System Architectures with Virtualized Resources in a Large-Scale Computing Infrastructure

by

Chang-Hao Tsai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

To my family

Acknowledgments

This thesis cannot be completed without help from many people. All of them are very important to me.

First of all, I would like to thank Professor Shin for his constant guidance, encouragement, and support. He patiently allowed me to explore many different fields while seeking for my own topic. He also continuously fostered my work with his wisdom and years of experience. His fatherly advice goes beyond research work and prepares me to take on bigger challenges in my career.

I am also grateful to my family. Their endless love and support always enable me to boldly deal with obstacles in the Ph.D. journey. They believe in me all times and they are always there when I need them. In particular, I want to express my deepest gratitude to my dear wife, Jessie, for her enduring love, encouragement, and understanding during my study. This thesis is dedicated to my family.

I would also like to thank Professors Prakash, Flinn, and Tilbury for their invaluable suggestions and comments to strengthen the dissertation.

My colleagues at the Real-Time Computing Lab also enriched my study and my life. I would like to thank John Reumann, Hani Jamjoom, Chun-Ting Chou, and Hai Huang for their mentoring, insights, and countless help since my first day here. Mohamed El-Gendy, Songkuk Kim, Abhijit Bose, Jian Wu, Zhigang Chen, Kyu-Han Kim, Jisoo Yang, Pradeep Padala, Karen Hou, and all other members also provided me suggestions and priceless friendship. I also thank BJ Monahgan and Stephen Reger for their administrative support.

Finally, I would like to thank my daughter, Maddie, for her giggles and smiles.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last couple of decades, the functions of the Internet have evolved from providing a simple text-only email service to information retrieval, multi-media streaming, on-line advertising, shopping, auction, banking, trading, and many other business/mission-critical tasks. Both businesses and people have now become relying heavily on the Internet for their everyday lives and activities. In order to provide these sophisticated services, new server architectures have been developed and more servers are deployed to provide scalable, highly available and reliable services.

When building a server system, designers usually over-provision its capacity to meet the peak demand in the foreseeable future. When multiple servers are needed, a server architecture is designed and then each server machine is configured. Early planning is necessary as server procurement may take days or even weeks. These servers are usually installed in designated rooms/areas in each server owner's premises. In order to achieve high availability, both backup power sources and redundant network connections are provided.

However, this practice suffers from several problems. First, for small businesses, the cost of having redundant network connections and backup power sources can be unaffordable. Besides, future server workloads are usually estimated very conservatively during capacity planning, yielding much larger servers than needed in practice. Workload fluctuations and long resource procurement delays in addition to the over-provisioning usually cause many of the servers to be severely under-utilized. Consequently, service providers need to pay a high initial cost and a higher operating cost to run Internet services even for the under-utilized servers.

1.1 Evolution of Server Infrastructures

Many data centers have been built and there have been numerous efforts to reduce the cost of operating Internet servers. Customers move servers into data centers to lower the infrastructure cost. Some data centers also provide the customers resources on demand. Customers request resources only when needed, and pay for the resources they have actually used. Furthermore, virtualization allows for flexible resource partitioning. Recently, cloud computing removes the burden of resource management by providing high-level services instead of raw computing resources. The evolution of server infrastructures is briefly described next.

1.1.1 Data Centers

Internet data centers facilitate server hosting. Some data centers are fully owned and operated by a single organization while others provide infrastructures and services to their customers. Inside a data center, racks of servers are organized into rows to increase server density on the limited floor space while allowing sufficient air flows for heat dissipation. Data center customers may rent rack spaces and mount their own server machines. They may also obtain pre-configured servers inside a data center. Typically, pre-configured servers are available in few different configurations only and the lease duration ranges from months to years. However, they reduce the resource procurement delay to hours or a few days and allow a server system to grow or shrink with the services they provide.

Data centers also provide a common network and electricity infrastructure to its customers. A data center usually connects to the Internet via multiple networks to improve redundancy. The electricity infrastructure of a data center may connect to one or several power grids via multiple power lines, and provide backups with uninterruptible power supplies (UPSs) and diesel power generators. In addition, both temperature and humidity are controlled inside a data center and the physical access to data centers is usually restricted to ensure physical security.

Since these infrastructures are shared among many servers and customers, data centers can benefit from statistical multiplexing by providing an infrastructure that is smaller than the sum of all customers' maximal demands. In addition, the economics of scale also enables data centers to provide these resources at a lower cost. By moving servers to a data center, a customer can reduce the infrastructure and operating costs.

1.1.2 Utility Computing

Although acquiring pre-configured servers from data centers allows a service to expand its capacity in a shorter time, idle resources cannot be returned to the data center or utilized otherwise easily. One approach to improving resource utilization is to allow more flexible resource allocation. When a customer needs more computing resources, the data center allocates unused resources to the customer. These new resources become ready for use very quickly (in minutes or hours.) If the resource need diminshes, the customer can return excessive resources to the data center. Customers are motivated to do this because they are only charged for the amount of resources for the duration they have held. This is similar to the usage of other utilities such as power and water. So, it is called *utility computing* and the data centers that support this scheme are called *utility data centers*. Examples of utility data centers include HP Utility Data Center (UDC), Sun Grid, and Amazon Elastic Compute Cloud (EC2).

In an utility data center, a complete physical machine, including CPU, memory, network bandwidth, and so on, can become the unit of computing resource allocation. If a data center operator runs an operating system (OS) that supports resource partitioning and metering on its servers, the operator can also divide its servers into partitions and allocate a subset of partitions to customers. A current trend in data centers is to use virtualization technologies for partitioning physical resources, which we will discuss next. Storage space is also another type of resources that is commonly seen in utility data centers (e.g., Amazon Simple Storage Service.)

1.1.3 Virtualization

Server virtualization creates an illusion of having multiple server hardware, called *virtual machines* (VMs), on top of a single physical machine. Each VM has one or more CPUs, some memory space, disk storage, network interfaces, and so on. Each VM also needs a complete OS to operate. In data centers, VMs can be used as resource containers that provide more fine-grained resource allocation than physical machines. Data center operators then allocate VMs to their customers, not physical machines. Because of its simpler abstraction, VMs can also be migrated from a physical machine to another without suspending their execution.

Virtual machine monitors (VMMs), also called hypervisors, are the piece of software that partitions physical resources and coordiates their use by VMs. Some VMMs depend on an underlying OS, called *host OS*, to manage physical resources. Other VMMs sit directly on the physical hardware and control the access to the underlying hardware. More advanced VMMs also support resource overbooking. For example, multiple VMs can share one CPU core and the total amount of memory space of all VMs may exceed the amount of physical memory installed on the physical hardware. For time-shared resources, schedulers are used to coordinate resource usage by VMs. While the resource utilization is improved, this may increase response time and decrease fault-tolerance. Usually, a high-level performance monitoring approach is used to detect performance issues and use VM migration to alleviate the problem.

Similar to the resource-partitioning facilities provided by certain OSes, VMs can be created in seconds and the amount of resource allocated to each VM can be adjusted to fit each customer's need at run-time. However, the abstraction of a VM gives its user the flexibility to choose an appropriate OS. Since customers have full control over the software running within their VMs, they can migrate their existing servers into VMs easily. Because of the flexibility in resource allocation, customers can operate their server VMs with a smaller headroom and enhance the utilization ratio. With an adequate performance monitoring strategy, data center operators can overbook resources to improve utilization further.

1.1.4 Cloud Computing

Cloud computing provides computing resources at a higher abstraction level in order to further reduce the cost of computation. Software-as-a-Service (SaaS) is a paradigm where software customers subscribe to a software package instead of buying and deploying a software by themselves. Typical examples include customer relationship management (CRM), enterprise resource planning (ERP), and other business software. These software packages usually support multi-tenancy, where a single instance of software installation can support multiple customers who subscribe to the service. Therefore, customers can share the cost of a larger infrastructure. SaaS service providers install SaaS-capable software in data centers and end-users access these software via the Internet by using a client program or simply web browsers. In addition to providing computing resources, data center operators may also provide cloud computing applications themselves.

As cloud computing moves more and more applications into data centers, the workload in data centers grows even higher. Although cloud computing customers do not need to manage computing resources themselves, the increased workload still translates to additional resource demand in data centers. In this dissertation, we focus on utility data centers that use virtualization technologies to partition and allocate virtualized computing resources to their customers or cloud computing applications.

1.2 Applications in Data Centers

Many different services and applications can benefit from the extensive computing resources in data centers. Web services can take advantage of their scalable and reliable infrastructure. Scientific computing can also use the resources in data centers for its computation needs. Furthermore, data centers can also host on-line game servers that demand a low-latency environment. We briefly discuss the requirements of these applications in this section.

1.2.1 Web Services

Web services are one of the most common applications in data centers. Examples include all sorts of Internet web sites that people use everyday and software repositories that distribute software updates automatically. A web service is usually implemented with a multi-tier server architecture, comprising front-end web servers, application servers, database servers, file servers, and so on. Web services are usually throughput-oriented with a response time requirement ranging from hundreds of milliseconds to seconds.

For web services that are accessed directly by end-users, their workload heavily depends on the number of users accessing it concurrently. The time-of-day and day-of-week effects can be easily observed in web server traces. The peak demand is often many times greater than the minimum demand. On the other hand, the workload of a software distribution web site usually coincides with new software releases, resulting in an even higher maximum-to-minimum demand ratio. If a service is provisioned for the peak demand and frequently running at its minimum capacity, the servers will be severely under-utilized most of the time during its operation. The high variation in demand makes utility data centers appealing to host web services.

1.2.2 Scientific Computing

For the scientific computing world, one common use of large clusters is to conduct scientific and engineering simulations in order to reduce the time and cost of conducting real experiments. Other applications include, but not limit to, data analysis and visualization, image manipulation, and so on. Typically, most scientific computing jobs are executed on dedicated computer clusters. However, with recent improvements in computation power, network bandwidth, and reduced interconnect latency, many computing clusters are built with the same technology used in data centers, such as multi-core processors and gigabit Ethernet.

Instead of building dedicated computer clusters, users can tap into the resources in data centers for scientific computing jobs. They may rely solely on data centers for all their needs or offload some jobs when their dedicated systems are overloaded. While those jobs may require a lot of resources for a long period of time, they usually have a relaxed response time requirement and do not need to be started immediately. Therefore, data center operators can postpone the execution when the resource demand is high.

1.2.3 On-line Game Servers

On-line game servers represent another type of workload in data centers. Similar to web servers, on-line games also employ a client/server architecture to connect all players in a game. First-person shooter (FPS) games, one of the most popular genre of computer games, are highly interactive during an active game session. Also, to be reliable, FPS game servers are expected to be responsive in order to provide satisfactory gaming experience.

The reliable infrastructure and high bandwidth Internet connections in data centers provide an adequate environment for game servers. A group of geographically-distributed game players can rent a server from a data center to host their own game sessions. Unlike web servers, those game servers only need to be run when players want to set up a game session. As network latency also factors in server responsiveness, choosing a data center that is closer to all players is preferred. Furthermore, in a utility data center, they only need to pay for the duration of the game session, making this approach more economic.

1.3 Challenges

While the underlying infrastructure provides a large amount of computing resource, developing software systems that can fully realize its potential is not straightforward. Over-provisioning is the most common cause of low resource utilization. Typically, an amount of computing resource is allocated to a server and the amount is fixed during the course of its execution. With a long resource acquisition time and a conservative prediction of future demand, service providers tend to build over-sized servers in order to avoid service degradation due to insufficient resources. However, it also results in low resource utilization and a higher operating cost.

Utility data centers and the adoption of virtualization technologies reduce the time it takes to add or remove resources from servers. Given a resource on-demand infrastructure, researchers have built adaptive controllers that manage the resource allocation to meet a pre-defined service level objective. Servers can be operated at higher utilization and resources can be utilized more efficiently.

When an application is spread across a cluster of servers, the application may also divide its work into smaller units and make a balanced workload distribution to improve utilization. However, as the scale of applications increases, using resources efficiently gets more difficult. In this section, we discuss several potential causes of inefficient resource usage.

1.3.1 Legacy Software Architecture

In multi-tier Internet web servers, each tier typically comprises a cluster of servers and a front-end load-balancer that dispatches the incoming workload to one of the back-end servers. Servers at the same tier usually provide identical or similar services, so a failed server can easily be replaced by other servers. The design of load-balancing algorithms not only needs to balance the workload on the servers and increase throughput, but the algorithms themselves also need to be scalable and fault-tolerant. Unfortunately, these goals are usually contradictory to each other and hard to be met. When the load in a cluster is not balanced, one needs more resources to handle peak demands, resulting in lower utilization.

Simple load-balancing algorithms such as round-robin or least-connection, simply consider each server as equally powerful. When servers have different processing capacity, each server is usually assigned a weight, which is used in the weighted version of such algorithms (weighted round-robin, weighted least-connection). These algorithms are highly scalable but they do not take locality into consideration. Hence, frequently-accessed files can be cached on all servers, resulting in inefficient use of memory.

Locality-aware algorithms, such as locality-aware request distribution (LARD) or URL hashing, improve cache efficiency in servers. However, they also assume each server has a fixed capacity, or the workload can be uniformly distributed to all servers. In addition, LARD keeps a request distribution history which is looked up and modified for each request. The history is a centralized and stateful component in LARD. It can become the bottleneck in large web sites and renders LARD unscalable. Although some load-balancing algorithms use a feedback control approach to adapt their approaches, no one has considered adapting server capacity, which is enabled by virtualization, as an alternative approach.

Another example is highly-parallel scientific computing programs, which are designed to balance the workload on all computing nodes. However, the scalability of each parallel program depends on the speed of CPU, network bandwidth and latency, and so on. If a computing node is waiting for the result from other nodes to continue its computation, the node can only sit idle, also resulting in lower utilization. For example, if the function of a computing job is to create a video stream and an audio stream concurrently, the audio creation program may wait for the video creation program to finish before continuing the process, resulting in idle cycles.

1.3.2 Naïve Redundancy Design

In a large data center, ten thousands or more computers are packed in a limited floor space with air conditioning and proper ventilation to keep the environment within these computers' limitations. Although each component may be rated for a long mean-time-between-failure (MTBF), failures are still likely to happen frequently in data centers. Hence, redundancy is also required to be integrated in software design. Without an appropriate plan against failures, Internet service provider may suffer significant monetary loss due to violation of service-level agreements.

Replication has been used extensively to improve reliability of computer systems. However, replication also uses more computing resources and reduces efficiency. For example, for a server cluster employing $N+1$ redundancy, the resource efficiency is reduced by $1/(N+1)$. Furthermore, when failures occur, not only some resources become unavailable to the server cluster, but existing locality may also become ineffective. Currently, data center customers usually acquire redundancy by adding spare resources but do not utilize the agility of resource allocation provided by virtualization.

1.3.3 Incomplete VM Abstraction

Each VM has a set of virtual resources and the OS within a VM also keeps a virtual system time, which applications utilize to learn the current time or achieve precise timing of certain events. When there are multiple VMs sharing a CPU, VMs that are not running cannot handle soft interrupts promptly. Therefore, the system time in each VM becomes non-contiguous and has a lower resolution. Events that apply to VMs can only be processed after starting the receiving VM, resulting in longer queuing delays.

While online game servers are not resource-intensive applications, they are required to be highly responsive so their users can enjoy good game-playing experience. One may use an admission-control-based approach to determine an appropriate level of consolidation such that the timeliness requirements can be met. However, it may also leave the system under-utilized if increasing the system load further may reduce its responsiveness. Associating timestamps with each event delivered to a VM provides the OS in the VM more information about the actual timing but such functionality is not included in existing VM abstraction.

1.4 Contributions

In this dissertation, we re-visit the system architecture in a large-scale computing infrastructure. Specifically, we assume that the infrastructure utilizes virtualization extensively for resource allocation, such as in utility data centers. Multiple VMs are consolidated to a single physical machine to improve resource utilization. In this section, we first describe the infrastructure with some assumptions and then outline the contributions of the dissertation.

1.4.1 Assumptions

There are many virtual machine monitor (VMM) implementations, each of which is different in one way or another. In this dissertation, we assume that the VMM can provide the following functions. First, the VMM is able to provide an accurate resource consumption reading on each resource for each VM. Second, for resources that are time-shared between VMs, the VMM utilizes a scheduler that can guarantee the minimal amount of resources that a VM can use even with the fiercest resource contention. These two requirements are the essentials to our resource-allocation design.

In addition, when multiple VMs are hosted on a physical machine and the unallocated physical resources cannot meet additional resource requirements of certain VMs, these VMs can be migrated to other physical hosts in the infrastructure. The VMM must support VM live migration, where the migrated VM is only unavailable to the users for a very short while ($< 100ms$).

1.4.2 Flexibility in Resource Allocation

Most of today's applications are designed for traditional, non-virtualized infrastructures. Therefore, they do not take advantage of the flexibility in resource allocation in a virtualized infrastructure. With this flexibility, some traditional software system design assumptions can be relaxed and resources can be utilized more efficiently.

We first use this flexibility to re-design Internet web server clusters in Chapter 3. As we have discussed in Section 1.3.1, existing front-end load-balancer designs assume that back-end servers each have a fixed capacity. There are two goals of the front-end load-balancer in a web server cluster, namely balancing the workload and increasing resource efficiency by improving locality. LARD improved cluster throughput and scalability by improving the locality at each server, but it also introduced a request distribution history, a centralized and stateful component in the server architecture, which can become a scalability bottleneck.

Instead of balancing server load, we separate the function of locality-aware server selection and load-balancing when the web server cluster is hosted on a VM cluster, called *virtualization-based resource allocation web (Vibra Web)*. The front-end uses a scalable URL-hashing approach to select the destination server for each incoming request. The unbalanced workload among the cluster of server VMs is then compensated by virtual resource transfer. In this work, we focus on the network bandwidth allocation while considering memory utilization. We applied the World Cup '98 web server trace and a synthetic YouTube-like workload to our design, which achieves better throughput and higher scalability.

We also design Vibra Web with fault-tolerance in mind. The redundancy feature of Vibra Web is described in Chapter 4. Redundancy in Vibra is considered in two parts: front-end load balancer and back-end server VMs. The front-end load balancer carries only little state information on ongoing connections so it can be easily arranged in a high-availability configuration. When a back-end server VM fails, we minimize the changes in the load-balancer configuration in order to retain existing locality. After a failure is detected, its workload and its virtual resources are immediately transferred to other surviving VMs. With these additional resources, the additional workload can be handled. Our evaluation shows that this approach maintains a higher throughput during the degraded operation than other load-balancing algorithms.

In Chapter 5, we propose a new CPU scheduling algorithm, *distributed proportional share (DPS) scheduling*, to allocate CPU shares to parallel programs. Instead of requesting a fixed number of nodes to be dedicated to a parallel program, the programmer only specifies the number of nodes and a minimum set of resources that the program needs. When a job is submitted to a computing cluster, the submitter also specifies a set of resource budgets, such as the maximum number of CPU shares that the program can use during its execution. When the job is scheduled to run, the cluster first creates a VM cluster that has the same number of nodes as the programmer specified, and execute the parallel program in VMs. During the execution, DPS monitors the actual CPU consumption at each VM and dynamically adapts the CPU share allocation.

In this scheme, the resource required by a parallel program is separated from the structure of a parallel program. Programmers are also relieved from optimizing algorithms to balance the workload. When workload imbalance is detected, DPS dynamically adapts the CPU-share allocation based on actual resource demand and conducts either a *fully-distributed share exchange* or a *bank-assisted share exchange*. We demonstrate that DPS can reduce the execution time of parallel programs and increase cluster utilization.

1.4.3 Filling Architectural Gap

Another reason for ineffective resource usage is due to the architectural gap between a VM and the hosting physical server. A VM shares many resources with other VMs on the physical server. In a non-virtualized environment, the OS handles physical resources directly and can optimize their usage. For example, an OS can use the elevator algorithm to improve disk I/O throughput. However, the OS in a VM, called the *guest OS*, does not have direct control of the disk hardware. Similarly, a guest OS typically shares a CPU with other VMs, and cannot schedule CPU time alone as in a real-time system.

As more game sessions are held on the Internet, the demand of a low-cost but highly-responsive game server also increases. While game servers can be consolidated on a virtualized platform, virtualization also reduces the responsiveness of a game server. For first-person shooting (FPS) games, a latency of less than 100ms is typically required for satisfactory gaming experience. Typically, game server hosting service providers employ an admission control system to meet the response time requirements.

In Chapter 6, we identified that an important piece of information, the packet arrival time at the physical NIC, is unavailable for a VM. For online game servers, an accurate timestamp is very important as the time is used as the user input time in dead reckoning calculation. Inaccurate timestamps can lead to a low-fidelity game state that does not reflect users' inputs.

