many hypervisor-level features implemented in Xen and controlled from Domain 0. Live migration moves a running guest VM instance from one Xen hypervisor to another with virtually no downtime. In

Xen, the live migration mechanism is *pre-copy*, meaning the memory pages making up the running instance are copied to the destination, while the VM is still running on the source. The copying of memory pages is done in iterations. Xen keeps track of the pages that have been modified, or *dirtied*, while each iteration is taking place so that it can resend those pages the following iteration. After a number of iterations, or if the number of dirty pages is sufficiently small, the VM is suspended. A suspended VM is a VM in which no VCPUs are actively running. At this time, the final dirty pages are copied to the destination, along with the VCPU state and all other information in order for the VM to be resumed on the destination. The VM downtime in live VM migration occurs from the time the VM is suspended on the source to the time the VM is resumed on the destination. The downtime can be as low as 60 ms, which is low enough to go unnoticed for most applications running in the VM.

Typically, live VM migration is performed between two different physical machines running Xen that are on the same subnet. After the VM migrates, the switches in the network must learn that the MAC address belonging to the VM now resides on a different physical machine. To trigger an update to the MAC learning table in the switches, the VM generates an unsolicited ARP response when it arrives at the destination. An ARP request is insufficient for VMs that migrate outside of a subnet, as the IP routing must also be updated. For this reason, live migration typically does not extend outside of a single subnet.

APPENDIX C
GLOSSARY OF TERMS

binary translation: A software virtualization technique, used by VMware [140], in which the binary code of VM running an unmodified operating system is made to explicitly transfer control to the hypervisor for privileged operations. See also *virtualization*, *hypervisor*, *virtual machine (VM)*.

cloud: See *cloud computing*. See also *cloud provider*, *cloud user*.

cloud computing: According to the National Institute of Standards and Technology (NIST), cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [112]. See also *resources*.

cloud computing model: See *cloud computing*.

cloud user: The entity that obtains resources (CPU, network, storage), often for a fee, from a cloud provider. A cloud user may be an individual, a small organization, or a large enterprise. See also *virtual resources*, *cloud provider*.

cloud provider: The entity that makes computing resources made available as a utility in the cloud computing model. Cloud providers, sometimes referred to simply as clouds, manage computing resources in the form of massive data centers consisting of racks upon racks of servers connected in a network. See also *cloud computing*, *server*, *resources*.

cloud extensibility: See *extensible cloud*.

cooperative swap: An application of network memory as an overload mitigation solution in which VM swap pages are written to and read from page repositories across the network to supplement the memory allocation of an overloaded VM. See also *network memory*, *memory overload*, *virtual machine (VM)*.

data center: A collection of computational resources (CPU, network, storage), in the form of racks of servers typically connected with a network, along with the physical infrastructure, power and cooling to operate them. See [34] for best practices in data center design. See also *server*, *physical infrastructure*.

Domain 0: In Xen, the control domain that interacts with the hypervisor proper in order to implement most hypervisor-level functionality. See also *hypervisor proper*.

enterprise deployment: See *enterprise workload*.

enterprise workload: A set of applications and the services and infrastructure required to support them. See also *network components*, *physical infrastructure*.

enterprise workload migration: Moving part or all of an enterprise workload—originally running on physical or virtual infrastructure—to a cloud. See also *enterprise workload*, *cloud provider*, *physical infrastructure*.

extended page tables (EPT): An Intel processor feature that allows VMs to modify their own page tables and directly handle page faults, thereby avoiding the overhead of interacting with the VMM for these operations [103]. See also *virtual machine (VM)*, *virtual machine monitor (VMM)*

extensible cloud: A cloud in which cloud users are able to modify the abstraction implemented by a cloud provider by augmenting its hypervisor-level functionality. See also *cloud provider*, *cloud user*, *cloud computing*.

flow policies: The paths—through switches, routers, and middleboxes—packets follow through a complex network. Flow policies can be encoded in low-level device configurations, including switches, routers, middleboxes and firewalls. See also *network components*, *middlebox*.

hardware-assisted virtualization (HVM): A mode of virtualization that Xen uses to run unmodified guest VMs. See also *virtual machine (VM)*.

homogenized clouds: Clouds which share a uniform interface and environment, supporting an identical VM image format and hypervisor-level services. See also *cloud provider*, *VM image*, *virtual machine (VM)*.

homogenization: The process of achieving a uniform interface and environment across clouds. See also *homogenized clouds*.

hyperthreading: An Intel processor feature that enables multiple threads to run on each processor core [107].

Infrastructure-as-a-Service (IaaS): A type of cloud in which the cloud user "is able to deploy and run arbitrary software, which can include operating systems and applications" [112]. In practice, an IaaS cloud provider offers a virtual machine (VM) abstraction. See also *cloud provider*, *cloud user*, *cloud computing*, *virtual machine (VM)*.

guest: See *virtual machine (VM)*.

hypervisor: For the purposes of this dissertation, a synonym for VMM. See also *virtual machine monitor (VMM)*.

hypervisor proper: In Xen, the thin software layer that multiplexes the bare hardware, not including Domain 0. See also *Domain 0*.

live VM migration: Moving a running VM from one physical machine to another with little or no interruption of service [57, 122]. See also *virtual machine (VM)*, *server*.

lock-in: A situation in which it is prohibitively expensive for a cloud user to switch from one provider to another [32]. See also *cloud user*, *cloud provider*.

machine memory: Physical memory on a physical machine. See also *server*.

memory ballooning: A technique that a VMM uses to reclaim memory from a guest VM, allowing it to perform as if it had been configured with less memory [145]. See also *virtual machine monitor (VMM)*, *virtual machine (VM)*.

memory overload: See *overload*, applied to memory resources. Memory overload can be characterized by one or more VMs swapping its memory pages out to disk, resulting in severely degraded performance. See also *page scan rate*, *paging rate*.

middlebox: Any device or server in a network performing functions other than the standard functions of an Internet Protocol (IP) router on the path between a source host and destination host [52] (e.g., firewall, network address translator (NAT), protocol accelerator, Wide Area Network (WAN) optimizer). See also *server*, *network component*.

migration: See *VM migration* or *enterprise workload migration*.

multi-cloud: Spanning two or more cloud providers. See also *cloud provider*.

nested virtualization: A hypervisor that runs one or more other hypervisors and their associated virtual machines as a guest of the underlying hypervisor [43]. See also *hypervisor*, *virtual machine (VM)*.

network components: Switches, routers, and middleboxes, on the network paths connecting servers in a data center. See also *servers*, *middlebox*.

network memory: Reading and writing memory on a machine across the network for efficient paging [36].

resources: See *physical resources* or *virtual resources*.

overload: A situation in which a VM's resource demands exceed the amount of resources it has been allocated. See also *page scan rate*, *paging rate*.

overprovisioning: A technique in which each VM in an enterprise workload is allocated enough resources to support relatively rare peak load conditions. See also *virtual machine (VM)*, *enterprise workload*, *virtual resources*.

oversubscription: A technique in which the amount of resources actually allocated to the VMs comprising an enterprise workload is less than the amount of resources that were requested. See also *virtual machine (VM)*, *enterprise workload*, *virtual resources*.

page scan rate: The rate at which an OS scans memory to find free pages. A high page scan rate indicates overload. See also *overload*, *paging rate*.

page sharing: A method VMMs use to eliminate redundant copies of pages across VMs by mapping the same pages into multiple VMs and protecting them with copy-on-write. [49, 145]. See also *virtual machine (VM)*, *virtual machine monitor (VMM)*.

paging rate: The rate at which an OS writes memory pages to secondary storage in order to free the memory for other uses. A high paging rate indicates overload. See also *overload*, *page scan rate*.

paravirtualization: A software virtualization technique, used by Xen [41], in which a guest operating system within a VM is modified to explicitly transfer control to the hypervisor for privileged operations. See also *virtualization*, *hypervisor*, *virtual machine (VM)*.