We augment the virtual NIC design and allow timestamps to be taken in the host OS. The timestamp is then delivered to a guest OS with the incoming packet itself. In this approach, the packet delivery time, the time it takes to deliver a packet to a guest OS, is not included in dead reckoning calculation so the resulting game state reflect users' inputs better. Therefore, the data center operator can consolidate more game-server VMs on a physical server and provide services at a lower cost.

1.5 Thesis Structure

The rest of the dissertation is organized as follows. Chapter 3 discusses the application of flexible virtual resource allocation on Vibra Web. The hash value of the URL string in an HTTP request is used to select a subset of server VMs in the cluster. The least-loaded server in the subset is then chosen as the dispatch destination. Vibra Web is evaluated using the World Cup '98 trace and a YouTube workload generator. The redundancy enhancement in Vibra Web is detailed in Chapter 4, where the advantage of our approach is compared against other methods. In Chapter 5, we introduce the model of a VM hosting cluster for parallel computing programs and the design of DPS, including the inference of upstream VM in *fully-distributed share exchange* and the *back-assisted share exchange* algorithms. We evaluate the performance of DPS on a Xen VM cluster with MPI programs. In Chapter 6, we present the packet delivery path of a game-server VM and detail our modified virtual NIC design with evaluation results. Finally, Chapter 7 presents future directions and conclusions.

Chapter 2
Related Work

In this chapter, we review previous work in the areas related to this dissertation. The discussion starts with utility computing models and continues with platform virtualization technology, especially its applications in utility data centers. We then revisit load-balancing algorithms for web server clusters and the use of consistent hashing for similar systems. We also look at parallel computing models and CPU scheduling in computing clusters. Last but not least, on-line gaming and game server hosting are briefly discussed.

2.1 Utility Computing

Many data centers have been built in recent years with widely different usage models. Some data centers are built to meet a single company's computing needs while some others, called *utility data centers*, partition their computing resources and allocate these resources to their customers on demand. In some cases, data center operators allow customer applications running in their data centers and charge for actual resource usage. This scheme is called *cloud computing* because the computation is happened in data centers or the network, which is typically drawn as a cloud in various illustrations.

An example of utility computing is Amazon's Elastic Compute Cloud (EC2) [1]. Amazon virtualizes and partitions its servers into VMs and allocates VMs to EC2 customers, who request VMs on-line and pay for them when the VMs are turned on. Each VM comes with a pre-installed OS and a set of applications. EC2 customers can modify the existing

[1]http://aws.amazon.com/ec2/

installation or install new applications to fit their needs. However, currently there are only 5 different virtual hardware configurations (small, large, extra large, high-CPU medium, and high-CPU extra large) and one cannot change the VM configuration on the fly. While the fixed configuration eases resource management, it may also lower resource utilization and limit the flexibility provided by virtualization.

There are also other utility computing services, such as Sun Grid Compute Utility [39] and HP Utility Data Center [68]. Sun Grid Compute Utility provides a billing model similar to Amazon EC2 while HP Utility Data Center mainly works with large companies. Unfortunately, although the development of these services expands the knowledge on resource modeling, allocation, autonomic computing, and other fields, the commercial operations of these services have ceased [51].

Google App Engine [2] is an example cloud computing infrastructure that allows customer applications running on Google's infrastructure. Unlike EC2, Google App Engine users do not need to manage VMs or OSes and can focus on the deployment of their applications. However, the applications must be programmed in Python and use a set of pre-defined application programming interfaces (APIs) to interact with other parts of the system. While its pricing has not been announced, the infrastructure monitors a list of metrics, including the number of (HTTP) requests served, CPU time, network bandwidth, the number of requests and the amount of data transfer to a data-store, the number of accesses to supporting services (*e.g.*, mail, URL-fetching, image manipulation), and so on. The actual placement of user applications, load-balancing, security model are all managed by Google and not revealed to the public at present.

Translating high-level operations to actual resource consumption can be difficult and non-intuitive. Therefore, this dissertation assumes a data center allocating resources to its customers because its economy model is easier to understand.

[2]http://code.google.com/appengine/

2.2 Platform Virtualization

While platform virtualization was first introduced to execute multiple OSes concurrently on expensive mainframes, in data centers, it is commonly used to partition resources and consolidate multiple servers on a fewer number of host machines to improve physical resource utilization. There are a number of different approaches to realizing system virtualization. In this section, we first briefly describe these approaches and then discuss two of the applications of virtualization, namely resource management and fault-tolerance, in data centers.

2.2.1 Classification

Many different types of platform virtualization have been implemented. They differ in VM abstraction and the layer of virtualization. Full virtualization creates an abstraction that has the full instruction set as the underlying hardware. Virtual machine monitors (VMMs), also called hypervisors, may employ a binary translation or a hardware-assisted method to achieve this goal. Some VMMs run on bare-metal directly (called Type 1 VMM) and some other VMMs are hosted inside an OS (called Type 2 VMM). Notable full virtualization VMMs include VMware, Microsoft Hyper-V, and QEMU.

The OS within a fully-virtualized VM, called a *guest OS*, does not require any modification as the VM has the same architecture as the underlying hardware. However, it incurs some overhead when privileged instructions are executed. Para-virtualization is another type of platform virtualization, where guest OSes are modified before being loaded in a VM. Privileged instructions in a guest OS are replaced by system calls to the VMM, called *hypercalls*, and consume less resources. Xen is the hypervisor that is best-known for using this approach. Modified Linux kernels and other OS kernels are provided with Xen. Newer version of Xen can also use hardware assistance and works in full virtualization mode.

Virtualization can also be realized in OSes, called *OS-level virtualization*. Instead of virtualizing the platform, the OS system-call interface is virtualized. One instance of OS kernel is shared by many VMs, and each has its own root file system, user ID space, process ID space, routing table, and so on. When a system call is invoked, the OS first determines which VM the caller process is within and then processes the system call accordingly. Clearly, one cannot mix VMs of different OSes in OS-level virtualization but it also incurs the least run-time overhead. Examples of OS-level virtualization include Virtuozzo/OpenVZ, Linux-VServer, and Solaris Zones.

All of these types of virtualization can be seen in data centers. The performance of different virtualization approaches have been compared [11, 71]. In this dissertation we adopt Xen and a modified Linux kernel as the virtualization platform because of their cleaner resource abstraction. However, our design is not VMM-specific and our implementation can be adapted to other types of VMM as well.

2.2.2 Resource Management

In a utility data center, VMs can be used as resource containers where hypervisors are responsible for enforcing resource allocations and monitoring actual resource utilization [21]. CPU is usually time-shared between VMs and there are many CPU schedulers that can allocate CPU cycles to VMs [1, 31]. Main memory is typically space-shared by VMs, but some VMMs, such as VMware, may swap out memory pages and allow memory overbooking. While it is easy to allocate memory to VMs, finding the actual needs is usually a guessing game. VMware ESX Server employs page sharing to reduce the actual memory footprints of VMs and utilizes a sampling approach to estimating the working set size [75]. Idle memory pages are reclaimed and reallocated to VMs that can make use of them. Geiger [49] assumes that the guest OS uses an unified buffer cache to manage its memory. By monitoring file system traffic, the working set size can be inferred even when the VM is thrashing. Disk and network I/O can also be regulated by I/O schedulers and network traffic shapers [41].

Monitoring and diagnostic tools have also been introduced to virtualization [43, 55].

While the CPU virtualization overhead is low in para-virtualization and OS-level virtualization, software-virtualized I/O operations have been identified as the bottleneck in hardware- and para-virtualizations [11, 55]. The high cost of I/O operations may be reduced with emerging hardware-assisted I/O virtualization, such as Intel's Virtualization Technology for Directed I/O (VT-d) or AMD's IOMMU. When physical resources become the bottleneck, one can migrate a VM from one physical host to another. Major VMMs also support live migrations, where VMs are only suspended for a very short while during migration [65].

2.2.3 Fault Tolerance

VM fault-tolerance techniques can be classified as either guest-OS-level or VMM-level approaches. Guest-OS-level products, such as Microsoft Cluster Service and Veritas Cluster Server, manage service availability by monitoring the applications and restarting failed VMs. Our approach is tailored for Web server clusters and integrates front-end dispatching algorithms and flexible resource allocation to improve performance during degraded operation.

VMM-level approaches, such as VMware HA (High-Availability) and everRun VM from Marathon Technologies, synchronize the execution of multiple VMs and tolerate physical server failures. These approaches are agnostic of guest OSes and may have a shorter recovery time. However, both the overhead in normal execution and the required amount of resource are also higher. Typically, web servers do not require the highest single-server availability because the integrity of higher-level semantics (like processing credit cards) are usually provided by other approaches.

2.3 Web Server Clusters

Since the emergence of the World Wide Web, there has been extensive research into the scalability and availability of web server clusters [29, 36]. The architecture of a web cluster can be classified by its request-dispatching and packet-routing algorithms. The survey paper [14] provides a detailed taxonomy and summarizes previous studies.

The goals of load-balancing algorithms include, but not limit to, increasing the throughput of the cluster and lowering the average response time. Moreover, most algorithms assume that the amount of computing resources of each back-end server is fixed during execution. Algorithms such as round-robin, least-connections, and their weighted counterparts are simple and scalable. However, they are not locality-aware, so the total throughput does not scale with the cluster size.

One can also use the URL hashing to select a server in a locality-aware fashion. For each incoming request, the URL string is examined and fed to a hash function. According to its hash value, a server is chosen to serve the request. Essentially, the URL namespace is partitioned by the hash function, creating an N-to-1 mapping where each URL is mapped to a server. The mapping is typically fixed unless the configuration of the cluster has changed.

Because each time a request to a certain object can only be dispatched to a certain server, the locality of the request dispatching is greatly improved as compared to non-locality-aware algorithms. If a previous request to a certain object already loads and caches the object in the memory, successive requests to the same object can be served from memory and achieve the goal of reducing response time and increasing throughput. The cache hit ratio is improved and the amount of disk I/O, which is very expensive compared to serving requests from memory, is reduced. Otherwise, if the object has nevered been encountered before, or if the object has been evicted from memory, the object will be loaded from disk or other storage device. Since no other servers could have the same object in memory, loading the object from disks is inevitable.

Since the hash function is stateless, URL-hashing itself is highly scalable—multiple front-ends using the same hash function can be deployed independently. However, URL-hashing itself does not have any replication capability nor load-balancing capability. Therefore, when a failure occurs, existing locality might be lost due to an inferior hash function.

Moreover, when the hash function does not distribute the workload evenly across active servers, some servers may be overloaded while others are not fully utilized.

Some locality-aware algorithms also balance the workload across active servers. LARD, locality-aware request distribution, improves locality by keeping a request-dispatching history [6]. Similar to URL-hashing, LARD improves the cache hit ratio in back-end servers by dispatching a request to the server that might have served a request of the same object before. In order to know which server has served a certain object before, LARD keeps a dispatching history in the front-end load-balancer. For each incoming request, a history look-up operation is performed to see whether the object has been served. If so, the previous dispatching destination is used again. If the object has not been seen before, the currently least-loaded server is chosen.

Unlike URL-hashing, the mapping between an object to a back-end server may change during the operation. In order to achieve load-balancing among back-end servers, LARD monitors the load of each back-end server in terms of the number of active connections. If the load of a server exceeds a certain threshold, the server is considered over-loaded and incoming requests toward that server will be re-dispatched and bound to the currently least-loaded server unless all servers are over-loaded. If a higher threshold is also violated at a certain server, incoming requests toward that server will be re-dispatched to the currently least-loaded server no matter what. By changing the mapping between objects to servers, LARD sheds load from over-loaded servers and moves it to other servers, achieving load-balancing.

LARD/R, a variant with replication capability, associates multiple servers with an object, and the least-loaded one is chosen as the dispatch destination. The set of servers for an object increases when even the least-loaded one in the set is considered over-loaded. On the other hand, a server is removed from the set if the current set of multiple servers does not need more, or has not been recently reduced for a certainperiod of time. This scheme can distribute the requests of one single hot object to multiple servers. It also improves fault-tolerance because an object may still be served from memory if multiple servers are associated with it and one vanishes due to server failure.

As the size of the cluster increases, LARD and LARD/R are more scalable than earlier designs. Its later developments improved scalability further by supporting persistent connections and separating network-layer connection distribution from request dispatching [8]. However, the request-dispatching history, which every front-end device must look up and update frequently, is still a centralized component. As the request rate increases, the frequent operations on the history may become the bottleneck in the system. Moreover, load-balancing in LARD depends on appropriate setting of thresholds. Setting the thresholds too high results in poor load-balancing. On the other hand, LARD degenerates into the regular least-connection algorithm if the thresholds are set too low. One may also improve the cache-hit ratio by employing cooperative caching middleware [13]. A simulation-based study [25] shows a comparable performance to LARD, but it also required a centralized component.

Hash-based load-balancing techniques have also been used in a cluster of proxy servers in local area networks. With adaptable controlled replication (ACR) [78], each proxy server is responsible for a partition of URL space, which is divided by a hash function. In addition, some space can be used for caching any objects. When a client request arrives at a proxy server, the server first checks its local cache and forward the request to the responsible server. In this design, frequently-accessed objects can be cached in all proxy servers, which may not be necessary. In addition, the space for caching any object must be large enough to be effective.

There are also algorithms that adjust the size of a cluster or use request-batching in order to improve energy efficiency [19, 33, 60, 74]. In these cases, each server's capacity is proportional to its CPU clock frequency and the number of servers in the cluster may vary with the workload. These methods use simple load-balancing algorithms in their design. When multiple services are hosted on a cluster, a resource-allocation mechanism has also been used to manage server performance [7]. Virtualization provides an alternative approach to partition and manage computing resource and can be used to improve some of these earlier designs.

2.4 Consistent Hashing

Simple hash-based designs do not handle the addition or the removal of servers well. A large number of requests can be reassigned to different servers, and the benefit of locality is lost. Consistent hashing is a technique that minimizes the change of the mapping with respect to server arrivals and departures. The domain of hash values is divided into partitions. Partitions may be of different sizes. When a new server is added, one partition is divided into two. Similarly, when a server departed, its partition is merged with either one of its two adjacent neighbors [50].

Using consistent hashing, one deals with the frequent arrival and departure of nodes in a peer-to-peer network. Both Chord and Pastry use consistent hashing to map keys to peer-to-peer nodes [63, 72]. While they adopt different message routing protocols, the look-up operation completes in predictable time in both networks. Other similar designs include Tapestry and CAN [61, 82]. On top of these networks, several applications, such as cooperative persistent storage and event notification systems, have been built and evaluated [15, 26, 30, 62].

Peer-to-peer networks generally have a much larger number of nodes than server clusters. In addition, server clusters usually do not change their size frequently. While a good hash function spreads keys evenly over all nodes, hot spots are usually not a concern in peer-to-peer networks. For example, frequent look-up of a hot key may overload the node that stores its value and results in a longer response time or even partial service outage. If the number of hot keys is smaller than the number of nodes, load imbalance may occur.

Amazon also uses consistent hashing to create a key-value storage, Dynamo, but all its nodes are owned by one organization, Amazon [28]. New servers can be deployed on demand to handle extra workloads. Its replication scheme spreads the load to a fixed number of nodes. Furthermore, load-balancing is improved by using virtual nodes (tokens) as another layer of indirection. A node can shed its workload by transferring some of its tokens and the associated data to another node. Our work differs from Dynamo in that we use the resource-allocation flexibility in the infrastructure to improve resource utilization. As we allow a larger extent of workload imbalance between nodes, our hashing design becomes simpler.

2.5 Distributed Parallel Computing

As the amount of data and the complexity of algorithms increases, multiple computers are frequently used in parallel to solve a single problem in shorter time. When running a distributed computing program, each compute node may run the same or different programs and they may communicate to each other while working toward a result. Depending on the nature of the problem to solve, one can use data and task parallelism to speed up a program.

2.5.1 Data Parallelism

When the data to be processed can be partitioned into large chunks and each chunk can be handled independently of each other, one can exploit data parallelism to achieve parallel processing. Each compute node can handle one chunk independently and the results, which are usually much smaller than the chunk size, are combined in a later stage. There are several systems that assist large-scale data parallelism program development such as Map-Reduce [27], Hadoop [3], and Dryad [48].

Since each data chunk can be processed independently, one can simply assign each idle compute node a chunk of data until all data are processed. Each node runs as fast as pos-

[3] http://hadoop.apache.org/

sible to process the data. Except in the final stage where all chunks have been dispatched, the workload is balanced in a compute cluster.

2.5.2 Task Parallelism

If data parallelism cannot be easily realized, one may divide the problem by its functions into several programs and run each program on a separate compute node. These programs may communicate frequently during their execution. For example, the data set of computer simulation of fluid dynamics or fast Fourier transforms cannot be divided into disjoint subsets, so these problems are usually implemented with a message passing library, such as the Message Passing Interface (MPI).

Since the execution of one compute node may depend on the data generated at another node, a node may become idle if the data that it depends on are not available. Moreover, each program may require a different amount of resource. The slowest node in a cluster can easily become the bottleneck of the system. Generally, it is much more difficult to place balanced workload and fully utilize all resources in a cluster in MPI programs. Our work is for the task parallelism programs that communicate and synchronize frequently during their execution. Virtualization and consolidation make idle cycles available to other VMs on the same physical host. DPS adapts the CPU cycle allocation so parallel programs can finish faster.

2.6 Cluster Scheduling

Improving the utilization of multiprocessor systems has been investigated extensively. In 1982, Ousterhout first introduced *coscheduling* where busy waiting was suggested instead of blocking at synchronization [57]. It was then called *gang scheduling* among researchers and the performance was analyzed by Feitelson [35]. It has been shown that scheduling multiple processes (or threads within a process) in a gang with busy waiting can reduce program response time for fine-grain synchronization activities. On the other hand, for coarse-grain synchronization processes, or for processes that have unbalanced workload between synchronizations, coordinated scheduling is not beneficial. Feitelson also proposed space-sharing packing schemes to improve utilization, where process migration between processors was also used to reduce fragmentation [34]. More recently, gang scheduling is also used in clusters as well [37, 47].

However, gang scheduling is not scalable, especially in a loosely-coupled environment where communication and synchronization costs are high. Instead, multiple implicit scheduling approaches are proposed to synchronize the execution of a parallel program across local schedulers using activities such as message send and receive, which imply synchronization [2, 22, 32, 56, 70]. These approaches use a combination of strategies to determine what to do while waiting for messages and when messages have been received. Optimizations are also being done in lower levels. By prudently selecting processes to co-schedule on a symmetric multiprocessor system or within a simultaneous multi-threading processor, contention on memory bandwidth or functional units can be avoided, and thus, utilization can be improved [4, 69].

Proportional-share schedulers, such as borrowed virtual-time (BVT) scheduling, lottery scheduling, and stride scheduling, are used to provide fairness in a resource-competing environment [31, 76, 77]. While proportional-share scheduling has also been applied to a network of workstations with implicit scheduling [9, 10], its use was limited to support fairness across a cluster. Tickets are not dynamically transferred at run-time. Tickets for multiple resources are discussed and exchanging resources with competing process is proposed [73].

Scheduling VMs is a much less explored issue, although it has a great deal of similarity to normal CPU scheduling. VSched tackled this problem by utilizing a real-time process scheduler and a controlling daemon tweaking scheduler parameters to schedule VMs running as processes in Linux [54]. While they applied VSched to parallel programs, the result showed that the actual computation rate is very sensitive to scheduling parameter choices (slice/period) and the real-time scheduler can provide performance isolation to VMs. We extend the concept by dynamically transferring CPU shares among peer VMs and also migrating VMs to further reduce program response time and cluster utilization.

2.7 On-line Game Servers

The game server workload and its traffic characteristics have been detailed in various studies [17, 18, 24, 52]. The online games under these studies range from first-person shooter (FPS) games, role-playing games (RPG), and massively multiplayer RPGs (MMORPG). Online gaming traffic patterns contain bursts of small packets with predictable long-term rates. The game server workload also exhibits the time-of-day and day-of-week effects. There are also several studies on the effect of loss, latency and jitter on user experience in FPS, racing, and a highly-interactive games [12, 45, 59, 79]. Both subjective and objective metrics have been used in these studies. They found that while a $<5\%$ packet loss and the existence of jitter have no impact on the quality of game play, a latency as low as 100ms can significantly degrade precision shooting performance in FPS games. For MMORPGs, the effect of latency is not as stringent as in FPS games. A game can still run smoothly with a 1250ms latency [38].

Hosting game servers in data centers has been proposed in [64, 66]. A provisioning manager is used to decide how resources should be allocated. They studied the quality of several metrics as workload estimators and concluded that smoothed metrics cannot accurately track server utilization when the session arrival rate is high. Although several CPU and network resource utilization-based metrics have been compared to each other, it is unclear whether the game server faces fierce competition from other hosted workload and its implication on responsiveness. Our work uses virtualization to consolidate game servers and monitors the time it takes to deliver user input packets to game servers.