physical infrastructure: The physical hardware components that make up a data center, including servers, networking equipment and storage equipment. See also *server*, *network components*.

physical machine: A computer containing physical resources upon which software is executed. See also *physical resources*.

physical resources: Computational resources provided by physical infrastructure and machines, including CPU, memory, network, storage.

processor core: The units in a processor that read and execute program instructions.

provider-centric: Defined and controlled by a cloud provider, rather than a cloud user. See also *user-centric*, *cloud provider*, *cloud user*.

provider lock-in: See *lock-in*

server: See *physical machine*.

split driver: A paravirtualized device driver architecture, used by Xen, in which a guest VM runs a front-end driver and Domain 0 runs a back-end driver. The front- and back-end drivers communicate with a paravirtualized interface. See also *Domain 0*, *virtual machine (VM)*.

stop-and-copy VM migration: Suspending a VM on one physical machine and resuming it on another physical machine [99]. See also *live VM migration*, *virtual machine (VM)*.

supercloud: A user-centric cloud computing environment that is not bound to any provider or physical resources and can span multiple cloud providers. See also *user-centric*, *cloud provider*, *cloud computing*.

sustained overload: Overload that lasts for a long duration. See also *overload, transient overload*.

third-party cloud: A cloud provider that is not the same entity as the cloud user. See *cloud provider*, *cloud user*.

traffic trombones: Network routing configurations that involve unnecessary crossings of expensive network links.

transient overload: Short, unexpected bursts of overload. See also *overload, sustained overload*.

user-centric: Defined and controlled by a cloud user, rather than a cloud provider. See also *provider-centric*, *cloud provider*, *cloud user*.

vendor lock-in: See *lock-in*

virtualization: The separation of a resource or request for a service from the underlying physical delivery of that service [143]. Applied to servers, virtualization yields virtual machines. See also *server*, *virtual machine (VM)*, *resources*, *virtual resources*.

virtual machine (VM): An efficient replica of a computer system in software, complete with all the processor instructions and system resources (i.e., memory and I/O devices) [72]. See also *virtualization*, *server*.

virtual machine monitor (VMM): A software layer that virtualizes hardware resources, exporting a virtual hardware interface that reflects the underlying machine instruction set architecture [145]. See also *virtual machine*, *virtualization*.

virtual resources: Virtual equivalents of physical resources, exposed to VMs by the VMM, and ultimately backed by physical resources. See also *virtual machine monitor (VMM)*.

VM density: The number of VMs per physical machine. See also *virtual machine (VM)*, *physical machine*.

VM image: The representation of a VM on disk, including a description of its virtual devices (virtual network interfaces, virtual CPUs, virtual storage devices) and the contents of its virtual storage devices. See also *virtual machine (VM)*.

VM instance: A running instantiation of a VM image, including its memory allocation and virtual devices (virtual network interfaces, virtual CPUs, virtual storage devices). See also *virtual machine (VM)*, *VM image*.

VM migration: Moving a VM instance from one physical machine to another [99]. See also *live VM migration*.

BIBLIOGRAPHY

[1] Amazon Virtual Private Cloud. http://aws.amazon.com/vpc/.

[2] BigIP. http://www.f5.com/products/big-ip.

[3] CloudSwitch. http://www.cloudswitch.com.

[4] Configuring ERSPAN. http://www.cisco.com/en/US/docs/ios/ios_xe/lanswitch/configuration/guide/span_xe.pdf.

[5] Continuous Systems, Nonstop Operations with JUNOS Operating System. http://www.juniper.net/us/en/local/pdf/whitepapers/2000317-en.pdf.

[6] ebtables. http://ebtables.sourceforge.net/.

[7] Fiber Channel over Ethernet (FCoE). http://www.t11.org/fcoe.

[8] Future of Cloud Computing Survey. http://www.futurecloudcomputing.net/2011-future-cloud-computing-survey-resul

[9] Interceptor. http://www.riverbed.com.