Chapter 3

Vibra: Virtualization-Based Resource Allocation for High-Availability Web Server Clusters

3.1 Introduction

World Wide Web is undoubtedly the most important application of the Internet. Individuals increasingly rely on it for their daily lives and companies conduct various types of business with it. After nearly 20 years of development, web server clusters have become the standard means for deploying scalable, high-availability web services. A web server cluster usually consists of a number of servers and a dispatcher of incoming requests. Typically, each web server is hosted on a separate physical server and a dispatching mechanism is designed to reduce the average response time, improve the throughput, and ensure the availability of the web service. Frequently, web server clusters are hosted in large data centers, where redundant power facilities and network connections are provided for uninterrupted operation and service.

By hosting web servers on physical servers, one may easily control the amount of resources that a service can utilize. However, the resources allocated to, but left unused by the service cannot be easily re-allocated for other purposes, thereby resulting in low resource utilization, high costs, and energy waste. As virtualization provides the flexibility of resource allocation that physical servers lack, it is becoming popular to deploy servers in virtual machines (VMs).

Instead of allocating physical servers to customers, data center operators can virtualize the servers and allocate virtual resources to the customers. Unlike physical resources, virtual resources like CPU, memory, network and disk bandwidths, can be divided into smaller units and dynamically (re)allocated to VMs. When one owns multiple VMs in a data center, virtual resources can also be transferred between his VMs. In other words, resources can be allocated in a hierarchical fashion: virtual resources are allocated to a customer who then creates VMs and assigns the virtual resources to them. We call this scheme *VIrtualization-Based Resource Allocation* (Vibra).

A *VM server cluster*—where each server is a VM and shares a pool of resources with other servers in the cluster—is akin to a typical physical server cluster (*e.g.*, both use redundancy for fault-tolerance and load-balancing). Each physical server has a fixed amount of resources, whereas the resource allocated to a VM can be adjusted dynamically at run-time. With the new flexibility empowered by virtualization, workload-distribution algorithms can be re-designed for VM server clusters. Specifically, we re-design a web server cluster, called a *Vibra Web*, to make existing systems utilize resources better and more scalable with virtualization.

Request-dispatching algorithms can be classified as *content-unaware* or *content-aware*. Content-unaware algorithms, such as round-robin and least-connection, are scalable but less efficient. For example, a popular file may be cached in every server, taking up extra memory space. On the other hand, content-aware algorithms, such as LARD [58] and HACC [81], improve resource efficiency and cluster throughput. However, they use a *centralized* request dispatcher in order to balance the workload among servers within a cluster while keeping track of locality. Such a request dispatcher is not scalable and can become a performance and reliability bottleneck within the cluster.

In Vibra Web, two key functions of a content-aware request dispatching algorithm—namely, improving locality and load-balancing—are cleanly decoupled from each other. In the front-end, a hash function applying to the URL in each request is used to partition the workload into disjoint subsets. This approach is called *URL-Hashing* in some load-balancers, albeit the actual hash functions used by commercial products are usually obscured on public documents. We use a very simple algorithm—the sum of the character code of the URL modulo the number of back-end servers—as the hash function. We call this algorithm *Charsum*. Each subset is only dispatched to one or a small number of servers to improve the locality. Since the hash function is stateless, the URL-Hashing scheme is scalable because multiple independent front-ends can be deployed simultaneously.

While URL-hashing is scalable, each subset may impose a different resource demand on each server. Instead of balancing the workload of subsets, we balance the resource utilization of servers. A resource-allocation agent is used to monitor the resource usage of each VM and signal resource transfers to the data center operator. The goal is to keep the resources utilized as evenly as possible. While the agent is also a centralized component, it is more scalable due to much fewer states and much less computation it requires. Making the agent fault-tolerant is also much easier than LARD for the same reasons.

The performance of Vibra Web is evaluated by using the trace from the World Cup'98 web site and a synthetic YouTube-like workload generator. Vibra Web is compared against other request-distribution policies, including round-robin, least-connection, LARD, and LARD/R. With the URL-hashing front-end in conjunction with the proposed resource-allocation agent, the average response time is reduced by up to 51% compared to LARD. Alternatively, the same service quality can be achieved with 20% less bandwidth than LARD. The throughput of the cluster is also as good as LARD when there is no hot-spot in the cluster and 32% faster than LARD when the concurrency level is high. The front-end also uses much less CPU and memory resources. Other than a 0.03μs overhead of calculating hash values, the front-end does not use more resources than simple dispatching algorithms.

This chapter is organized as follows. We describe the architecture of a VM-hosting data center and Vibra Web in Section 3.2. The URL-hashing scheme and the resource-allocation

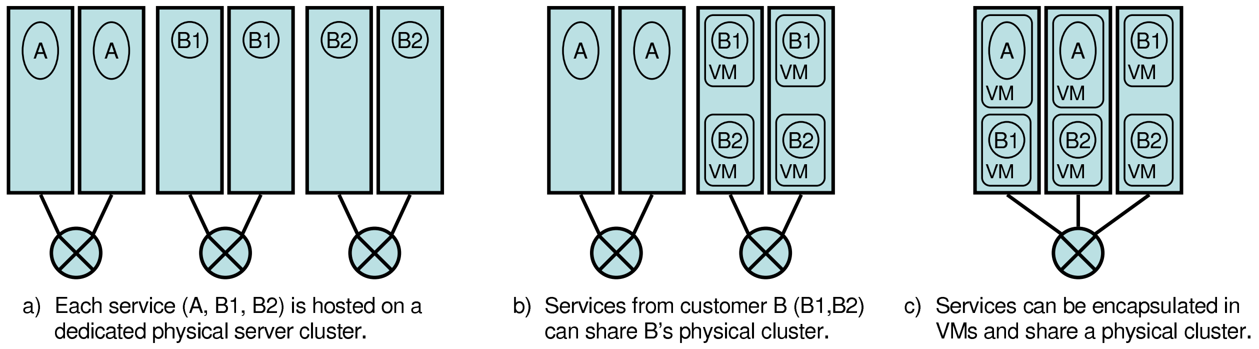

A
A
B1
B1
B2
B2
a) Each service (A, B1, B2) is hosted on a dedicated physical server cluster.
A
A
B1 VM
B1 VM
B2 VM
B2 VM
b) Services from customer B (B1,B2) can share B's physical cluster.
A VM
A VM
B1 VM
B1 VM
B2 VM
B2 VM
c) Services can be encapsulated in VMs and share a physical cluster.

Figure 3.1 Comparison of a physical server cluster, customer-initiated virtualization, and data center-managed virtualization.

agent are described in Section 3.3. Our implementation of Vibra Web is detailed in Section 3.4 and the experimental evaluation results are presented in Section 3.5. We discuss an alternative approach and the limitation of Vibra in Section 3.6 before summarizing this chapter in Section 3.7.

3.2 Vibra VM Hosting

Clustering is commonly used for Internet servers to provide scalable and high-availability services. This section first presents the model of a traditional physical server cluster and then discusses the rationale behind the hierarchical allocation of virtual resources in large data centers.

3.2.1 Physical Server Clusters

Using a physical server cluster for each Internet service is the simplest way for a data center operator to make resource-allocation guarantees for their customers. A number of physical servers and a front-end request distribution device are allocated for each service. The operator may also impose an egress network bandwidth limit for each cluster. The model of a physical server cluster is depicted in Figure 3.1(a).

The number of physical servers in a cluster is usually determined based on the estimation of peak demand. In addition to this peak-load-based cluster sizing, the server redundancy requirement for fault-tolerance makes the server cluster utilization usually very low, especially for small services. In an extreme case where the workload can be handled by just one server, this redundancy requires the addition of another identical server, resulting in a 50% reduction of server utilization. While sharing physical servers between clusters for handling peak demand and meeting the redundancy requirements (albeit to a less extent) can improve utilization, these setups are rarely used due to long re-targeting time and complicated (re)configuration. As a result, excessive resources cannot be easily utilized.

3.2.2 Virtualization in Data Centers

Server virtualization allows resource-allocation guarantees while providing security boundaries and performance isolation between VMs. Each VM is assigned a fraction of CPU, a portion of main memory, a virtual storage, virtual network interfaces, and so on. Some virtualized systems also allow resource-allocation adjustments without power-cycling VMs. Data center customers running multiple services can virtualize their servers and deploy VMs of different services on top of a physical server cluster. We call this scheme *customer-initiated virtualization* (Figure 3.1(b)).

In addition to sharing hardware resources, a cluster's egress network bandwidth is also shared among hosted services, which can be regulated by shaping traffic in either VMs or VM monitor (VMM). With proper VM sizing and placement, the redundancy can be preserved at a lower cost while benefits on utilization are limited by the number of services and the demand of each service.

Alternatively, data center operators can virtualize physical servers *a priori* and allocate VMs to customers. Essentially, the VMM becomes an extension of the data center's infrastructure and manages physical resources on the physical server. We call this scheme *data center-managed virtualization* (see Figure 3.1(c)). As long as each VM is placed on a different physical server, the redundancy level for a VM cluster is the same as that for a physical server cluster. In this scheme, excessive resources can be returned to the data center if customers are given incentives to do so. When the resource demand increases, customers can also ask for more resources, which can be allocated to, and used by VMs. If a physical server cannot meet the resource demand, live VM migration can be used to change VM placements inside a data center and create more headroom without service interruption.

3.2.3 Vibra VM Clusters

When a customer owns multiple VMs in a data center, virtual resources can also be transferred between his VMs. VMs that execute his service(s) can share a common pool of resources, and the resources in a data center are allocated in a hierarchical manner: the data center operator first allocates an aggregate amount of resource to a customer which is then distributed to his VMs. In other words, when acquiring computing resources from a data center, a customer only needs to specify the total amount of resource and the number of VMs that he needs for his application(s). Since this allocation is based on virtualized resources, we call it *VIrtualization-Based Resource Allocation* (Vibra).

In Vibra, the aggregated amount of resources allocated to a customer or a service is fixed. Compared to a service-level-oriented contract, the data center operator can easily make a commitment to a resource-allocation-contract. On the other hand, a customer still has the flexibility to distribute resources according to his needs. We use this flexibility to design a web server cluster in the next section.

This scheme also frees customers from packing multiple VMs into fewer physical hosts to reduce cost. Moreover, if the demand grows beyond a cluster's capacity, one can merely increase the resource pool and allocate the newly-acquired resources without incurring any downtime. This scheme can also remove the need for over-provisioning servers.

Although the insertion of a VMM may incur some overhead of running servers, its advantages—namely, a secure and flexible way of resource allocation, and decoupling server system design from resource provisioning—can better utilize data center resources and better control server performance.

3.3 Vibra-Enabled Web Cluster

We use the Vibra hosting model to build a scalable and resource-efficient web server cluster, *Vibra Web*. Today's web server workload is mainly composed of two parts: *CPU-intensive* dynamic content creation and *bandwidth-intensive* static content service. In this chapter, we build a web cluster that serves static content to exemplify the use of Vibra on a typical load-balancing application. We focus on two resource types: network bandwidth and memory. CPU utilization is not considered here because it is usually not a bottleneck when static files are served.

The design of a content-aware cluster can be simplified by using Vibra. Instead of moving the workload among servers for load-balancing, Vibra transfers resource allocations among VMs and balances the utilization of VMs. Vibra Web composed of two main components: (1) URL-hashing, a stateless and scalable request dispatcher, and (2) a resource-allocation agent. The design of the two components are detailed next.

3.3.1 URL-Hashing Dispatching of Requests

The goal of the locality-aware request dispatcher is simply redirecting requests for high locality affinity. Imbalance in VM workloads is handled by the resource-allocation agent which moves virtual resources among VMs.

Due to the usually enormous number of requests that a popular web site has to process, the dispatching algorithm must be scalable. This requirement is similar to that of CPU cache design, where the cache look-up algorithm needs to scale with large cache size and high clock frequency. *Stateful* algorithms—whether they keep track of currently-cached

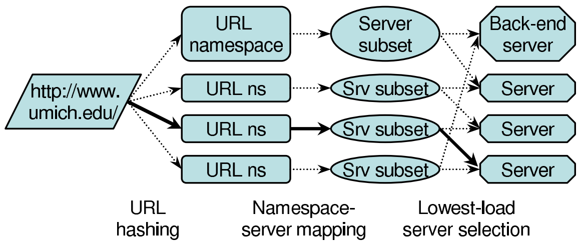


Figure 3.2 The architecture of a URL-hashing web server cluster. The mapping between different layers is shown in dotted lines and an example is drawn with bold lines.

objects or previous dispatching history—are unsuitable for handling an excessive number of requests to popular web sites.

Our design consists of three steps as depicted in Figure 3.2. It first divides the namespace of incoming requests into multiple mutually-exclusive sub-namespaces. Each sub-namespace is then mapped to a subset of back-end VMs, where the least-loaded VM is selected to serve an incoming request. Cache affinity is improved because requests of the same sub-namespace are always handled by one subset of back-end VMs.

In case of HTTP requests, hashing the URL string is an effective and scalable way of dividing the namespace. A low-cost hash function is preferred as the imbalance caused by an uneven hash value distribution can be compensated by adaptive resource allocation/migration. Assuming that the cluster consists of N VMs ($VM_1,\ldots,VM_N$), we use a hash function that returns N different hash values, and therefore, creates N URL sub-namespaces.

After a URL string has been sorted into a sub-namespace, there is a 1-to-1 correspondence between a sub-namespace and a subset of back-end VMs. In order to improve scalability, we make all server subsets contain an identical number of VMs and have a fixed mapping. We choose a simple, stateless function to create server subsets. Assuming there are R VMs in subset i ($i = 1,\ldots,N$), we simply place VM_i and the next $R-1$ servers into subset i (with wrap-around). We call R the *replication factor*. The dispatcher then simply looks at the load of these R VMs and choose the least-loaded to assign an incoming request. The number of active connections is used to estimate a VM's load.

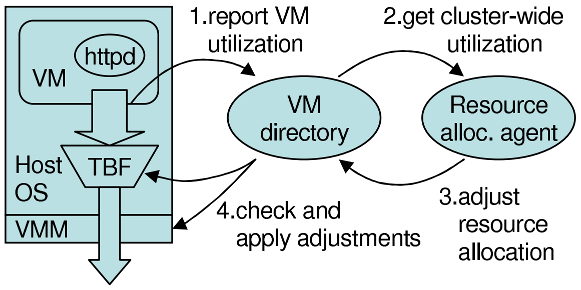

1.report VM utilization
2.get cluster-wide utilization
httpd
VM
VM directory
Resource alloc. agent
Host OS
TBF
VMM
4.check and apply adjustments
3.adjust resource allocation

Figure 3.3 Dynamic resource allocation using an agent with the help of a data center-managed VM directory.

The purpose of having multiple VMs to serve a URL sub-namespace is two-fold. First, it creates another layer of load-balancing for each sub-namespace. Second, it also facilitates failure recovery because a frequently-accessed object can be cached in more than one VM. If $R = 1$, while the cache affinity is the highest, the cluster solely depends on Vibra to balance resource usage. Also, note that when $R = N$, this architecture reduces to the least-connection dispatching algorithm and loses all benefits from locality. This design is akin to the associativity in CPU cache design.

3.3.2 Resource-Allocation Agent

Having a small replication factor inevitably introduces some imbalance in back-end resource demands. For example, a popular video file in YouTube can generate much more traffic than other larger but less popular files, which may even force more frequently-accessed files to be paged out. We would like to re-distribute resources in order to balance the utilization of back-end VMs.

For each type of resource, a resource-usage monitor periodically reports the current demand level. The resource-usage monitor can reside either at the VMM or within each VM. Depending on the monitoring approach used, different sample frequencies and a smoothing function can be used for a different type of resource. The reason for applying a smoothing function is to obtain a more stable reading instead of transient demands.

Combined with the current resource allocation, the agent then determines a new allocation and then informs VMMs to take appropriate actions (Figure 3.3). We use network bandwidth allocation as an example. Assuming that a web cluster consists of N VMs and has been allocated a total bandwidth of BW Mbps, the customer can arbitrarily allocate $bw_i > 0$ Mbps out of the total bandwidth to $VM_i (i = 1, \ldots N,)$, as long as $\sum_{i=1}^{N} bw_i \leq BW$.

Assuming that the recent resource demand can be used to predict resource demand in the near future, we use a simple policy that aims to achieve an equal bandwidth utilization across all VMs. In other words, excessive bandwidth, if any, is distributed to VMs proportionally to their current usage. If the current bandwidth utilization of each VM is u_i Mbps ($i = 1, \ldots, N$), we estimate current bandwidth demand d'_i by applying an exponential moving average to previously-estimated bandwidth demand d_i as:

$$d'_i = \alpha \cdot d_i + (1 - \alpha) u_i. \tag{3.1}$$

The new bandwidth bw'_i allocated to VM_i is then calculated as:

$$bw'_i = \frac{d'_i}{\sum_{i=1}^{N} d'_i / BW}. \tag{3.2}$$

When making changes, the agent first reduces the bandwidth allocation from the over-provisioned VMs, and then increases the bandwidth allocation for the rest of VMs. This will always keep the total bandwidth within the entitled amount.

If the physical host failed to provide enough resources, the resource-allocation agent can ask the VMM to migrate the VM to another physical host or instruct the VM owner (the customer) to use a web cluster to replace the bottleneck server. Unlike mere addition of servers to the cluster, this method does not affect the files cached in other servers at the cost of a second URL-hashing computation and request distribution.

This proportional resource-allocation policy is suitable for typical server clusters, where all servers are of the same value. If a customer provides different classes of services to end-users, resource distribution can be skewed toward high-valued services as well.

3.4 Implementation

We implemented Vibra using Xen [11] along with a custom-made VM directory service, which keeps track of cluster-wide VM placements and resource allocations. We have also built an application-layer HTTP request dispatcher and a bandwidth-allocation agent to realize Vibra Web. The implementation details are discussed next.

3.4.1 Xen VM Hosting

There are many different VMMs available for x86 hardware. We chose Xen (version 3.0.2) for its physical resource abstraction, low overhead, and open source. However, the concept of Vibra can also be applied to any other VMM.

Xen-modified Linux kernel 2.6.16 is used as both the host OS (domain 0) and the guest OS. The size of main memory allocated to each VM can also be changed on-the-fly, although at present the new memory size cannot be larger than the initial allocation at the time of its creation. Virtual Ethernet interfaces (eth*n*) created in VMs are connected to virtual interfaces (vif*n*.*m*, where *n* is a Xen domain ID and *m* is an interface ID) in domain 0, where virtual interfaces are bridged. While we do not manage CPU time allocation in this chapter, Xen also implemented various CPU schedulers.

3.4.2 VM Directory

The VM directory is set up as a centralized repository for (1) customers' resource bounds, (2) VM resource allocation and utilization, (3) physical host capacity and utilization, and (4) VM-to-physical host mapping.

Physical hosts and VMs periodically report their current resource utilization to the VM directory. Agents make allocation decisions based on the information available in the directory. An agent manipulates resource allocations by modifying the corresponding entries in the VM directory. The directory validates each change by checking the customer's resource bound and physical host utilization. It then looks up the VM location for the VM-to-physical mapping and sends out the allocation request to the VMM on behalf of the agent. Although not discussed in this chapter, a data center can also use the directory to identify physical hosts that become bottlenecks or adjust VM placements to save power. VM migration can then be initiated to change the VM placements.

3.4.3 URL-Hashing Request Dispatcher

The URL-hashing HTTP request dispatcher is built as a user-space process. It listens on a TCP port and waits for incoming connection requests. For each HTTP request, it reads the header and parses the URL. Based on the object name (not including the search part, i.e., anything after the question mark '?' in a URL), a hash value is calculated. We use a very simple hash function—the sum of the character code values of the URL modulo the number of back-end servers—to sort a URL into one of the sub-namespaces and its corresponding VM subset, where the least-loaded server is chosen as the dispatching destination. We call the simple algorithm *Charsum* and also incorporate the commonly used MD5 and SHA-1 algorithms [1] for comparison. The dispatcher implements HTTP/1.1 and maintains persistent connections to back-end VMs. If all connections to the destination VM are in use, it will create a new connection before relaying the request and the response between the two ends. Otherwise, it simply reuses existing connections. It also keeps the client-side connections open as defined in HTTP/1.1 and each HTTP request in a TCP connection can be dispatched to a different server.