[10] Introduction to Cisco IOS Flexible NetFlow. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/ps6965/prod_white_paper0900aecd804be1cc.pdf.

[11] Linux-HA. http://www.linux-ha.org/wiki/Main_Page.

[12] netfilter: firewalling, NAT and packet mangling for Linux. http://www.netfilter.org/.

[13] Network Load Balancing Technical Overview. http://technet.microsoft.com/en-us/library/bb742455.aspx.

[14] Nicira. http://www.nicira.com.

[15] PaceMaker. http://www.clusterlabs.org.

[16] Sun Cluster. http://docs.oracle.com/cd/E19787-01/819-2993/index.html.

[17] Sun imagines world of virtual servers.

[18] The Cisco Network Simulator & Router Simulator. http://www.boson.com/netsim-cisco-network-simulator.

[19] The Linux Kernel Archives. http://www.kernel.org/.

[20] tinc. http://www.tinc-vpn.org/.

[21] Using the Cisco IOS Command-Line Interface. http://www.cisco.com/en/US/docs/ios/fundamentals/configuration/guide/cf_cli-basics.pdf.

[22] vCider. http://www.vcider.com.

[23] VMware Launches Virtualization Software for Windows-Based Intel Servers. http://www.vmware.com/company/news/releases/gsx_win_release.html.

[24] VPN-Cubed. http://www.cohesiveft.com/vpncubed/.

[25] vSphere. http://www.vmware.com.

[26] xentop. http://www.xen.org/.

[27] memcached. http://www.danga.com/memcached/, May 2003.

[28] Station and Media Access Control Connectivity Discovery (802.1AB), 2005.

[29] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/, October 2008.

[30] OpenStack. http://www.openstack.org/, October 2010.

[31] Rackspace. http://www.rackspace.com/, September 2012.

[32] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, Indianapolis, IN, June 2010.

[33] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of ACM ASPLOS*, San Jose, CA, October 2006.

[34] ADC Telecommunications, Inc. Tia-942: Data center standards overview. `http://www.adc.com/us/en/Library/Literature/102264AE.pdf`, January 2006.

[35] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *ACM SIGCOMM CCR*, 38(4):63–74, 2008.

[36] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[37] Mauro Andreolini, Sara Casolari, Michele Colajanni, and Michele Messori. Dynamic load management of virtual machines in a cloud architecture. In *Proc. of ICST CLOUDCOMP*, Munich, Germany, October 2009.

[38] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[39] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP Performance Over Wireless Networks. In *Proc. of ACM MobiCom*, Berkeley, CA, November 1995.

[40] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proc. of ACM SIGCOMM*, Toronto, Canada, August 2011.

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

[42] Jeff Barr. ”amazon ec2 beta”. August 2006.

[43] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *Proc. of USENIX OSDI*, Vancouver, BC, Canada, October 2010.

[44] O Berghmans. Nesting virtual machines in virtualization test frameworks. *Masters thesis, University of Antwerp*, May 2010.

[45] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of ACM SOSP*, Copper Mountain, CO, December 1995.

[46] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proc. of ACM VEE*, San Diego, CA, June 2007.

[47] Ryan Breen. Vtun. *Linux Journal*, 112, August 2003.

[48] Michael Brenner, Markus Garschhammer, Martin Sailer, and Thomas Schaaf. CMDB - Yet Another MIB? On Reusing Management Model Concepts in ITIL Configuration Management. In *Proc. of IEEE/IFIP DSOM*, Dublin, Ireland, October 2006.

[49] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 25(4):412–447, November 1997.

[50] Lennert Buytenhek. `brctl` Linux man page.

[51] Roy Campbell, Indranil Gupta, Michael Heath, Steven Y. Ko, Michael Kozuch, Marcel Kunze, Thomas Kwan, Kevin Lai, Hing Yan Lee, Martha Lyons, Dejan Milojicic, David O'Hallaron, and Yeng Chai Soh. Open Cirrus^{TM}cloud computing testbed: federated data centers for open source systems and services research. In *Proc. of USENIX HotCloud*, San Diego, CA, June 2009.

[52] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234, February 2002.

[53] B. Carpenter and K. Moore. Connection of IPv6 Domains via IPv4 Clouds. RFC 3056, February 2001.

[54] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. Appscale: Scalable and open appengine application development and deployment. In *Proc. of ICST CLOUDCOMP*, Munich, Germany, October 2009.

[55] Cisco Systems, Inc. Understanding VLAN Trunk Protocol (VTP). `http://www.cisco.com/application/pdf/-paws/10558/21.pdf`, July 2007.

[56] Cisco Systems, Inc. Cisco nexus 1000v series switches data sheet. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.html`, April 2012.

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

[58] Tim Clark. Rightscale. http://www.rightscale.com, 2010.

[59] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proc. of AFIPS Fall Joint Computer Conference, part I*, November 1965.

[60] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[61] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. of USENIX NSDI*, San Francisco, CA, April 2008.

[62] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of USENIX OSDI*, Monterey, CA, November 1994.

[63] Distributed Management Task Force, Inc. (DMTF). Open virtualization format white paper version 1.00. `http://http://www.dmtf.org/`

sites/default/files/standards/documents/DSP2017_1.0.0.pdf, February 2009.

[64] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of ACM SOSP*, Big Sky, MT, October 2009.

[65] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[66] Emmanuel Cecchet and Julie Marguerite and Willy Zwaenepoel. Performance and Scalability of EJB Applications. In *Proc. of OOPSLA*, Seattle, WA, November 2002.

[67] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP*, Copper Mountain, CO, December 1995.

[68] Eucalyptus Systems, Inc. Eucalyptus open-source cloud computing infrastructure - an overview. http://www.eucalyptus.com/pdf/whitepapers/Eucalyptus_Overview.pdf, August 2009.

[69] R. M. Fano. The MAC system: the computer utility approach. *IEEE Spectrum*, 2:56–64, January 1965.

[70] Dan Farber. "Utility computing: What killed HP's UDC?". http://www.zdnet.com/news/utility-computing-what-killed-hps-udc/138727.

[71] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of ACM SOSP*, Copper Mountain, CO, December 1995.

[72] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[73] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on

shared-memory multiprocessors. In *Proc. of ACM SOSP*, Charleston, SC, December 1999.

[74] Alexander Graf and Joerg Roedel. Nesting the virtualized world. In *Linux Plumbers Conference*, Portland, OR, September 2009.

[75] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, Spain, August 2009.

[76] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proc. of ACM SIGCOMM PRESTO*, Seattle, WA, August 2008.

[77] F. Gruenberger. Computers and communications - toward a computer utility. *Prentice Hall*, 1968.

[78] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR*, 38(3), July 2008.

[79] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. of ACM SIGCOMM*, Spain, August 2009.

[80] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proc. of ACM SIGCOMM*, August 2008.

[81] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of USENIX OSDI*, San Diego, CA, December 2008.

[82] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing Extensible IP Router Software. In *Proc. of USENIX NSDI*, Boston, MA, May 2005.

[83] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describ-

ing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411, December 2002.

[84] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proc. of ACM VEE*, Washington, DC, March 2009.

[85] Michael Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. of ACM VEE*, Washington, DC, March 2009.

[86] Michael R. Hines and Kartik Gopalan. MemX: supporting large memory workloads in Xen virtual machines. In *Proc. of IEEE VTDC*, Reno, NV, November 2007.

[87] S. Hopkins and B. Coile. AoE (ATA over Ethernet). `http://support.coraid.com/documents/AoEr11.txt`, February 2009.

[88] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1.2), January 2008.

[89] Jill Jones. *Empires of Light: Edison, Tesla, Westinghouse, and the Race to Electrify the World*. Random House, 2004.

[90] Rick Jones. netperf. `http://www.netperf.org/netperf/`.

[91] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of ACM SOSP*, St. Malo, France, October 1997.

[92] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *Proc. of ACM/IEEE SC*, New Orleans, LA, November 2010.