We use the `epoll` I/O event notification facility in modern Linux kernels for its high scalability. Higher performance can be achieved by porting the dispatcher into kernel and using TCP splicing or experimental TCP connection hand-off protocols [6, 46, 58]. A detailed analysis of these techniques can be found in [14]. For the purpose of comparison, we also implemented round-robin, least-connection, LARD and LARD/R in the same

[1]We use the MD5 and SHA-1 implementation in LGPL Crypto library.

program.

3.4.4 Traffic Classification and Policing

As mentioned in Section 3.2, besides CPU, memory, and other resources, the VMM also needs to manage the network bandwidth of each VM. We use the `netfilter` (a.k.a. `iptables`) to implement traffic classification and policing.

We first classify network traffic as egress or intra-datacenter (internal) traffic. We use the source IP address to identify the sending VM and the destination address to classify traffic. Once packets are classified, they are placed into different queues created by the `tc` (traffic control) command. Each class of traffic is put into a different token bucket to limit its bandwidth utilization.

3.4.5 Bandwidth-Usage Measurement

Ideally, the agent would like to know the actual resource demand of each VM with some degree of accuracy. Unfortunately, for example, a transport layer protocol such as TCP can adapt its transmission rate to the available bandwidth and, therefore, conceal the real need. One possible method to estimate the actual demand is to look into the per-connection TCP transmission queue and the NIC buffer in the guest OS. Although this approach might give us a better estimate of bandwidth demand of an existing traffic source, it is not always possible to instrument in a guest OS to gather such information. We therefore use the egress traffic matching rule at the VMM and take the packet-enqueuing rate as the current bandwidth demand. Intranet traffic is not counted because bandwidth within a data center is much cheaper and abundant. The statistics, including the number of packets and bytes that a VM has sent, are polled using the `iptables` command every second and logged in the VM directory, where the exponential moving average function is applied. There is also an API to traverse rules if a lower overhead is desired.

3.4.6 Estimation of Working Set Size

It is non-trivial to measure the working set size. In Linux, the virtual memory system uses two lists, active and inactive page lists, to approximate the least-recently-used (LRU) strategy. While the total number of active pages is supposed to estimate the working set size, in reality it does not—the default virtual memory system tends to keep a large active list and moves active pages to the inactive list only when memory space is needed.

We modify the Linux kernel such that the active list size can be used to estimate the working set size. In a static-content web server workload, we expect that most memory pages, and also most memory pages in the active list, are used to store web objects. These files are only mapped into a virtual address space when they are being accessed. On the other hand, web server programs are mapped into one or more virtual address spaces for most of the time.

Similar to the `/proc/sys/vm/drop_caches` function, we created a function, named `deactivate_caches`, that moves active but non-mapped memory pages into the inactive list. We also marked those pages as "referenced" so that the next time they are accessed they will be moved back into the active list (otherwise, they will only be marked as "referenced"). At that time, all memory pages in the active list must be mapped into some virtual address space. In addition, the "swappiness" factor (`/proc/sys/vm-/swappiness`) is set to 0 to slow down the process of moving pages from the active list to the inactive list.

We then wait for a short period of time. The size of active list increases as cached blocks are referenced and used as an indicator of the working set size. If we read the size of active list too soon after deactivating caches, the result might be biased toward a transient behavior. On the other hand, the default active list management may kick in if we wait too long. The effect of this wait time and also the overhead of moving pages between active and inactive lists are evaluated in Section 3.5.

3.4.7 Resource-Allocation Agent

We implemented the proportional resource-allocation agent described in Section 3.3.2. The agent runs in one of the customer's VMs and connects to the VM directory for all necessary information. Note that the communication with the VM directory is treated as intranet traffic and hence, is not limited by the egress bandwidth quota.

The frequency of adapting resource allocation depends on how frequently the workload pattern changes. While frequent bandwidth adaptation accelerates the speed of draining the packets queued due to the lack of bandwidth, it may also affect TCP window size adaptation adversely. We tried different adaptation frequencies and the results are reported in Section 3.5.

On the other hand, adapting memory allocation is a much more expensive operation. Newly-added memory pages show up as free space in a VM and wait for uncached objects to be loaded. Removing memory pages usually signals the virtual memory manager in the VM for page-out activities. Although most memory pages are clean in a web server, scanning page lists and modifying page tables are still not cheap. Therefore, memory page transfers should be invoked less frequently. In the next section, we evaluate the overhead of different invocation frequencies.

3.5 Evaluation

We have built a testbed to evaluate the performance of Vibra Web. The VM cluster consists of 4 PCs as host machines, each equipped with a Pentium-4 2.66GHz CPU and 1GB of main memory. Another machine (P4 2.26GHz) acts as the request dispatcher and also hosts the VM directory service. All of them are interconnected via a gigabit Ethernet switch. A second network interface on the request distributor is connected to a machine that generates clients' traffic.

Dispatching algorithm	Round-robin	Charsum Hash	MD5 Hash	SHA-1 Hash
% of requests processed	25/25/25/25	27/27/20/26	29/17/28/26	27/22/24/27
% of total transfer size	25/25/25/25	14/64/10/12	39/25/16/19	45/15/25/15
# of distinct objects	2821/2766/2833/2803	1167/1159/1143/1187	1179/1156/1164/1157	1170/1159/1198/1129
Total size of distinct objects	34/34/33/27 MB	15/13/14/14 MB	11/12/16/17 MB	20/13/13/11 MB

Table 3.1 Statistics of each subset of requests or sub-namespaces using different dispatching algorithms.

3.5.1 Trace Analysis

We first use the server log from the 1998 FIFA World Cup web site [5], which is still the most extensive and publicly available web server logs. The workload is mostly composed of static files. We first choose a 10-minute trace (6:10pm-6:20pm GMT+2, June 30, 1998), during which the peak demand of the Romania–Croatia game took place. This specific clip is chosen because, as our trace analysis shows, it represents a worst-case scenario for URL-hashing in real life.

We first statically analyze various aspects of the trace using different request distribution algorithms (round-robin and URL-hashing with Charsum, MD5, and SHA-1 algorithms) to compare different hash functions and justify the need of dynamic resource allocation in a cluster. LARD and the least-connection algorithms are not included here because their results depend on run-time state and actual server configuration. For the same reason, we set the replication factor R to 1 in URL-hashing.

From Table 3.1, we can see that while URL-hashing approaches greatly reduces the number and the total size of distinct objects on each sub-namespace, the requests are not distributed evenly, no matter which hash function is being used. In terms of number of requests in each sub-namespace, SHA-1 distributes the requests most evenly while the sub-namespace 1 in MD5 has 75% more requests than sub-namespace 2. However, in term of total transfer size, the distribution by MD5 is more even than the ones by Charsum and SHA-1. The total transfer size of sub-namespace 2 is more than 6 times greater than the one of sub-namespace 3 when Charsum is used. The ratio is also as much as 2.4 times when MD5 is used. This indicates the need of dynamic resource allocation.

Moreover, the URL-hashing reduces the number and the total size of distinct objects significantly. We note that the total size of distinct objects is most evenly distributed when Charsum is used. As there is no one hash function outperforms others in all measures, and also that the results are likely to be specific to the trace we use, we cannot dismiss any hash function after the analysis.

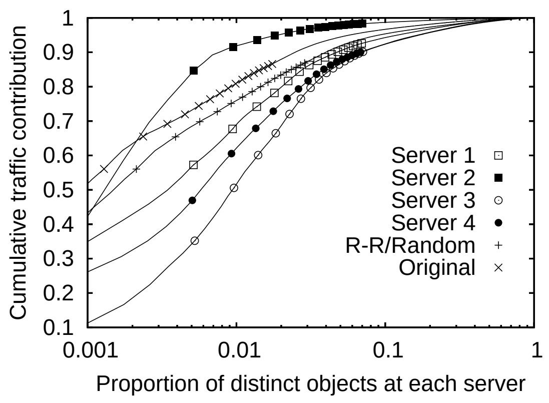


Figure 3.4 The CDF of total throughput by objects sorted by their contributions.

While the average object sizes (not weighted by their access frequency) do not differ significantly after hashing, differences in bandwidth demand suggest the existence of *hot objects* in this trace. We use the Charsum algorithm as an example. For each sub-namespace, we first sort the objects by their contributions to network traffic (number of bytes transferred), and plot a cumulative distribution in Figure 3.4. From the figure, we can see that a very small percentage of files on sub-namespace 2 contribute to most of the server's bandwidth usage. The distribution is different for each sub-namespace, suggesting that the workload model becomes different after hashing. In contrast, if round-robin is used, the shape is the same for all servers and is very similar to the original workload.

A closer examination shows that merely 8 files contribute to 90% of traffic in sub-namespace 2. They are splash screens, the program and statistics (or status) [2] of the ongoing game in both English and French, and two embedded images. In particular, the English version of splash screen contribute to about 37% of traffic in sub-namespace 2, or 24% of total traffic. Coincidentally, two pairs of words, “prog”/“stat” and “english”/“french,” have the same hash value using Charsum, so the four most frequently-accessed files of a game all fall into the same sub-namespace. When SHA-1 and MD5 are used, 4 and 3 out of these 8 URLs are hashed to the same server, respectively.

As different games were being played, the *hot spot* in a workload may also shift from

[2]We can only guess the content from file names.

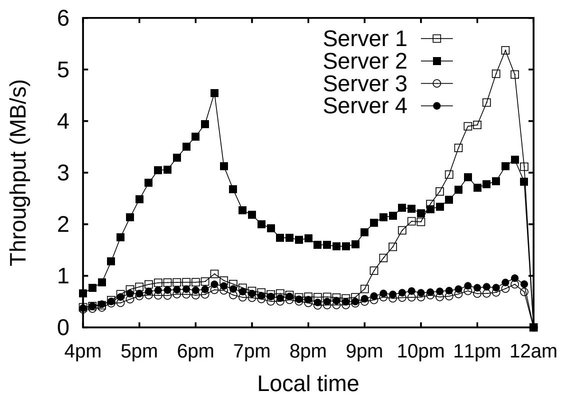


Figure 3.5 The throughput of each sub-namespace using Charsum. The hot spot moves from sub-namespace 2 to sub-namespace 1.

one server to another. This effect can be seen easily with an 8-hour throughput history that covers both games played on June 30, 1998 (Figure 3.5). The Charsum algorithm makes both splash screens and information about the first game on sub-namespace 2, which started at 4:30pm. The second game started at 9pm and its game information is hashed to sub-namespace 1 instead, while the splash screens are still served from sub-namespace 2. This strong evidence confirmed the need for dynamic bandwidth allocation.

3.5.2 Overhead Analysis

In addition, we also compare the cost of different dispatching algorithms. We apply each hash function to the 90,000 URLs in the World Cup trace and measure the average time it takes to hash a URL string and make a dispatching decision. Round-robin serves as the control group here as it simply loads URLs into memory and simply discards them. Although Charsum generates undesirable collisions, it only takes 0.28μs in hashing, or a merely 0.03μs more than round-robin. Comparing to 1.79μs and 2.39μs for MD5 and SHA-1 (or 1.54μs and 2.14μs more than round-robin), Charsum is significantly cheaper. For LARD, we preloaded all 90,000 URLs into its dispatching history and its takes 1.86μs for each look-up.

In addition, the memory overhead of LARD is much more significant. It uses more than

Dispatching algorithm	Round-robin	LARD	Charsum	MD5	SHA-1
Cost per hash/look-up (μs)	0.25	1.86	0.28	1.79	2.39
Increases in memory resident set size (4K memory pages)	6	1413	6	20	21

Table 3.2 The CPU and memory overheads of dispatching algorithms.

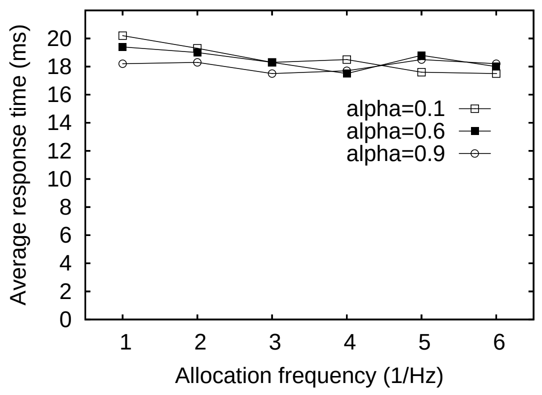


Figure 3.6 The effect of different smoothing factors and the frequency of bandwidth allocation.

5600KB memory to store all 90,000 URLs in main memory, *i.e.*, about 64 bytes for each distinct object. Comparing to round-robin, Charsum does not use any additional memory. MD5 and SHA-1 only use a few more memory pages because of the library we used. In a production system, the number of distinct objects can be several orders-of-magnitude more than that in this test case, resulting in even higher memory and CPU overhead. We argue that a cheaper hash function is favorable as workload imbalance can be handled by dynamic resource allocation. Therefore, for the rest of evaluations, only Charsum algorithm is used in URL-hashing.

3.5.3 Trace-Based Evaluation

Before applying dynamic bandwidth allocation and comparing URL-hashing with other algorithms, we first evaluate the effect of the smoothing factor α in Eq. (3.1) for calculating the bandwidth demand and the frequency of bandwidth allocation. In this evaluation, the dispatcher is running the URL-hashing algorithm with $R = 2$. Each server is allocated 48MB of memory and 2.7MB/s of available bandwidth. These numbers are so chosen that dynamic allocation is necessary to run the workload. Experiments are run repetitively such that the 95% confidence interval is within 5% of mean value. The results are plotted in Figure 3.6.

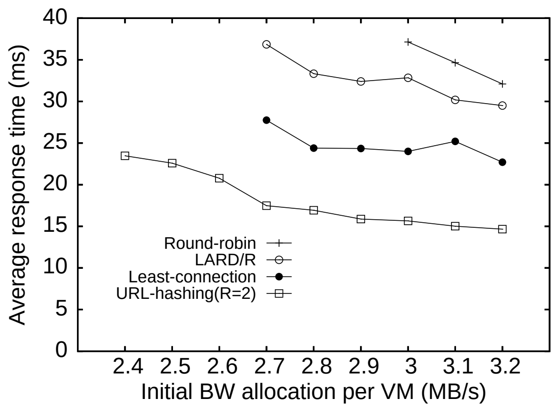


Figure 3.7 Average response time with different request dispatch algorithms.

We tried different settings but the difference in average response time is found to be within 10%. With frequent bandwidth allocations, using a larger α is found to result in a shorter average response time. The opposite trend is less obvious in the graph. A large α is also less likely to generate dramatic but transient changes in resource allocation. For $\alpha = 0.9$, an adaptation interval of 3s is found to be optimal. We choose $\alpha = 0.9$ and adapt bandwidth allocation every 3s for our evaluation.

We compare URL-hashing in a Vibra-enabled web cluster with other approaches, including LARD, least-connection, and round-robin in a fixed bandwidth web cluster. LARD is configured with the parameters published in its original paper [58]. We first compare them in terms of average response time using the 10-minute trace described above. From Figure 3.7, given the same amount of bandwidth, URL-hashing with $R = 2$ is shown to be able to reduce response time, on average, by 35% and 51%, over least-connection and LARD, respectively. In addition, round-robin does not work if less than 3.0MB/s is given to each VM. LARD and least-connection do not work with less than 2.7MB/s, either. URL-hashing not only can work with 2.4MB/s, but also performs better than the others. On the other hand, the average response time of URL-hashing with 2.4MB/s is very similar to least-connection with 3.0MB/s. That is, URL-hashing achieves the same performance level with a 20% reduction in bandwidth need.

We have also measured disk read activities during the 10-minute evaluation period.

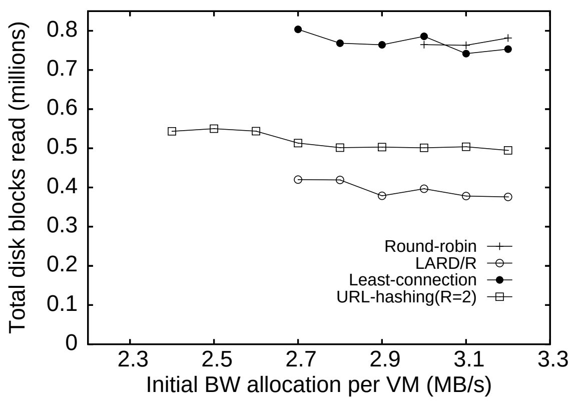


Figure 3.8 Total number of disk blocks read when different request dispatch algorithms are used.

Figure 3.8 shows that both the content-unaware dispatching algorithms read in a similar number of disk blocks. URL-hashing with $R = 2$ and LARD read in 33% and 49% fewer blocks, respectively. When bandwidth is scarce, disk activities are slightly increased because, with a prolonged service time, the number of concurrent connections is also increased and more memory is used for process images and network buffers (instead of disk cache).

While fewer disk activities imply better cache hit ratio and scalability, LARD has a *centralized stateful* dispatcher which may eventually become the bottleneck. By contrast, content-unaware approaches have the most scalable front-end but result in lower cache hit ratio and higher disk activities, whereas the URL-hashing is a balanced approach and can make a trade-off with the replication factor.

Throughput Comparison

In another experiment, we measured the throughput of the cluster with different levels of concurrency. Instead of replaying the trace according to the recorded request arrival times, the client opens a fixed number of connections to the request distributor concurrently and issues requests as fast as possible. Each VM is allocated 64MB memory and 6MB/s bandwidth initially for this experiment. The number of concurrent connections is increased from 140 to 220 and the throughput is plotted in Figure 3.9.

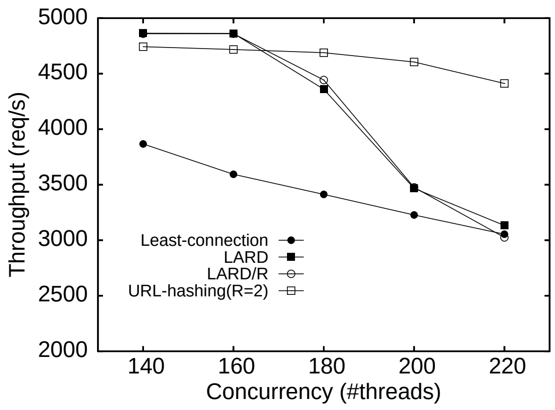


Figure 3.9 Comparing the throughput of different algorithms at different concurrency levels using the World Cup trace

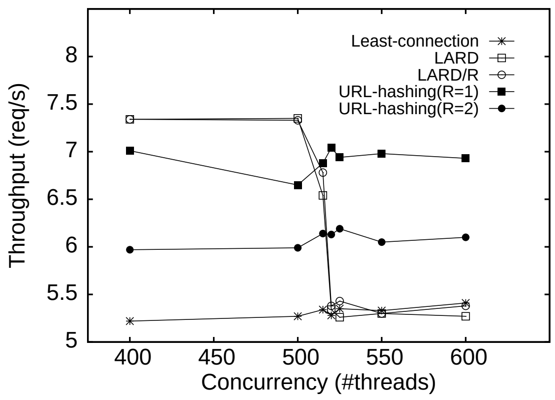


Figure 3.10 Comparison of the throughput of different algorithms at different concurrency levels using YouTube-like workload

As shown in the figure, the throughput of the least-connection algorithm decreases as the load increases, and the least-connection is the slowest among all algorithms we tested due to its excessive disk activities. With 140 concurrent connections, LARD and URL-hashing outperform the least-connection by 26% and 23%, respectively. While LARD is slightly better than URL-hashing at low concurrency levels, its performance degrades significantly with increasing loads, because the hot objects generate significantly more traffic than others and the network link becomes the bottleneck.

3.5.4 YouTube Workload

Besides the World Cup trace, we built a YouTube workload generator and fed the Vibra Web cluster with much larger file sizes and bandwidth demands. There have been a number of previous studies on YouTube traffic characterization [16, 20, 40]. The workload generator first creates a number of files with random IDs, each of which represents a video clip. The bit-rate and length of each video clip follow the statistical model reported in [20]. The accesses to these files follow Zipf's law with $\beta = 0.56$ [40]. We use a different source for the popularity distribution because the result was derived from a *real* trace, instead of a data set generated by a crawler. The crawler follows the related video links so the data set may be biased toward popular videos. Since the relationship between popularity and video size (or length) is unknown, we assume they are uncorrelated.

Using the workload generator, 2,000 video files, which take up a total of 12GB disk space, are generated. The workload consists of 4,000 requests and the size of total data transfer is 25GB. To serve much larger workloads like this, we allocate 768MB memory to each VM and a total of 80MB/s bandwidth to the cluster. Since no request inter-arrival time statistics were available, we could only benchmark the throughput of the cluster. The number of concurrent connections is increased from 400 to 600 and the results are plotted in Figure 3.10.