[93] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[94] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application performace management in virtualized server environments. In *Proc. of IEEE/IFIP NOMS*, Vancouver, Canada, April 2006.

[95] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. of ACM SIGCOMM*, Seattle, WA, August 2008.

[96] A. Kivity, Y. Kamay, and D. Laor. KVM: The Kernel-Based Virtual Machine for Linux. In *Proc. of Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[97] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and Frans M. Kaashoek. The Click Modular Router. *ACM Transactions on Compuer Systems*, 18:263–297, August 2000.

[98] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, , Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of USENIX OSDI*, Vancouver, Canada, October 2010.

[99] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. of IEEE WMCSA*, Calicoon, NY, June 2002.

[100] Maxim Krasnyansky. Universal TUN/TAP device driver, 1999.

[101] H. Andres Lagar-Cavilla, Joseph Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and M. Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proc. of ACM EuroSys*, Nuremberg, Germany, April 2009.

[102] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proc. of ACM HotNets*, Monterey, California, 2010.

[103] Felix Leung, Gil Neiger, Dion Rodgers, Amy Santoni, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualzation. *Intel Technology Journal*, 10(3), August 2006.

[104] T. Li, B. Cole, P. Morton, and D. Li. Cisco Hot Standby Router Protocol (HSRP). RFC 2281, March 1998.

[105] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-00`, August 2011.

[106] Evangelos P. Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, January 1996.

[107] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. 6(1), 2002 Intel Technology Journal.

[108] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[109] E. Michael Maximilien, Ajith Ranabahu, Roy Engehausen, and Laura C. Anderson. IBM Altocumulus: A Cross-cloud Middleware and Platform. In *Proc. of ACM OOPSLA Conf.*, Orlando, FL, October 2009.

[110] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 38(2), April 2008.

[111] Larry Mcvoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, January 1996.

[112] Peter Mell and Timothy Grance. The NIST definition of cloud computing. In *National Institute of Standards and Technology Special Publication 800-145*, September 2011.

[113] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. NetBench: a benchmarking suite for network processors. In *Proceedings of the IEEE/ACM ICCAD*, 2001.

[114] Aravind Menon and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference*, Boston, 2006.

[115] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, and Songnian Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[116] Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, June 2009.

[117] B. Mitchell, J. Rosse, and T. Newhall. Reliability algorithms for network swapping systems with page migration. In *Proc. of IEEE CLUSTER*, San Diego, CA, September 2004.

[118] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *Proc. of ACM SIGCOMM*, Toronto, Canada, August 2011.

[119] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. of ACM SIGCOMM*, August 2008.

[120] S. Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, March 2010.

[121] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[122] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proc. of USENIX Annual Technical Conf.*, Anaheim, CA, April 2005.

[123] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A Network Swapping Module for Linux Clusters. In *Proc. of Euro-Par*, Klagenfurt, Austria, August 2003.

[124] OpenVPN Technologies, Inc.

[125] Anthony PERARD and Stefano Stabellini. Virtio on xen. In *Linux Plumbers Conference*, Santa Rosa, CA, September 2011.

[126] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martín Casado, and Scott Shenker. Extending Networking Into the Virtualization Layer. In *Proc. of ACM HotNets*, New York, NY, October 2009.

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

[128] Hangwei Qian, Elliot Miller, Wei Zhang, Michael Rabinovich, and Craig E. Wills. Agility in virtualized utility computing. In *Proc. of IEEE VTDC*, Reno, NV, November 2007.

[129] Bruno Quoitin, Luigi Iannone, Cédric de Launois, and Olivier Bonaventure. Evaluating the Benefits of the Locator/Identifier Separation. In *Proc. of ACM/IEEE MobiArch*, 2007.

[130] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. SecondSite: disaster tolerance as a service. In *Proc. of ACM VEE*, March 2012.