Since the video IDs are randomized and encoded in URL strings, URL-hashing simply divided all videos in equal-sized bins. Each bin also has similar popularity and file size distribution. In other words, there is no obvious hot-spot in the cluster. Both URL-hashing and LARD, and their replication derivatives take advantage of reduced disk accesses and are able to achieve a higher throughput than the least-connection algorithm. LARD and LARD/R are about 40% faster than the least-connection algorithm, while URL-hashing (where $R = 1$) and URL-hashing/R (where $R = 2$) are about 30% and 14% faster, respectively. Since the working set size is much larger than the memory size, URL-hashing performs better as more videos can be cached in memory. While LARD/R also has a replication logic, none of the servers in the cluster is underloaded, so no replica was created in this experiment.

When the number of concurrent connections increased to 515, both LARD and LARD/R start to degrade dramatically. By the time it reaches 525, both of them performed like the least-connection algorithm, and URL-hashing/R is 32% faster than LARD. It is because LARD and LARD/R simply choose the least-loaded server for dispatching a request when the destination server is overloaded. When all servers are overloaded, LARD and LARD/R simply degrade to the least-connection. This experiment shows that the performance of LARD and LARD/R depends on correct adjustment of the under-load and overload threshold parameters. Adjusting the replication factor in URL-hashing is straightforward because the ratio of working set size to memory size is easier to estimate.

3.5.5 Dynamic Memory Allocation

Since the overhead of estimating working set size is much higher than that of estimating bandwidth demand, we first evaluate our approach by changing the measurement frequency. From Figure 3.11, we can see that increasing the deactivating period from 10s to 60s does not have any impact on average response time, meaning that our approach has a very low overhead while a periodic spike of 5ms in average response time can be observed. On the other hand, dropping caches is extremely expensive and hence infeasible.

Longer sampling periods, however, result in larger active list sizes because more objects are touched during the period. In Figure 3.12, we plotted the number of memory pages in the active list as a server handling web workload. Irrespective of the sampling frequency, the active list size does not oscillate much. This means that dynamic memory allocation can be performed less frequently and a lower sampling frequency is sufficient.

We applied dynamic memory allocation to our testbed, but found that the active list size difference between servers was not significant. The reason for this is actually simple: while hot objects require a lot of bandwidth, they don't take up much of memory space. It can

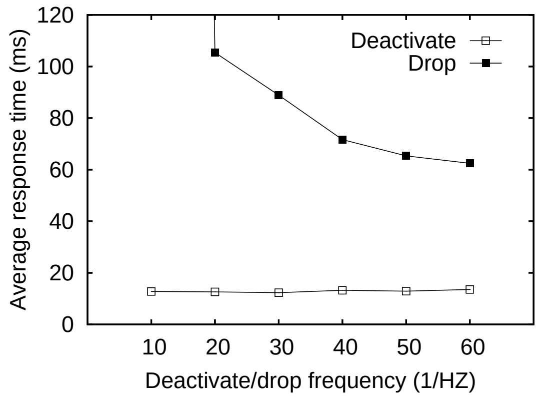


Figure 3.11 The impact on average response time by deactivating pages periodically.

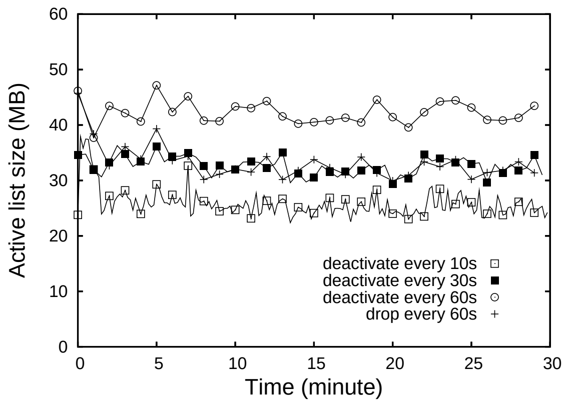


Figure 3.12 Working set size estimation by deactivating pages periodically.

also be seen from Table 3.1, where the numbers of distinct objects are not much different among servers. With proper tuning of parameters, disk activities are reduced on three of four servers. While a better cache hit ratio is achieved, the benefit cannot be translated into a reduction in average response time. Thus we do not combine it with dynamic bandwidth allocation and perform any further evaluation.

3.6 Discussion

In this section, we consider an alternative hash-based approach to design a locality-aware web server cluster. We also compared it to Amazon's Dynamo, a hash-based key-store system which has some similarities with our work. Before concluding this chapter, the limitations of Vibra is discussed.

3.6.1 Alternative Hash-Based Approach

In this work, we divide the URL namespace into equal-sized pieces where the number of sub-namespaces is equal to the number of back-end server VMs we have. When there is a imbalance in resource demand of each sub-namespace, we transfer virtual resource allocation between VMs to improve resource efficiency and performance. Another approach is dividing the URL namespace into many pieces, much more than the number of back-end servers, and dynamically mapping each sub-namespace to back-end servers to achieve load-balancing. Since the hot spots are usually consisting of very few objects, one might need to divide the URL namespace into very small partitions to reduce the peak of a hot spot.

In order to realize the alternative approach, the back-end servers need to measure the resource demand of each sub-namespace. To measure throughput, the web server process can apply the same hash function again to each URL and report throughput for each sub-namespace separately. However, both CPU time and memory space demand are difficult to measure in the web server process itself. Another option is to serve each sub-namespace in a different logical partition in an OS, where the throughput of each logical partition can be measured. CPU time usage can also be measured easily if logical partitions are used. However, it is still difficult to measure working set size for each sub-namespace as the buffer cache is shared between all partitions. After knowing the resource demand of each sub-namespace, one may adapt the sub-namespace-to-server mapping at the front-end to achieve load-balancing, *e.g.*, by moving one or more sub-namespaces from the most-loaded server to the least-loaded one.

Comparing to our approach, the front-end load-balancer needs to keep a small number of state, which does not reduce its scalability as the state is not updated frequently. Multiple front-end load-balancers operating with a slightly out-of-sync mapping is also acceptable as the locality is only reduced during the synchronizing period. However, as the infrastructure may already be virtualized by the data center operator, hosting each sub-namespace inside a logical partition only increases overhead. In addition, in Vibra each type of resources can be adapted independently, but all types of resources are coupled together when fitting multiple sub-namespaces on a server. In the future, we would like to implement this alternative approach and compare it with Vibra quantitatively.

3.6.2 Comparing to Dynamo

As we have briefly introduced in Chapter 2, Amazon's Dynamo is a key-value store that also uses hashing to partition its key space and distribute the workload among a cluster of servers. Similar to the alternative approach that we discussed above, the key space is divided up to 30 times more than the number of servers, *i.e.*, each server is handling 30 small partitions. The number of partitions in an instance of Dynamo (a cluster of servers) is fixed during the operation. The partitions are always evenly distributed to all servers. In other words, only the addition or removal of servers changes the number of partitions assigned to each server.

This design is adequate for Dynamo because they assumed there are enough hot spots in key space that through hashing and partitioning hot spots can be evenly distributed to all servers. Clearly, the distribution of their workload presents enough diversity so the load imbalance can be handled. However, an instance of Dynamo is dedicated to a very simple task, such as shopping carts, session management, or product catalog. For these services, the size variation of each key-value pair and the resource demand of each look-up request are much more moderate than a typical web site, where the large objects can easily take up more than 100MB of memory space and creating dynamic content can use much more resources than serving static content.

3.6.3 Limitations

Vibra also has its own limitations. First of all, in this work we only address the imbalance in network throughput, our workload does not manifest enough variation in working set size. As we only deal with static content, we simply assume that CPU demand does not create a bottleneck.

We also assume that the underlying physical server can provide enough computing resources for a VM. If one server VM is the only VM running on the physical server and the physical server cannot satisfy its resource demand, the physical server becomes a bottleneck. In this case, we can migrate the VM to a larger physical server, or increase the replication factor to shed some workload to other server VMs.

Our goal is to balance the demand, and the utilization, of network bandwidth across a cluster of server VM. However, the proportional distribution is not optimal when we put other considerations, such as the interaction of upper layer protocols or file size distribution in the workload, into the picture. While Vibra Web serves as the framework of virtual resource transfer and we have showed promising results by transferring bandwidth allocation, further research is needed to improve the distribution strategy.

3.7 Summary

While load-balancing web clusters have been studied for a long time, virtualizing data centers and deploying web servers on VMs create many new challenges and opportunities. The resource allocation to VMs can be adapted dynamically and hierarchically as we described in the Vibra architecture. We then use this new flexibility to re-design a web server cluster.

With the Vibra Web server cluster proposed in this chapter, we showed that a locality-aware request dispatcher can be more scalable. Higher cache affinity is achieved by using the hash values of URL strings to select a subset of back-end servers. A cluster's performance is improved further by transferring resources between VMs to compensate for unbalanced resource demands.

We use both the World Cup trace and a YouTube-like workload generator to evaluate Vibra Web. Our experimental results have shown that the average response time is reduced by up to 51% and Vibra Web can achieve the same service level as others with 20% less bandwidth. We also showed that URL-hashing incurs less overhead than LARD. In future, we would like to apply the Vibra architecture to more and other cluster applications.

Chapter 4
Redundancy in Vibra

4.1 Introduction

People rely on many Internet services in their everyday lives. E-mails, instant messages, on-line trading, shopping, and even gaming, all these services require very high availability and consistent performability. Otherwise, service providers may incur huge monetary loss and their users may flock to competitors' services ruthlessly. While the servers are hosted within data centers that have redundant power facilities and network connections, failures can still occur even to the most reliable hardware. In large server installations, failures may happen everyday.

In a virtualized server-hosting infrastructure like Vibra, a hardware failure can result in a wider impact than in a traditional infrastructure as it can bring down all VMs running on top of it, which may involve multiple customers. Although virtualization cannot directly prevent any hardware failure, in this work we show that it can assist in building a more resilient server architecture.

Hypervisor, the software that enables virtualization in a system, can isolate certain types of faults in a server environment. While operating systems (OSes) and server programs have tens of years of development, their large code bases make them difficult to become error-free. When a software error happens and results in a degradation in performability, the hypervisor can simply destroy the problematic VM and restart it. Although hypervisor itself is also a piece of software, its simpler functionality and smaller code base make it easier to be verified and become more reliable.

When a hardware failure is detected, all VMs running on top of it need to be restarted. To accelerate the recovery, each VM can be restarted on a different physical server because the boot-up process is usually resource-intensive. However, while booting up a VM is usually quicker than booting up a physical server due to fewer hardware auto-detections and self-tests, the VM cannot provide any service and results in a lost of productivity.

In the previous chapter, we re-designed a web server cluster, *Vibra Web*, to provide scalable and efficient web services. In this chapter, we propose a reconfiguration scheme to improve the performance of a Vibra Web server cluster during the failure recovery stage. Upon the detection of a failed VM, its workload can be transferred and the resources allocated to it can be re-allocated to one or more VMs in the same VM cluster, called *surviving VMs*, given the destination physical host has enough unallocated resources. With additional resources, surviving VMs can handle additional workload and compensate for the lost of productivity.

In addition, instead of naïvely changing the hash function in the URL-hashing request dispatcher to match the number of operational nodes in the cluster, we modify the mapping between hash values to server subsets to retain the existing locality in surviving VMs and share the faulty server's workload among a number of surviving VMs. This approach improves the throughput during the recovery stage. While it is similar to consistant hashing schemes, previous works simply assume load imbalance is harmless and do not address hot spots at the same time [50, 63, 72].

When a replacement VM is started, it can regain the faulty VM's resources. The original hash value assignment in URL-hashing can be restored. With all these approaches combined, the impact on service level can be minimized. We demonstrate the effectiveness of our redundancy scheme by showing that its throughput degradation is minimal.

The rest of the chapter is organized as following. We first review the Vibra service model and discuss various points of failures in Section 4.2. In Section 4.3, the enhancements to the URL-hashing algorithm are detailed. The performance is evaluated in Section 4.4 before we summarize the chapter in Section 4.6.

4.2 Models

Data centers can employ virtualization technology to consolidate servers and improve physical server utilization. In addition, virtualized resources can be easily added to and removed from virtual machines (VMs). With these flexibilities, we created a framework, called *VIrtualization-Based Resource Allocation* (Vibra) and built a web server cluster, *Vibra Web*, which consists of a front-end request dispatcher, back-end web servers that hosted in VMs, and a resource-allocation agent. Incoming HTTP requests are dispatched to one of the VM according to the URL-hashing algorithm. Imbalanced workload is then handled by virtual resource re-allocation.

However, virtualization itself does not prevent failures from happening. Placing VMs blindly to physical servers can even deteriorate availability. In the presence of potential failures, adding service redundancy to improve service availability is usually more cost-effective than building a more reliable physical server. We next discuss common redundancy approaches for each component in a Vibra Web.

4.2.1 Request Dispatcher Redundancy

The request dispatcher has a stateless design. To improve front-end redundancy, multiple request dispatchers can be deployed with typical high-availability approaches. For example, when two request dispatchers are being used, one can be designated as the master node and the other one as the slave node. The master node then periodically sends heartbeat signals to the slave node. If the heartbeat signals are not correctly received by the slave node in a timely fashion, the slave node can superseded the master node by cutting off its power supply and taking over its MAC and IP addresses.

During the transition, active HTTP connections are disrupted as the TCP connection states are lost. While the fault is exposed to end-user, it is not a problem because the integrity of higher level abstractions, such as an on-line shopping session, can be designed to tolerate interrupted HTTP connections, *e.g.*, by rolling back a transaction. Since the redundancy design in URL-hashing request dispatchers is no different from other stateless load-balancing algorithms, we do not discuss this type of failure any further.

4.2.2 Local Network Redundancy

While servers in data centers can only be physically accessed by a limited number of personnel, local networks can still suffer from hardware failures, loose connectors, and, unfortunately, cable-biting mice. Redundancy in local networks is usually implemented by deploying redundant network links and switching devices.

Both request dispatchers and physical servers can connect to multiple switches with one or more links each. Switches can also connect to each other and form a loop or other topologies that provide redundancy. Configuration protocols such as spanning tree protocol (STP) are able to respond to link failures and keep the network connected. Local network redundancy is already extensively implemented. Because the failures are transparent to upper layers, we simply assume local networks are reliable.

4.2.3 Hardware and VMM Redundancy

Although physical server are usually equipped with redundant power supplies and fans, failures can still happen. For example, most chip-sets do not have a redundant design. If a chip-set failed, the system may become unusable. Similarly, when an uncorrectable memory error is detected, most servers cannot mask the fault and need to stop. When a hardware failure is detected, all hosted VMs also failed while the problem (such as failed memory) may only affect one of them.

Virtualized server also need to guard against failures in the virtual machine monitor (VMM). VMM is the software that virtualizes the hardware and allows multiple OSes to run simultaneously on top of it. Since it provides simpler functionality than traditional OSes, its code base is much smaller and easier to verify. However, a software bug can cause protection error which also stops all VMs running on top of it.

Clustering is a common technique to guard against hardware and VMM failures. In particular, each VM in a VM cluster needs to be placed on a different physical server. Otherwise, redundancy is compromised. Data-center operators can only consolidate VMs from different clusters on a physical server.

Recovering a failed physical server is also easier if the server is virtualized. Traditionally, a failed physical server can only be substituted by another physical server of the same or better (or, at least, similar) configuration. However, a failed VM-hosting physical server can be recovered by restarting individual VMs on multiple working physical servers. Therefore, it is not necessary to have a single spare host that has a computing power equal to, or larger than the utilization of the failed server. Also, it is also faster if multiple VMs are booting up concurrently on multiple physical servers.

4.2.4 OS and Software Failure

In a hosting framework like Vibra, both the OS and server software are provided by data center customers. While modern OSes are more extensively tested, a bug in server software can result in interrupted service. Fortunately, both OS and server software are running within a VM and VMM is able to contain this kind of failure.

Recovering from software errors in Vibra is not different from that in physical server cluster settings. One can simply restart the VM to reset it from the faulty state. One can also use multi-version programming to reduce the chance of the same error happening on other servers. During a reboot, the resource-allocation agent can transfer some of the VM's resources to other VMs of the cluster.

4.2.5 Fault Model

For the rest of this chapter, we assume that failures are from faulty hardware or VMM. Since each VM in a cluster is placed on a different physical server, each failure may render at most one VM in each VM cluster inoperable. We also assume that failures are of the fail-stop type and can be detected in a timely and reliable manner (*e.g.*, by detecting the absence of heartbeat signals). We do not discuss the detection mechanism in this work. Multiple failures may happen simultaneously. Although this work is focused on a single failure, our enhancements can be easily extended for multi-failure scenarios.

4.3 Redundancy in Vibra Web

In a typical physical cluster setting, once a server failure occurs, the cluster simply loses part of its processing power. The performance of the cluster remains degraded until the failed server is restored. Service-level degradation, however, is unfair to the customers if the failure is caused by unstable hardware or VMM, because the customers may not get what they paid for. Unallocated server resources in a data center are not easily accessible, either.

We propose two mechanisms in Vibra Web to reduce service degradation caused by server failures. First, upon occurrence of a server failure (hence loss of a customer's VM on that server), the data center operator can allow the customer to allocate additional resources, if available, to his surviving VMs. The amount of additional resources the customer can allocate is commensurate with what the failed VM had before its failure. The resource-allocation agent can then issue resource-allocation commands so that the VM cluster for the customer can regain all its resources on the failed VM. Second, the request dispatcher can be adapted to use the transferred resource more efficiently. We augment the URL-hashing algorithm such that existing locality in back-end servers is preserved. Although we only consider egress network bandwidth in the following discussion, the scheme can also apply to other types of resource.

4.3.1 Resource Compensation

If a failure occurred to a physical server where VM_j is currently hosted, VM_j becomes non-operational and cannot provide any service. The amount of bandwidth BW_j currently assigned to VM_j becomes unavailable to the VM cluster. The data center can credit BW_j Mbps back to the resource-allocation agent, which can, in turn, allocate that bandwidth to any of surviving VMs in the cluster, given the physical servers hosting the surviving VMs have enough unallocated resources.

When the data center operator recreates VM_j, possibly on a different physical server, the failed VM initially has BW_j Mbps to use. While most servers do not require egress bandwidth during boot-up, other types of resources, like CPU shares and memory, are mandatory in booting up a new server. To allow the newly-created VM to warm up, the data center operator can reclaim the extra bandwidth only after a certain period of time.

4.3.2 Adapting Conventional Algorithms

For simple load-balancing algorithms like round-robin or least-connection, dealing with a failed server is as simple as removing a server from its configuration, resulting in a cluster with one less servers. In locality-aware algorithms, however, requests destined for the failed VM cannot be delivered.

For example, in LARD (and LARD/R), upon detection of a server failure, one can traverse the dispatching history and remove all entries with the failed server as (one of) its destination. Another approach is to treat the failed server as deceased and invalidate entries when the failed server is referenced. Since a replacement VM will not contain any state (cached files) when it is newly created if the latter approach is used, the dispatcher should treat the replacement VM as a new server and continue to invalidate old entries until all references are accounted for.

Since the design of LARD and LARD/R assumes all back-end servers to have the same processing power, the resource allocated to the failed VM can simply be re-distributed evenly to the surviving VMs (assuming that the physical servers hosting them have enough unallocated resources).

4.3.3 Adapting URL-Hashing Algorithm

The URL-hashing algorithm also needs to be changed in order to respond to a server failure. However, unlike LARD, URL-hashing algorithm is stateless, and thus, need not invalidate any state.

One way to avoid sending requests to the failed server is to choose another hash function returning $N-1$ different values that correspond to $N-1$ sub-namespaces. Similarly, $N-1$ back-end server subsets are also created with replication factor R and have a 1-to-1 mapping to each sub-namespace. We call this the *naïve* approach. However, this approach does not respond well to a failure. For example, in case of $R=1$, the naive approach will lose 50% of locality for all cluster sizes in the worst case which happens when either the VM that was handling the first sub-namespace or the VM that was handling the last sub-namespace fails. The average locality loss decreases as the size of a cluster increases. If we assume that all back-end servers have an identical failure probability, for a cluster of 10 servers, the average loss of locality is 37%. Obviously, this naive approach does not perform well during the degraded operation.

We plotted the average locality loss for different cluster sizes and different replication factors in Figure 4.1. The average loss of locality is reduced if R is greater than 1. However, it is still much worse than the optimal case, as described below.

With a cluster of N servers and replication factor R, each server handles R/N of the entire URL namespace when the URL-hashing algorithm is used. Ideally, only those R instances (each replica is counted as a different instance) need to be moved to other surviv-

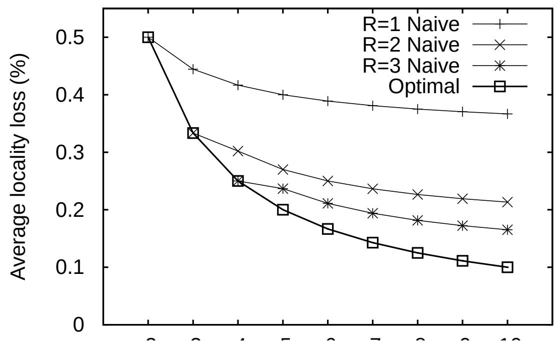
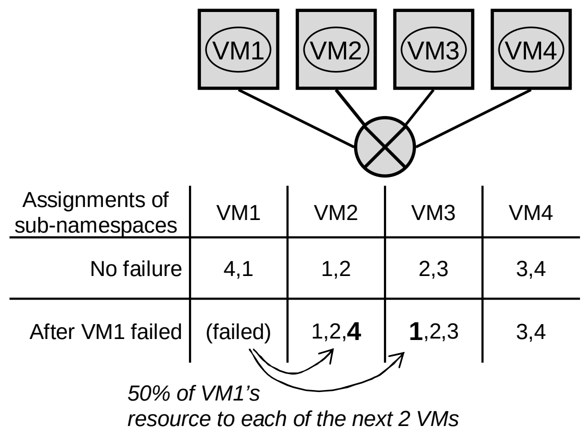


Figure 4.1



Assignments of sub-namespaces	VM1	VM2	VM3	VM4
No failure	4,1	1,2	2,3	3,4
After VM1 failed	(failed)	1,2,**4**	**1**,2,3	3,4

Figure 4.2 The possible location of cached objects before and after recovering from a failed server in a cluster.

ing VMs in the cluster. Since there are a total of $R \cdot N$ instances of sub-namespaces, in the optimal case, only $R/(R \cdot N) = 1/N$ of locality is lost. That is, only the states (cached files) on the failed server become unavailable.

To achieve the optimality, we again exploit the flexibility of resource allocation in Vibra, and create an uneven namespace distribution among the surviving VMs. When VM_j failed, instead of choosing a hash function that has $N-1$ values, we simply skip VM_j in the generation of server subsets. In other words, subset i now contains the next R *available* servers starting from VM_i. For example, subsets j and $j+1$ both contain $VM_{j+1}, \ldots, VM_{j+R}$, and subset $j-1$ contains $VM_{j-1}, VM_{j+1}, \ldots, VM_{j+R-1}$.

An example of this optimized approach is presented in Figure 4.2 under the assumption that a cluster contains 4 VMs ($VM_1, \ldots, VM_4$) and uses a hash function that returns values between 1 to 4 to select a subset of VMs. When VM_1 failed, it is marked as unavailable, and therefore, requests in namespace 4 will be dispatched to the least-loaded of VM_4 or VM_2, and requests in both namespace 1 and 2 will be dispatched to the least-loaded of VM_2 or VM_3. Namespace 3 remains unaffected. The difference between this alternative server selection and the original selection is shown in bold in the figure. In this case, only 2 out of 8 sub-namespace instances are changed, which turns out to be optimal.

The R VMs next to the failed VM in the ordered list of VMs will each have one more sub-namespace than what they have before the server failure, in order to take over the workload of the failed VM. Since we do not keep track of resource usage at the sub-namespace level, we simply assume that each sub-namespace consumes the same amount of resource on the failed server. To accelerate the recovery, $1/R$ of the resource held by the failed VM is transferred to each of the next R VMs. The resource-allocation agent still moves virtual resources between live VMs as usual, to cope with workload imbalance. Once the failed VM is restored, its original workload and resources can be transferred back to it.

Similar to that in [80], the recovery of ongoing requests on the failed VM can be handled by making partial requests (using the "Range" HTTP header) to the newly-assigned VM. Partial responses are then concatenated and returned to the clients.

4.4 Evaluation

We reuse the same testbed in Chapter 3 to evaluate the performance of various request dispatching algorithms during degraded operations. We first use the World Cup trace to evaluate the throughput of the cluster during normal operation, where the 4 VMs in the cluster are all in operational state. Each VM is hosted on a different physical server and configured with 72MB of main memory and 6Mbps of network bandwidth. The client is configured to initiate 250 concurrent connections to the cluster. A comparison of throughput for different scenarios is plotted in Figure 4.3.

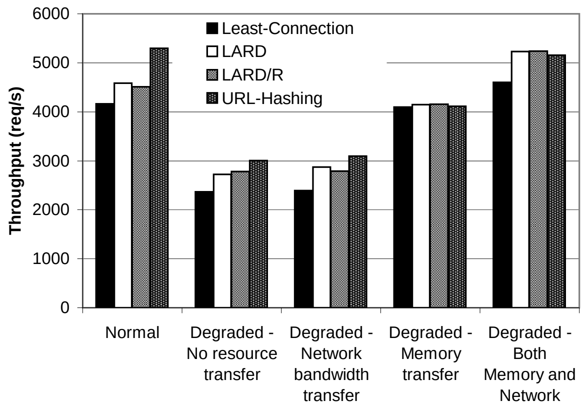


Figure 4.3 Comparison of the throughput of different algorithms for different degraded scenarios.

During normal operation, URL-hashing/R outperforms LARD, LARD/R, and least-connection by 15–27%. The performance gain came from the bandwidth reallocation between VMs and fewer disk accesses. When one of the four servers failed, all algorithms took a significant performance hit. Without transferring any of the failed VM's resource to the surviving VMs, the disk I/O is found to become the performance bottleneck. While URL-hashing/R still outperforms the others, its lead was cut to 8–27%.

Since the bottleneck was in disk I/O, when the network bandwidth of the failed VM was made available to the surviving VMs, there was only a 1–3% performance gain in throughput. On the other hand, if the memory of the failed VM was transferred to the surviving VMs, we observed a 37–73% performance gain over the case without any resource transfer. The least-connection algorithm gained most from the additional memory as more files can be cached in the memory. The number of disk I/O accesses was reduced significantly with the additional memory, and hence, network bandwidth became the bottleneck.

When both network bandwidth and memory of the failed VM were transferred to the surviving VMs, the performance became comparable to that of normal operation. Least-connection, LARD, and LARD/R actually outperform themselves even in normal operation by 11%, 14%, and 16%, respectively, due to the larger per-VM memory size. URL-hashing/R achieved 97% of its normal throughput. While it became slightly (1–2%) slower than LARD and LARD/R, URL-hashing is observed to be able to adapt itself in case of

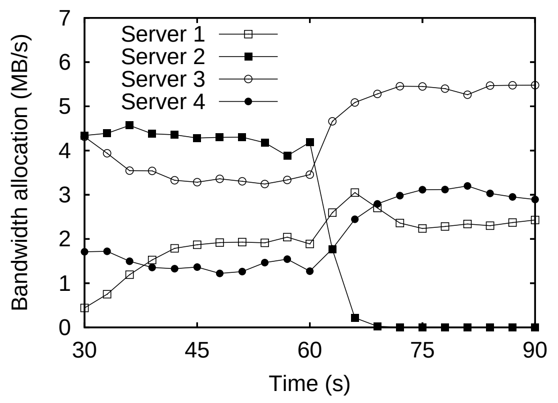


Figure 4.4 An example of failure recovery in a Vibra-enabled web server cluster.

degraded operation. No matter how many VMs are available in a cluster, URL-hashing/R can always provide the highest throughput.

We then evaluate the redundancy scheme by injecting faults into our testbed, where each VM is allocated 2.7MB/s at first. As one can see from Figure 4.4, four servers were serving requests initially before server 2 was killed at $t = 60$s. According to our design, the workload of failed server 2 was then moved to servers 3 and 4. The increased workload on server 4 also resulted in an increased workload on server 1, albeit at a lesser degree. With one less server in the cluster for the remaining 9 minutes of evaluation, the disk read activity level was increased by 14%. The average response time was increased from 19ms to 23ms, which is still lower than any other dispatching algorithms we tested. Moreover, to survive server failures, the fixed resource-allocation schemes (*e.g.*, LARD) require additional resource for each VM. For example, we need to allocate 3.6MB/s for each VM to survive the loss of 1 server and maintain the same service quality (average response time of 32ms). In this case, Vibra Web saves 25% of bandwidth and performs better than LARD.

4.5 Related Work

VM fault-tolerance techniques can be classified as either guest-OS level or VMM-level approaches. Guest-OS-level products such as Microsoft Cluster Service and Veritas Cluster Server manage service availability by monitoring the application and restarting failed VMs. Our approach is tailored for Web server clusters and integrates front-end dispatching algorithms and flexible resource allocation to improve performance during degraded operation.

VMM-level approaches such as VMware HA (High-Availability) and everRun VM from Marathon Technologies synchronize the execution of multiple VMs with tolerant physical server failures. These approaches are agnostic to guest OSes and shorten recovery time but have higher overhead in normal execution and resource commitments. Web servers do not require the highest single-server availability because the integrity of higher-level semantics (like processing credit cards) are usually provided elsewhere in a multi-tier system.

4.6 Summary

In large-scale computer system deployments, hardware failures are becoming more and more frequent and unavoidable. Instead of simply creating redundancy to achieve high-availability, we showed that by integrating fault-tolerance into the system design, we can achieve better performance in degraded operation. Data center customers can benefit from it by using fewer resources to achieve their service-level demand.

In this work, we first discussed various types of failures and approaches to making the Vibra infrastructure and the Vibra Web more reliable. With the flexibility in dynamic resource allocation, we enhanced Vibra Web to respond to server failures better. The design conserves as much locality as possible and adapts resource allocation to modified workload distribution. It can handle multiple failures as well. Evaluation results showed that Vibra Web can keep up to 97% of its throughput when 1 of the 4 server is failed. Comparing to LARD in a fixed resource-allocation cluster, Vibra Web allows a customer to request 25% less bandwidth resource while being able to survive the lost of 1 server in a 4-server cluster.

Chapter 5

DPS: A Distributed Proportional-Share Scheduler in Computing Clusters

5.1 Introduction

Recently, cluster computing has become a popular means of realizing a growing number of applications to lower prototyping and experimentation time and cost. Scientists and engineers use computing clusters to simulate the effects of a large number of application design choices and operating parameters. Computer animations, financial operations, bio-informatics studies, space exploration, and many other fields all require precise modeling and extensive computation on models to obtain accurate and detailed results. Without enormous computing power, recent advances in these application areas would not have been possible.

In order to meet the aforementioned computing needs, larger and larger computing clusters are being built. A computing cluster may consist of more than 100,000 nodes while each node may contain one or more single- or multi-core CPUs. Coupled with high-speed, low-latency interconnects, a cluster provides unprecedented computing power in a limited space and enables the realization of even more complex applications. Many companies, including Google, Microsoft, Amazon, and Sun, also built data centers, where even more servers are hosted in a building, and allow their customers to obtain computing resources and run their applications. As the architecture of a data center and a computing cluster has many similarities and shares more and more components, data center can also be used to carry out the tasks that were usually run on computing clusters. For example, data centers have been involved in rendering computer generated graphics in movies [1]. Private companies also invest in building large computer clusters and data centers and offer its computing powers to other companies [2].

Parallel programs, which may simultaneously utilize more than one node, are frequently used to solve large-scale complex problems. When users submit parallel programs for execution, they usually specify resource requirements, such as the number of computing nodes, amount of memory and disk space. Although large clusters may sometimes devote all their computing power to a single large parallel program, they often employ a space-sharing schedule of multiple parallel programs in order to improve their utilization and throughput.

If each parallel program can always utilize all the resources allocated to it, the resource utilization of the entire cluster would be very high. Unfortunately, as the problem size scales up and more detailed models are used, it becomes very difficult to create parallel programs that place balanced loads on all of the allocated nodes.

Depending on the type of application and how parallel programs are constructed, unbalanced resource utilization can result for several reasons. First, a parallel program may comprise a few smaller collaborative parallel sub-programs, each optimized for a specific purpose and running in parallel with others. When they are put together, the demands for executing different sub-programs are likely to be different and vary with time. Therefore, one sub-program may have to wait for the result from another sub-program (thus staying idle). The synchronization cost would also increase with the problem size. Moreover, both extensive use of program libraries and rapid application-development requirements force the programmers to spend less time on optimization of the program structure, producing less-balanced and lower-quality programs.

Since *Space-sharing* alone cannot provide satisfactory overall utilization, *time-sharing*

[1] HP and DreamWorks Give Innovation a Starring Role in ”Shrek 2.” http://www.hp.com/hpinfo/newsroom/press/2004/040419a.html

[2] Tata’s supercomputer Eka is fastest in Asia. http://economictimes.indiatimes.com/articleshow/msid-2539368,prtpage-1.cms

becomes an obvious alternative. In a time-sharing environment, gang scheduling and many other co-scheduling schemes have been proposed to reduce the program response time (the wall-clock time it takes to complete the program's execution) by implicitly synchronizing send and receive processes [2, 22, 32, 34, 35, 56, 57, 70]. However, they either suffer from high overhead, or require protocol-specific kernel modifications. Also, computing resources allocated to a parallel program are usually distributed to all nodes in a uniform fashion, and idle CPU cycles will result if the sub-programs place unbalanced resource requirements on the allocated nodes. Although the low utilization itself may not be a serious problem in research-oriented clusters, it is important to commercial utility clusters. If users are charged even for unused resources in a cluster, the cluster would be neither cost-competitive nor attractive to the users.

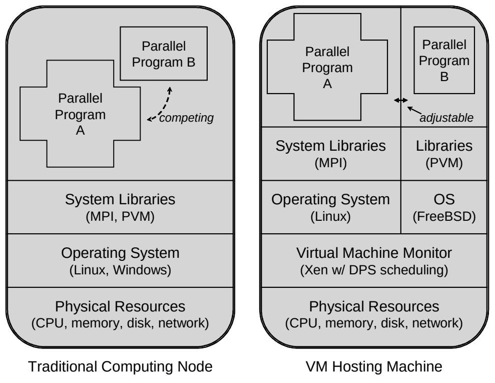
Instead of optimizing parallel programs over all possible inputs, we manage resource allocations to reduce idle cycles and improve overall utilization. Specifically, we enable programmers to decouple *program specification*—the number of nodes the program requires to run on—from *resource specification*—the maximum run-time resource usage—of a parallel program. For example, one program can use 4 single-CPU nodes but no more than 2.5 CPUs at any time. Time-sharing CPU schedulers can then be utilized to optimize resource usage.

In this chapter, we proposes a *distributed proportional-share* (DPS) CPU scheduling mechanism for parallel programs in a cluster computing environment. When a parallel program is submitted for execution, a process is created at each node and each process is initially given a certain share of CPU time. A proportional-share CPU scheduler is utilized at each node and the total share does not exceed the specified maximum. To deal with the intrinsic workload imbalance, part of the CPU share can be "transferred" at run-time from one process to another. We determine workload imbalance by monitoring CPU usage and the communication between processes. When a process A does not consume all its allocated CPU share and its continuing execution depends on other processes, we transfer part of A's CPU share to its peer processes. By periodically transferring CPU shares, the CPU share of each process becomes proportional to its relative load (to other processes'). It can also capture dynamic workload shifting. The program response time is, therefore, reduced, and the resources that were allocated to, but left unused by a process, can be utilized by other processes without affecting its performance.

During the course of transferring CPU shares as described above, bottleneck nodes can be easily identified. Schedulers can avoid creating new processes on bottleneck nodes. If there are more than one parallel program running on the bottleneck node, a migration mechanism can be triggered to alleviate the bottle-necking problem.

However, migrating processes of parallel programs is not trivial. Typical operating systems may not be able to provide enough isolation for security-minded customers, either. For these reasons, we encapsulate parallel programs in virtual machines (VMs). Instead of simply creating processes on each node, a VM is created first and a process of the parallel program is spawned within. Each node may host many VMs and such nodes are called *host machines*. The hypervisor is then responsible for scheduling CPU and other resources. VMs are also much easier to migrate between hosts [65]. This design ensures a high degree of performance isolation between resource-competing programs. On this platform, we implemented the DPS CPU-scheduling mechanism. Compared to uniform static CPU-share allocation, DPS is shown to reduce program response time significantly, while improving cluster utilization. Although the actual improvement depends on the real workload, we demonstrate that computing clusters can be utilized much better by separating program specification (required number of nodes) from maximal resource allocation and utilizing the share-transferring scheduler.

Our main contributions are (i) separating the maximum resource usage from program specification and (ii) using the proposed DPS scheduling mechanism to improve both program response time and cluster utilization. Two DPS scheduling strategies, namely *fully-distributed share-exchange* and *bank-assisted share-exchange*, are proposed and im-



Parallel Program B
Parallel Program A
competing
System Libraries (MPI, PVM)
Operating System (Linux, Windows)
Physical Resources (CPU, memory, disk, network)
Traditional Computing Node
Parallel Program A
Parallel Program B
adjustable
System Libraries (MPI)
Libraries (PVM)
Operating System (Linux)
OS (FreeBSD)
Virtual Machine Monitor (Xen w/ DPS scheduling)
Physical Resources (CPU, memory, disk, network)
VM Hosting Machine

Figure 5.1 A comparison between traditional computing node and VM hosting machine.

plemented on a specifically-designed VM hosting cluster that provides secure and flexible resource allocation and performance isolation. These schemes not only lower the cost of running a parallel program on the cluster, but also ease the creation of parallel programs.

The chapter is organized as follows. We start with the design of DPS in Section 5.2 which is followed by the share-exchange models in Section 5.3. The implementation of the proposed schemes is detailed in Section 5.4 and evaluated in Section 5.5. Section 5.6 summarized the chapter.

5.2 Design

A cluster supporting *distributed proportional-share* (DPS) scheduling is designed in two tiers. We first introduce a virtual machine (VM) hosting cluster, which enables performance isolation and security enhancement for running parallel programs. Each computing node is transformed into a *host machine* (see Figure 5.1). On the hosting cluster, we then build the DPS scheduler to provide coordinated resource scheduling across all nodes. This two-tier architecture is similar to a normal operating system (OS), where the lower-tier manages the underlying resources and the upper-tier schedules processes to coordinate usage of resources. We next elaborate on the design of DPS.

5.2.1 Cluster Design

Traditional cluster design relies on many caveats which make flexible resource-allocation difficult. We first discuss the shortcomings of the traditional design and the advantages of VM hosting clusters.

Traditional Design

In a traditional computing cluster, each node is loaded with an OS (*e.g.*, Linux or Windows), some parallel computing middleware (*e.g.*, PVM, MPI), and user programs. User programs are stored in a network-accessible storage while each node is also equipped with some scrub disk space for temporary storage.

When a user submits a program to a cluster, he usually provides a specification of the program, such as the number of nodes, the amount of memory, and scrub disk space on each node. He also specifies an estimated program execution time (the CPU time it takes to complete the program), so the cluster scheduler can reserve enough resources for its execution.

When a parallel program is executing in a cluster that only employs space-division multiplexing, it is entitled to utilize all memory and scrub disk space on the nodes it acquired, even if it doesn't require all the resources for its execution. It is very difficult to make the idle resources usable by others until the program completes its execution at which point mechanisms, such as back-filling, can kick in.

When there are more than one parallel program running on a node, the OS must schedule them on the CPU according to a certain policy like FCFS (First-Come-First-Serve). With a proportional-share CPU scheduler, each program can be guaranteed to receive a given share of CPU time. However, even when the total memory demand on a node doesn't exceed its physical capacity, a program still competes with others for buffer cache and network usage. Moreover, configuration or software version conflicts may even render time-sharing impossible. In a commercial cluster, parallel programs also require strict protection from others to prevent any information leak between them.

VM Hosting Cluster

In order to provide better protection between multiple resource-competing parallel programs running on a single node, one can use OS-level virtualization or similar OS facilities to logically "partition" the node, as is commonly done in very large servers or mainframes. It creates an illusion of having multiple smaller servers, each with a share of CPU, memory, and other resources. Although each partition can have its own software configuration, all partitions may be required to use the same kernel and typically share a buffer cache. In addition, partitions usually do not 'migrate.' Consequently, the partitioning itself does not provide the level of flexibility we would like to have.

Instead of using partitions, we decide to use platform virtualization as a vehicle to execute parallel programs. Similar to partitioning, each VM can be configured to have a portion of CPU time, a fixed amount of memory, local disk space, and network interfaces. Parallel programs execute on a cluster of VMs, instead of physical hosts. In a VM hosting cluster, standardized VM images can be packaged to provide common program execution environments; customized VM images can also be made to include proprietary software packages.

In addition to the ability of assigning CPU shares to VMs without tweaking the OS scheduler, for some hypervisors, we can even re-allocate memory on-the-fly. VMs can also be migrated between host machines. Two or more VMs can be hosted on a single host; they may even be assigned to the same parallel program. With these schemes, idle resources can be re-assigned, and hence, more parallel programs can be hosted/accommodated at the same time.