[131] B. Rochwerger, D. Breitgand, A. Epstein, D. Hadas, I. Loy, K. Nagin, J. Tordsson, C. Ragusa, M. Villari, S. Clayman, E. Levy, A. Maraschini, P. Massonet, H. Muñoz, and G. Tofetti. Reservoir - when one cloud is not enough. *IEEE Computer*, 44(3):44–51, 2011.

[132] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proc. of USENIX LISA*, November 1999.

[133] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network file system. In *Proc. of USENIX Annual Technical Conf.*, 1985.

[134] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[135] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of USENIX OSDI*, Seattle, WA, October 1996.

[136] Vivek Shrivastava, Petros Zerfos, Kang won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware Virtual Machine

Migration in Data Centers. In *Proc. of IEEE INFOCOM Mini-conference*, Shanghai, China, April 2011.

[137] M. Sridharan, K. Duda, I. Ganga, A. Greenberg, G. Lin, M. Pearson, P. Thaler, C. Tumuluri, N. Venkataramiah, and Y. Wang. NVGRE: Network Virtualization using Generic Routing Encapsulation. `http://tools.ietf.org/html/-draft-sridharan-virtualization-nvgre-00`, September 2011.

[138] Alexander Stage and Thomas Setzer. Network-aware migration control and scheduling of differentiated virtual machine workloads. In *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, Canada, May 2009.

[139] Standard Performance Evaluation Corporation. Specweb2009 release 1.10 banking workload design document. `http://www.spec.org/web2009/docs/design/BankingDesign.html`, April 2009.

[140] Jeremey Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of USENIX Annual Technical Conf.*, Boston, MA, June 2001.

[141] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, Canada, May 2009.

[142] VMware. "VMware vsphere, the first cloud operating system, provides an evolutionary, non-disruptive path to cloud computing". `http://www.vmware.com/files/pdf/cloud/VMW_09Q2_WP_Cloud_OS_P8_R1.pdf`, 2009.

[143] VMware, Inc. Virtualization Overview. `http://www.vmware.com/pdf/virtualization.pdf`, 2006.

[144] VMware, Inc. vSphere Networking. `http://pubs.vmware.com/vsphere-50/topic/com.vmware.ICbase/PDF/vsphere-esxi-vcenter-server-50-networking-guide.pdf`, 2011.

[145] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[146] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes No Longer Considered Harmful. In *Proc. of USENIX OSDI*, San Francisco, CA, December 2004.

[147] Brian Walters. VMware Virtual Platform. *Linux Journal*, 63, July 1999.

[148] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus van der Merwe, and Jennifer Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proc. of ACM SIGCOMM*, Seattle, WA, August 2008.

[149] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. In *Proc. of USENIX Annual Technical Conf.*, Anaheim, CA, April 2005.

[150] Steven Weber and Rema Hariharan. A new synthetic web server trace generation methodology. In *Proc. of IEEE ISPASS*, Austin, TX, March 2003.

[151] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proc. of ACM SoCC*, Indianapolis, IN, June 2010.

[152] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[153] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of USENIX OSDI*, Boston, MA, December 2002.

[154] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *Proc. of ACM VEE*, Newport Beach, CA, March 2011.

[155] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. of ACM EuroSys*, Bern, Switzerland, April 2012.

[156] Timothy Wood, H. Andrés Lagar-Cavilla, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Pipecloud: using causality to over-

come speed-of-light delays in cloud-based disaster recovery. In *Proc. of ACM SoCC*, Cascais, Portugal, October 2011.

[157] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. CloudNet : Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. of ACM VEE*, Newport Beach, CA, March 2011.

[158] Timothy Wood, Prashant Shenoy, and Arun Venkataramani. Black-box and gray-box strategies for virtual machine migration. In *Proc. of USENIX NSDI*, Cambridge, MA, April 2007.

[159] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proc. of ACM VEE*, Washington, DC, March 2009.

[160] Alan Zeichick. Processor-based virtualization, AMD64 style, part I,II. `http://developer.amd.com/documentation/articles/pages/630200614.aspx`, June 2006.

[161] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of ACM SOSP*, Cascais, Portugal, October 2011.