By running parallel programs on VMs, we can also provide better performance isolation and security measures. From a parallel program's point of view, it is the only program running within the VM, and hence, no other programs would compete for any of its acquired resources. The guest OS can also be configured to provide a very minimum set of services, so its security can be enhanced.

Once resources are virtualized, the specification of a parallel program also needs to be adapted accordingly. Since VMs are dedicated to a single parallel program, the specified number of nodes a parallel program requires only reflects the structure of the program. When submitting a parallel program, one can separately specify its maximum instantaneous CPU usage. Moreover, he need not break down the CPU allocation into each VM as the cluster resource scheduler—which we introduce below—can dynamically move resources between all VMs as needed to match the relative resource needs.

This scheme also frees programmers from the job of optimizing the program structure for load-balancing among the nodes. This may sometimes not even be possible because the input and, therefore, the workload may change dynamically. The load-balancing problem is actually translated into the problem of dynamically allocating resources. Upon completion of a program's execution, the history of resource allocation can also provide programmers a hint on potential bottlenecks, so he can improve the program.

In addition, dynamic resource allocation also enables users with a limited resource budget to execute unmodified large-scale parallel programs. He only needs to submit the program with a low CPU-usage cap and the resource scheduler will optimize resource allocation subject to the limited resource. While the program execution is prolonged, the resource utilization is improved (*getting the best bang for the buck!*).

Although the insertion of a hypervisor may incur some overhead while executing parallel programs, the advantages it provides—namely, more flexible and secure resource allocation and decoupling program structure from resource usage—can better utilize cluster resources and enhance throughput.

5.2.2 Scheduler Design

A DPS scheduler is composed of two levels. A cluster resource manager (CRM) controls cluster-wide resource allocation, and each host machine has a local resource scheduler. Although we only implement CPU scheduling in this chapter, our approach can be extended to the scheduling of other resources such as memory and network I/O. Each guest OS also has its own resource scheduler, where each parallel program can adopt any policy as the CPU time slots and memory space given by the node-level scheduler are under its full control. In what follows, we first present the design of a node-level CPU scheduler and then discuss the role of CRM.

Node-level CPU Scheduler

The CPU scheduler on each host machine utilizes a proportional-share scheduling algorithm, such as lottery scheduling [76] or borrowed virtual time (BVT) scheduling [31], to assign time slots to VMs. We choose a work-conserving CPU scheduler so that idle slots are also distributed proportionally to runnable VMs. The node-level scheduler is also responsible for keeping track of each VM's CPU shares and actual CPU utilization, and for creating and migrating VMs when the CRM instructs it to do so.

If a VM does not fully utilize the CPU shares it owns, the node-level scheduler may transfer its excess CPU shares to other VMs of the same parallel program. We will henceforth call the other VMs *peer virtual machines*. Each parallel program can employ different strategies to decide the amount of each transfer and its timing. We have designed distributed and centralized share-exchange strategies. Both strategies are illustrated in Figure 5.2 and discussed below.

Distributed Strategy In the *fully-distributed share-exchange* strategy, a node-level scheduler transfers CPU shares directly to another node-level scheduler. The target of the CPU-share transfer can be determined by analyzing the network traffic between VMs.

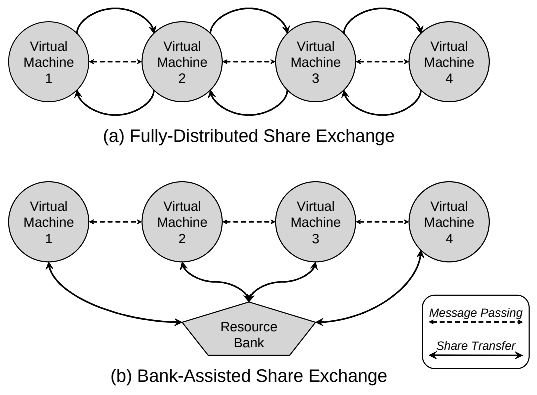


Figure 5.2 Two share-exchange strategies in DPS.

Specifically, if a packet from a remote peer VM (VM_r) unblocks the execution a locally-hosted VM (VM_l), the execution of VM_l is said to depend on VM_r, called an *upstream virtual machine* of VM_l. For example, a process in a VM may wait for a result from a process in another VM to continue its execution.

If excess CPU-shares are available for a VM, the node-level scheduler solely determines the amount of share to be transferred and initiates the transfer to its upstream VMs that do not have excess shares. Another node-level scheduler (receiver) can either accept or reject the transfer. The receiver usually accepts any CPU-share transfer, unless its CPU has been fully booked. When this happens, the receiver recognizes that the host machine is a bottleneck and notifies the CRM to seek for migration.

Centralized Strategy Another approach is to create a centralized CPU-share bank for a parallel program, called a *bank-assisted share-exchange*. Node-level CPU schedulers deposit excess resources into an account that represents unallocated CPU shares of a parallel program. If a VM fully utilized its CPU share and the host machine is not fully booked, the node-level scheduler requests more CPU shares from the bank. Otherwise, if the host machine is fully booked, it notifies the CRM and seeks migration. A banker process, which is supplied with a parallel program, is responsible for redistributing banked shares to all

VMs of the parallel program.

Comparison The fully-distributed share-exchange strategy is more robust than the centralized counterpart. A node-level CPU scheduler that fails to participate in share-exchanges only renders a limited amount of CPU shares unavailable to other VMs of the same parallel program. On the other hand, the fully-distributed strategy only transfers shares between peer VMs, so it will take more time to propagate excess shares to where they are needed while the bank-assisted strategy can always relay shares by one hop. Also, if the imposed workload fluctuates among distant peer VMs (in terms of their dependency), the bank-assisted strategy is more likely to achieve the global optimum of share distribution, where the fully-distributed strategy may only find local optima. We have evaluated these two strategies under different parallel program topologies in Section 5.5.

Effects on other VMs When there are multiple VMs running on one physical host, due to our choice of a work-conserving CPU scheduler, unallocated CPU time slots are also distributed proportionally to runnable VMs (for free). When one VM gains CPU shares, not only the guaranteed CPU time is increased for the VM, the VM will also get more CPU time from the unallocated CPU time slot pool (in terms of the portion of unallocated slots). Note that the minimum CPU time for other VMs on the same host are still guaranteed by the scheduler according to the CPU shares these VMs have. However, these VMs will lose free CPU time from the unallocated time slot pool. The throughput of these VMs may be reduced but the minimum CPU time agreement is not violated. If the reduction of CPU time makes other VMs the bottleneck of the corresponding cluster, they can also obtain shares from their peer VMs or bank accounts.

Cluster Resource Manager

The role of CRM is to manage the parallel programs that are currently executing in the cluster. Each host machine periodically reports its actual resource utilization to the resource manager. Based on the complete picture of current resource usage, the CRM may initiate VM migration to redistribute workload among host machines. A cluster scheduler, which manages the submission of parallel programs and selects programs to execute from a wait queue, may also decide to squeeze more programs into the cluster to improve utilization. The design of a cluster scheduler, however, is not the focus of this chapter.

A workload redistribution can be triggered by the identification of bottleneck host machines. A bottleneck host can be identified by a node-level scheduler that rejects incoming CPU shares due to its insufficient capacity. CRM can also declare a fully-utilized host machine to be a bottleneck host. Once a bottleneck host has been identified, the CRM can migrate the hosted VMs on the host to a less utilized node or a more powerful host. As migration either needs to suspend a VM for a short while or incurs some CPU overhead (see the evaluation results in Section 5.5.3), we propose a simple heuristic to determine which VM to migrate to which destination.

To select a VM to be removed from the bottleneck host, one can choose either a smaller VM (that takes up less memory and disk space) or a less-utilized VM (that uses fewer CPU cycles). If using normal (suspend-and-resume) migration, a smaller VM is easier to move. However, the suspended VM may also block the execution of its peer VMs and prolong the program execution. On the other hand, live-migration keeps a VM running and only suspends it while moving the remaining dirty memory pages [23]. Since a less-utilized VM usually modifies its memory pages less frequently and is easier to fit into idle resources, we opt to migrate the least-utilized VM.

We considered two strategies to choose a migration destination. The first one is to migrate the least-utilized VM to the least-utilized host machine; the second is to find the best-fit host machine such that the predicted CPU utilization is closest to 1. We choose to

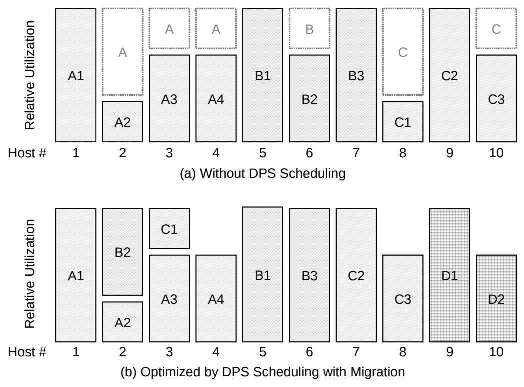


Figure 5.3 Parallel programs are packed together while DPS ensures resource allocation. Previously wasted resources (dashed boxes) are now utilized and a new program “D” is placed for execution.

migrate to the least-utilized host so it can obtain more CPU shares if needed.

To further improve cluster utilization, the cluster scheduler can allocate more programs to the cluster. For this, the CRM may repack current running parallel programs into fewer host machines. An example is provided in Figure 5.3 to demonstrate the process. It is similar to the traditional bin-packing problem, which is NP-hard. A better heuristics, which takes migration cost and past utilization history into consideration, is needed in order to improve host utilization and avoid frequent migrations.

5.3 Share-Exchange Models

We now formalize the cluster and program models we will use and then introduce the fully-distributed and bank-assisted share-exchange models. [3]

5.3.1 Cluster Model

We consider a VM hosting cluster of M host machines that are connected to each other via a high-speed low-latency switched network. A storage infrastructure is attached to the

[3]For simplicity, the subscript for each program and the time index are omitted in the discussion. Subscripts i and j indicate a specific VM and a specific host machine, respectively.

cluster and can be accessed from anywhere. A server node is also attached to the cluster and is powerful enough to perform services such as CRM or banker processes. Here, we assume that there are sufficient network and storage bandwidths, and the execution time of a program is not affected by its placement.

The computing capacity $C_j(j = 1 \ldots M)$ of each node is defined as the number of CPUs installed. The virtualization overhead can be incorporated by slightly reducing C_j. For example, to compensate a 5% CPU overhead in virtualization on a dual-CPU SMP node, one can advertise a computing capacity of 1.9 CPU instead of 2. The computing capacity is divided into CPU shares. If a VM can utilize multiple CPUs, the size of a CPU share can range between 0 and C_j. Otherwise, the share size lies between 0 and $min(1, C_j)$. Without loss of generality, we assume the cluster is homogeneous so that, when a CPU share is transferred between any two hosts, the *exchange rate* of CPU share is always 1. For heterogeneous clusters, an exchange rate needs to be established *a priori* to account for relative computing power.

5.3.2 Program Model

When a user submits a program for execution, he must supply the *program specification* and *resource specification*. Among other things, program specification includes the number of nodes N, which can be smaller than, equal to, or larger than M. Virtual machines, $VM_1 \ldots VM_N$, are created when a program starts its execution.

Resource specification includes the maximum instantaneous CPU usage W and the selection of a share-exchange strategy—*fully-distributed* or *bank-assisted*. Initially, for $VM_i(i = 1 \ldots N)$, a CPU share $w_i = W/N$ is assigned. Other resource specifications such as maximum execution time and utility specification such as deadline and reward are not used in the model for share-exchange.

Each VM_i may be hosted in different host machines. Distributed node-level schedulers report the incremental CPU time t_i used by VM_i every T seconds and conduct a fully-distributed or bank-assisted share-exchange. The reporting process need not be synchronized among all VMs. The CPU utilization u_i by VM_i during the last sampling period is then defined as t_i/T. The proportional-share node-level scheduler is configured to guarantee u_i is at least w_i if T is large enough and VM_i can fully utilize its CPU time. Note that the scheduler is also work-conserving and, therefore, u_i may be much larger than w_i. An excess share e_i for VM_i is defined as $e_i = max(w_i - u_i, 0)$. For example, when two CPU-intensive VMs having 0.5 and 0.25 CPU share, the actual CPU utilization will be 0.67 and 0.33, respectively, and there is no excess share. On the other hand, if a VM has 0.5 CPU share and results in 0.1 CPU utilization in the last sampling period, there is 0.4 excess CPU share.

5.3.3 Fully-Distributed Share-Exchange

In the fully-distributed share-exchange, we redistribute excess shares to the upstream VM, which is inferred from network packets (as discussed in Section 5.4). A list is maintained to contain the current upstream VMs for each VM_i, and old entries expire after a certain period of time. We assume the upstream VMs list contains P VM.

For each VM_i, the node-level scheduler on its host machine redistributes its excess shares e_i to all P upstream VMs at the end of each sampling period. VM_i can also withhold part of the excess shares. The purpose of withholding is to leave some room for workload fluctuation since a VM does not proactively ask for shares from its peer VMs. Given a withholding factor $wh_i (0 \leq wh_i \leq 1)$, the number of shares s_i that node-level scheduler will send to each upstream VM is calculated as:

$$s_i = \frac{e_i \cdot (1 - wh_i)}{P}. \tag{5.1}$$

Note that when the withholding factor wh_i is set to $1/(P+1)$, VM_i withholds the same amount of share as it will transfer to each upstream VM. After each successful share transfer, s_i is deducted from w_i. In the above example, if the VM having 0.4 excess CPU share has 3 upstream VMs and withholds $1/(P+1)$ of excess shares, it will initiate a share transfer of 0.1 share to the 3 VMs and, if all transfers are successful, retain 0.2 CPU share at the end of re-distribution.

5.3.4 Bank-Assisted Share-Exchange

In the bank-assisted share-exchange, all excess shares e_i, if $e_i > 0$, are transferred to the bank. These shares are deducted from w_i immediately. Otherwise, if the capacity of the host machine is not fully allocated, the node-level scheduler notifies the bank of the CPU share required to match a VM's allocation to its utilization. This amount is also limited by the remaining share of the host machine r_j. We call this amount *the share shortage* f_i, which can be written as $f_i = min(u_i - w_i,\ r_j)$. For example, a VM having 0.5 CPU share but utilizing 0.67 CPU has 0.17 CPU-share shortage.

The banker process redistributes all the excess shares deposited by node-schedulers every B seconds. B can also be smaller than, equal to, or larger than T. However, B is usually an integer multiple of T. Assuming the bank has a balance of E shares at the end of a redistribution period and $F = \sum f_i$ share shortage, the shares s_i that the banker process will send to each VM are calculated as:

$$s_i = max(f_i, f_i \cdot \frac{E}{F}). \tag{5.2}$$

For example, if the banker process has a total of 0.3 excess share and a total share shortage of 0.6, the VM that needs 0.17 share will be given 0.085 CPU share.

If the total amount of excess shares are greater than the total amount of share shortage, remaining excess shares are proportionally distributed to all VMs. VMs do not withhold CPU shares in this scheme because they can explicitly ask the banker for more shares.

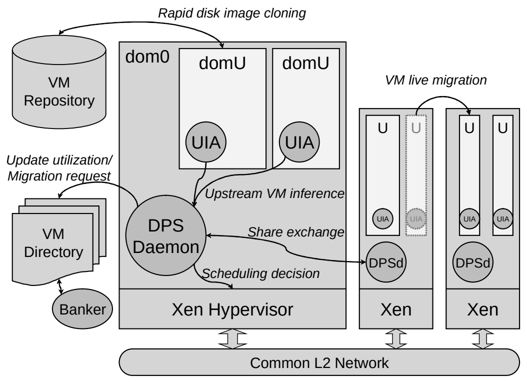


Figure 5.4 The architecture of a VM hosting cluster.

5.4 Implementation

We implemented the VM hosting cluster and DPS scheduler on Xen. We first discuss the construction of a Xen VM cluster and auxiliary services, namely, the VM repository and the VM directory. Then, we describe the implementation of the DPS daemon, which runs on each host machine to perform many DPS functions, and the details of the upstream inference system. The VM directory and the banker process implement the cluster resource manager (CRM), and the Xen scheduler and the DPS daemon cover the function of node-level scheduler discussed in Section 5.2. An overview of the VM hosting cluster implementation is illustrated in Figure 5.4.

5.4.1 The Xen Cluster

There are several virtual machine monitors (VMM) that can be used to build a VM hosting cluster on x86 hardware, which is most commonly used in computing clusters. Among them, we choose the Xen VMM because of its low CPU overhead and live-migration capability. Live-migration can make the downtime very short (less than 1 second for most workloads [23]), and we evaluate its impact on parallel program execution in Section 5.5. We use Xen 2 instead of Xen 3 as the newer version was not stable when we started this project.

Platform virtualization provides a flexible and secure method for allocating resources to parallel programs. Although virtualization inevitably incurs overhead, the para-virtualization approach adopted by Xen keeps the overhead at minimum [11]. Instead of inspecting binary code and rewriting privileged instructions before program execution, Xen requires the guest OS kernel to use Xen hypercalls. Considering the emerging hardware virtualization, we expect the overhead to be reduced even further in future.

To schedule CPU usage for multiple VMs, Xen provides three different schedulers: borrowed virtual time (BVT), Atropos, and round-robin. BVT is a proportional-share scheduling algorithm, which also yields low response times for interactive processes [31]. Atropos is a soft real-time scheduler which can set aside a specified amount of CPU time and manage latency for each VM [53]. It can also distribute the slack CPU time between real-time and best-effort VMs. Round-robin changes turn to execute all runnable VMs. Among these schedulers, both BVT and Atropos can provide a CPU-share guarantee. Since BVT is simpler than Atropos, we use BVT to schedule VMs.

Xen provides three different interfaces to interact with. At the lowest level is a C library `xc`, which issues hypercalls to communicate with the Xen VMM. The Xen daemon `xend`, which is built on the library, exports information and creates a control interface via HTTP. The command line tool `xm` accesses `xend` and provides similar functionalities.

We use the simpler `xm` command to create and migrate VMs. On the other hand, we use the low-overhead `xc` library to periodically extract actual CPU-usage information from the hypervisor and set scheduling parameters in the DPS daemon, as described below. However, some information, such as the mapping between VM name/ip and Xen domain ID (which is a unique identifier within the Xen VMM) is only available in `xend`. As we need to look up the domain ID for each CPU share transfer, we access the data files created by `xend` directly to avoid the protocol overhead.

Xen VMs are connected to the network via virtual Ethernet interfaces. A virtual interface can be bridged to the Ethernet that the driver domain is connected to, or routed via the IP layer in the driver domain. For easy VMs migration, we adopt the bridging approach. After a VM migration, the Linux virtual bridge in the destination host machine can learn the change. Since the VM is still connected to the same Ethernet, all VMs can send packets to it without any change in configuration.

5.4.2 VM Cluster Infrastructure

Other than running the Xen VMM in each host machine, we also need a few infrastructure services to operate a VM hosting cluster. We created a VM repository to enable migration, a VM directory to support resource management, and a banker process while using the bank-assisted mode.

VM Repository

One limitation of VM migration is that the disk image must be accessible via the network. We create a RAID-0 disk array to store all VM disk images and make them available network-wide via NFS. We also create a standard VM disk image for executing PVM applications. The size of the disk image is 1GB and it does not contain any configuration information, so cloning a VM only needs to copy a disk image in place. Once a VM boots up, it configures itself via DHCP and NIS. After all VMs of a parallel program are ready, user programs are loaded from the network and start their execution. In larger installations, a storage-area network and a directory service can be used instead.

VM Directory

When transferring shares, the node-level schedulers and the banker processes only know the address of the share-receiving VM, but not the address of the host machine that hosts the VM. To deal with this issue, we create a VM directory to provide a mapping between the IP address of a VM, its host machine's name/IP address, and the Xen domain ID of the VM.

Xen domain ID is the unique ID of a VM. However, its uniqueness is valid only within a host machine and it cannot be specified while creating or migrating VMs. To deal with this deficiency, we use the data files created by `xend` to map a domain ID to VM's MAC address and then uses the `ethers` map in NIS/YP to find the hostname and IP of the VM (the DHCP server is configured to assign IP addresses according to the `ethers` map.) The mapping is stored in the VM directory once a VM is created and after a VM migration.

Besides the mapping, the DPS daemon periodically updates a host machine's overall utilization to the VM directory. The VM directory also serves as the bank to store excess CPU shares of each parallel program. The banker process, which is discussed below, is also part of the VM directory.

The Banker Process

We implemented the banker process as a thread of the VM directory service to reduce the communication overhead. A pluggable interface is defined for user-supplied banker processes (routines). Banker processes wake up periodically to redistribute excess CPU shares to their VMs. Each banker process may have a different wake-up frequency which is configurable by the user. Since banker processes require very little resource, any reasonable hardware can host the VM directory and many banker processes (threads) easily.

5.4.3 The DPS Daemon

At the core of the DPS scheduling is the DPS daemon running within the host OS (domain 0) on every host machine. Its function includes creating VMs, interacting with the node-level scheduler within Xen, reporting utilization to the VM directory, receiving the upstream inference result from hosted VMs, and conducts share exchanges in the Xen cluster. We implemented it in C and use the `xc` library to control the scheduler.

In the fully-distributed mode, the DPS daemon periodically retrieves the information on each VM's CPU utilization from Xen. According to the current CPU-share allocation, it calculates the excess share each VM has by using Eq. (5.1) in Section 5.3. The excess CPU shares are then evenly distributed to the corresponding VM's upstream. The DPS daemon looks up the host machines of each upstream in the VM directory and connects to the remote DPS daemons directly. It may also receive CPU shares from remote DPS daemons at any time. After each successful share-exchange, the DPS daemon changes scheduling parameters to reflect the changes in resource allocation.

In the bank-assisted mode, the DPS daemon sends and receives shares to and from the banker process, and the scheduler parameters are then changed accordingly.

5.4.4 Upstream Inference

In order to identify the upstream of a hosted VM, we need to correlate the packets received by a VM and how it affects the processes running in a VM.

Every packet received by a hosted VM is actually redirected via the driver domain. After the device driver in the driver domain received a packet destined for a VM, the host OS either bridges or routes the packet to the back-end of the receiving VM's virtual interface, which then uses Xen hypercalls to re-map the received packet and send an event (a virtual IRQ) to the receiving VM via a Xen event channel. The event may wake up the guest OS if it was blocked (idle), and the guest kernel receives the packet from the front-end of the virtual interface. Standard protocol processing is then applied and wake up parallel processes, if necessary.

Given the path of packet delivery, we can intercept every packet in and out of VMs at the driver domain, Xen VMM, or each guest OS. Inferring upstream VMs in the driver domain or Xen VMM is a more generic approach as it is VM-neutral. However, either inside Xen VMM or the driver domain (by issuing hypercalls), we can only learn the virtual processor's state (running or blocked) but not the parallel process' state inside a VM.

The virtual processor's state is different from the parallel process' state as not every incoming packet may wake up the blocked parallel processes. For example, data packets may not wake up the receiver process that is not waiting for the input. On the other hand, a data-less TCP ACK packet may indicate a successful data transmission or a connection establishment, and unblocks a process. Unless the protocol is well understood, it is very difficult, if not impossible, to infer dependency outside a VM.

We instrument the Linux kernel in unprivileged domains to infer the upstream relationship. After a packet is delivered to user space, the kernel will wake up all processes waiting for it. The sending host's ID is passed with wake-up functions. If the receiver process isn't running nor already in the run-queue, we declare the sending host is an upstream VM, and report its IP address and port number in the kernel log. A user-space program, the upstream inference agent (UIA), is then reading from the log periodically and proxies such information back to the DPS daemon running in the host OS (domain 0).

Some filtering has to be applied on the upstream VM in UIA. For example, locally-generated packets via a loop-back interface are ignored. Packets between CRM, DPS daemon and UIA are also not part of a parallel program. Messages to and from infrastructure services (*e.g.*, DHCP, NIS, NFS, NTP, *etc.*) also need to be excluded. Note that our scheme can also detect I/O dependency. When the exchange rate between CPU shares and I/O throughput is clearly defined, we can also trade excess CPU shares to improve I/O performance.

We have also considered identification of upstream VMs by instrumenting function calls in user-space. However, this methods suffers from the loss of scheduler's context *before* the reception of a packet. For example, a `recv()` call may cause a context switch to other threads of the same process or other processes in the same VM. By the time the call returns, the process is already in execution. If the call returns immediately, one may use some performance counters (*e.g.*, number of context switches) to infer the CPU state of the past. However, the ability of predicting past CPU state diminishes with time and system complexity. Therefore, we conclude that instrumenting in the guest kernel is simple and accurate in identifying upstream VMs.

5.5 Experimental Evaluation

In this section, we first introduce the testbed setup and benchmark programs. Micro-benchmarks are then used to evaluate the overhead associated with the VM hosting cluster. Finally, we evaluate the effectiveness of the DPS scheduling for a few representative scenarios.

5.5.1 Testbed Setup

We constructed a testbed to evaluate the performance of the proposed VM hosting cluster and the DPS scheduler. The VM hosting cluster consists of 4 Dell Dimension 4550 PCs as host machines, each equipped with a Pentium-4 2.66GHz CPU and 1GB of main memory. All host machines are interconnected via a Fast Ethernet switch. Xen 2.0.7 is installed and takes up around 18MB of memory. About 110MB of memory is designated to domain 0, where the `xend` and the DPS daemon are running. With this setup, up to 896MB of memory can be allocated to unprivileged domains (VMs). Another Dell PC (P-4 2.26GHz, 1GB memory) is dedicated to hosting a VM repository and the VM directory service. The VM repository is a software RAID-0 disk array exporting via NFSv3. The disk array is dedicated to this project and the utilization of this machine and network was very low during the evaluation.

We primarily use a PVM parallel program called `patterns` in the evaluation. The program `patterns` was developed for the Virtuoso project [42]. It models after bulk-synchronous parallel (BSP) style applications; many parallel benchmark programs, such as the NAS parallel benchmark (NPB), are also of this type. The program `patterns` can be configured to run with any number of nodes and produce the various topologies commonly used by parallel programs, such as n-D mesh, n-D toroid, reduction tree, n-D hypercube, all-to-all, *etc*. The length of program execution is controlled by the number of iterations. In an iteration, each node receives a message from one of its neighbors, repeats a simple computation for a number of elements, and sends a message to one of its topological neighbors. Both the amount of computation and the message size are configurable. The amount of computation is specified in the number of multiply-add, memory read, and memory write operations for each element, while the message size is specified in bytes. During the evaluation, we fixed the amount of computation to be 10^6 multiply-add operations, and read/write 1MB of memory. The message size is set to 1KB. These numbers are chosen such that the expensive I/O virtualization overhead, which we show below, is less dominant.

The original `patterns` program can only generate a uniform amount of computation workload on nodes. However, this may not represent the real-world workload. To model unbalanced workloads, we modified the `patterns` source code such that for node $i(i = 1 \ldots N)$, the amount of computation is scaled to $1/i$ of the specified amount (10^6 in the evaluation). The message size is kept unchanged. When running `patterns`, for each node we created a VM that uses 128MB memory and the standard PVM disk image as its execution environment.

We also created another serial program `eatcpu` to simulate a single-node CPU-intensive application. It runs at the lowest priority to avoid performance degradation of other processes in the same scheduling domain and merely consumes all CPU time available to it. This behavior is very similar to wide-area distributed computing projects such as SETI@home [3] or Folding@home [67]. While the `patterns` program has some CPU share allocated for each VM, guaranteeing a minimum number of CPU cycles it will received, the `patterns` program and the `eatcpu` program also compete for unallocated CPU cycles. Without competition, `patterns` can simply take whatever CPU cycles it

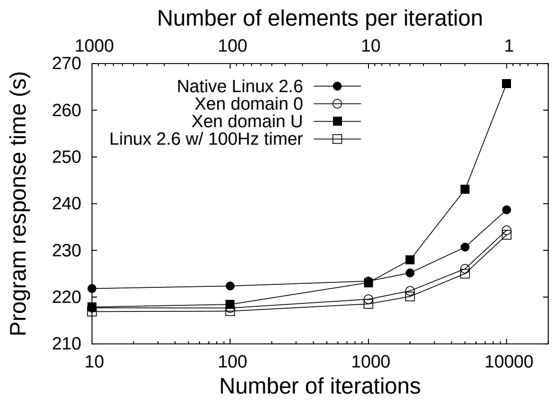


Figure 5.5 Xen virtualization overhead.

needs.

5.5.2 VM Hosting Overhead

We first examined the overhead associated with the execution of parallel programs in virtualized environments. The `patterns` program was executed in three different modes: *native*, *Domain-0*, and *Domain-U*. In native mode, the program was executed in Linux without any virtualization; this mode served as the baseline of our experiment. In Domain-0 and Domain-U modes, the program was executed in domain 0 and an unprivileged domain, respectively. In all cases, no other program was competing for any resource. We configured `patterns` to create a 4-node workload with the all-to-all topology, and each VM was hosted in a different host machine. We fixed the length of program execution to a total of 10,000 elements on each node but changed the number of elements at each iteration to generate different amounts of network I/O. The program response times in different modes are compared and plotted in Figure 5.5.

At first, the results didn't match our expectation as explained below. Although running in domain U is slightly slower than running in domain 0, the performance in native mode was actually *worse* than running in either domain 0 or domain U. Only when network I/O is frequent (few elements per iteration), the program response times were longer in domain

U, where the I/O virtualization overhead was dominant.

After many unsuccessful attempts of identifying the difference in configuration that leads to this counter-intuitive results, we suspected the kernel timer frequency is the culprit, even when there was no other task to schedule. We then lowered the frequency from 1000Hz (default in Linux 2.6) to 100Hz (default in Linux 2.4 and Xen Linux in both domain 0 and domain U) and the program response time was decreased by 2% (as plotted in Figure 5.5.)

With this fix, we observed that the CPU virtualization overhead is very small; it is about 0.4% and 0.6% for domain-0 and domain-U mode, respectively. Frequent network communication and a longer network packet processing path raises the overall overhead to 14% in domain-U mode [55]. We expect new hardware design will lower the I/O virtualization overhead.

Moreover, it took only 18 seconds to boot up the standard VM image, where only 2.5 seconds CPU time was used. We concluded that the virtualization overhead is small, and a proper kernel configuration can make a performance improvement in VM hosting cluster. By providing an optimized kernel for specific types of parallel programs, resources can be utilized more effectively.

5.5.3 Live-Migration

It has already been shown in [23] that live-migration can keep the downtime of moving a VM within a cluster in less than 1 second for most workloads. We also experimented with the live-migration of Xen VMs to assess its impact on parallel programs, especially on a fully-utilized host. In this experiment, we configured `patterns` to run on 3 VMs hosted in 3 different host machines. The fourth machine was used as the migration destination. On the machine that hosts node 1 of `patterns` in an unprivileged domain, we also ran a copy of `eatcpu` in domain 0. This configuration makes the host machine the bottleneck, and we can control the scheduler to allocate various amounts of CPU shares between these

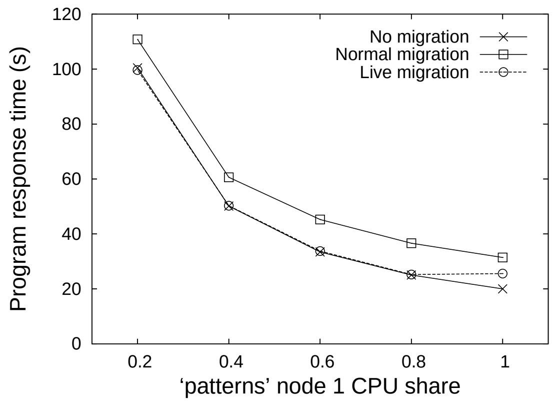


Figure 5.6 Xen VM migration overhead.

two programs.

We compared the program response times for three cases: no migration, one normal (stop-and-copy) migration, and one live-migration during the program execution (10 iterations of 100 elements). In order to make the result comparable to that of "no migration," we also ran a copy of `eatcpu` within the domain 0 of the migration destination. In addition, Xen does not migrate the scheduler parameters between hypervisors, and hence, we set the parameters as soon as the migration is completed. The result is plotted in Figure 5.6.

The program response time is inversely proportional to the CPU share it has been given (node 1 is the bottleneck node). From the figure, it can be easily seen that normal migration takes 10–12 seconds to migrate a 128MB VM. The migration time was limited by the network bandwidth (Fast Ethernet in our case) and proportional to the configured VM memory size. The effect on the program response time also depends on how other running VMs dealing with an unreachable node. On the other hand, live-migration incurs essentially no overhead to the program response time. The only exception is that when the host machine was dedicated to `patterns` (no `eatcpu` was running in domain 0). Although live-migration did not incur any overhead to the program response time, it actually took longer, 24 seconds in this case, to migrate the VM. The transit time also depends on how frequently the memory pages become dirty. Since parallel programs tend to modify many memory pages in a short period of time, this supports our design decision of migrating the least-utilized VM from a bottleneck host machine.

5.5.4 DPS Scheduling

The strength of the DPS scheduling can best be shown in a resource-competing environment with unbalanced workloads. The testbed is set up in such a way that 50% of CPU time on each node has already been booked, and only the remaining 50% of CPU time is available for competition. In the absence of competition, even a program with a small share of CPU time can use a full CPU under a work-conserving scheduler. For the same reason, competing with a light background workload does not make sense either. We deployed `eatcpu` in domain 0 of each host machine to create this immovable background workload.

On top of this background workload, we executed `patterns` to create a 4-node workload with the linear topology (1-D mesh). The linear topology is chosen so that we may also compare the fully-distributed and bank-assisted share-exchange strategies. The share-exchange was configured to be performed once every 5 seconds for this workload (10 iterations of 100 elements). Conducting the share exchange too often would make the two strategies less distinguishable. On the other hand, infrequent share-exchanges would make the potential benefits of DPS less visible. For the fully-distributed strategy, we set the withholding factor wh_i to $1/(P+1)$, *i.e.*, the withheld amount is the same as the amount to transfer to each upstream VM.

The `patterns` were submitted with a range of resource specifications, from 0.1 to 0.5 CPU per node, to see how DPS handles different scenarios. We compared the program response times with and without DPS, and plotted the results in Figure 5.7.

Without DPS, the CPU shares were fixed and evenly distributed on all host machines. The program response time gets prolonged when fewer CPU shares were given. Note that a 0.1 CPU share only means that at least 10% CPU time is available for this specific VM,

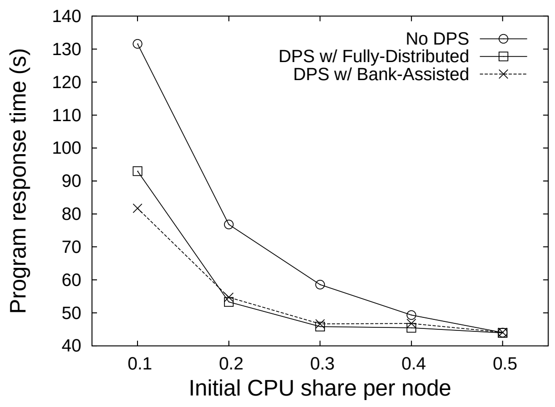


Figure 5.7 DPS reduced the program response time by up to 38%.

and therefore, the response time was not 5 times that of executing with 0.5 CPU share.

With an initial CPU share of less than 0.5 per node, there was some unallocated CPU time that DPS can exploit. After a few share-exchanges, CPU shares were redistributed to match the workload. A CPU-share trace is plotted in Figure 5.8.

From Figure 5.7, we can easily see that DPS significantly reduces the program response time. The user can submit a program with less CPU time budget (*e.g.*, a total of 1.2 CPUs) while having a response time very close to a more costly submission (*e.g.*, a total of 2.0 CPUs). The performance of DPS also improves with uncommitted resources. When only 60% of CPU time is allocated, DPS with the bank-assisted share-exchange strategy cuts down the response time by nearly 38%.

The performance of DPS also depends on the underlying share-exchange strategy. The bank-assisted share-exchange strategy can redistribute the excess shares from one end of the linear topology to the other within one redistribution cycle, while the fully-distributed strategy needs 3 cycles. This effect is pronounced, particularly when only 0.1 CPU-share was allocated to each node initially. In this case, the most demanding node 1 has enough room to accommodate excess shares from the least utilized (also the farthest), node 4. Therefore, the benefit of the banker process can be seen easily. We also repeat the same configuration with the all-to-all topology. Since all nodes can be reached from each other in

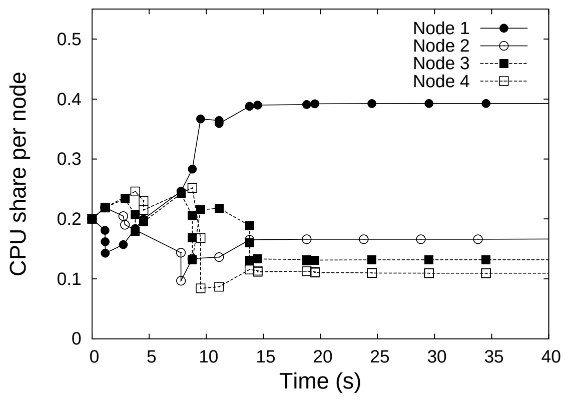


Figure 5.8 CPU share redistribution. Each node was given 0.2 CPU share initially and DPS with fully-distributed share-exchange strategy was able to redistribute CPU shares to match workload distribution in 3 cycles.

one hop, the benefit of the banker process disappeared. We also monitored the CPU usage of the DPS daemon and found the overhead of DPS to be negligible.

In addition, we note that the benefit of DPS is obtained from the redistribution of CPU shares. The response time of `patterns` is reduced because the bottle-neck VM gets more CPU cycles from the unallocated CPU time slot pool. The progress of `eatcpu` is slowed during the execution of `patterns` because it gets less "free" CPU time from the same pool. However, `eatcpu` still gets at least 0.5 CPU share so the scheduler does not violate any agreed arrangements. The net effect is that the time slots that used by `patterns` are more closely-packed and `eatcpu` gets more time slots once `patterns` is finished.

5.5.5 Workload Redistribution

DPS redistributes CPU shares at run-time to match resource demands, but each host machine has a limited capacity and may not be able to accept every share transfer. When this happens, CRM (VM directory in our implementation) can shed some workload via VM migration.

We set up two copies of the `patterns` program, each copy was configured to create a 4-node workload. We artificially aligned the program copies such that two VMs of the same node number are initially hosted in the same machine. We executed the program for a longer period of time (100 iterations of 100 elements) so CRM may have enough time to do migrations.

Without VM migration, the host that accommodates two copies of VM_1 quickly becomes the bottleneck. Both copies of `patterns` took 430 seconds to complete. With VM migration, one copy of VM_1 was migrated to the least utilized host machine (that hosted two copies of VM_4) soon after the start of the experiment, and the two copies of `patterns` were finished in 219 and 316 seconds, respectively. While the workload was not perfectly balanced at the end, this performance gain demonstrates the effectiveness of workload redistribution.

5.5.6 Large-Scale Simulation Results

In order to demonstrate the capability of DPS on a larger cluster, we built a simulator to simulate running BSP-style applications on a cluster with a proportional share CPU scheduler. It uses the credit CPU scheduler from Xen 3 [1]. The inputs to the simulator include the number of physical hosts and a number of BSP programs. The specification of each simulated BSP program includes the number of of VMs that it needs, the initial placement, and the actual CPU demand for each VM. The actual resource demand is only known to the node-level CPU scheduler for simulation purpose but concealed from DPS. Dependencies between VMs can also be given as inputs. While the simulator does not simulate network latency, it enables upstream VM inference in DPS. All programs start at the same time and the simulator reports the response time of each program.

We first use the simulator to show the benefits of time-multiplexing BSP programs and how much DPS reduces program response times. We simulate a 16-node cluster with a variable number of 16-VM programs. Each program has the same structure (using 16 nodes) but different actual CPU demand. The actual CPU demand of each VM is a random number uniformly distributed between 10ms to 100ms per BSP iteration. The actual CPU demand

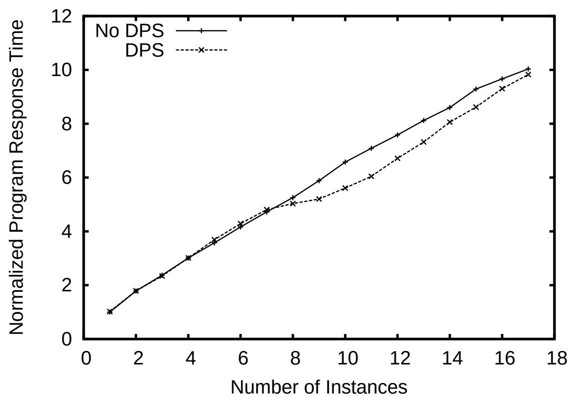


Figure 5.9 DPS reduces program response times in a cluster of increasing number of parallel programs.

of each VM is also different in each instance of the program. Each VM of a program is also assigned 0.1 CPU share initially and is placed on a different physical host, so each 16-VM program use all 16 nodes in the cluster. In other words, if we run N instances of the 16-VM program, each physical host has N VMs and $N/10$ of CPU time been assigned. Statistically, the workload of each physical host is identical. When $N = 1$, the cluster is dedicated to only one instance of the program and we normalized response times to this number.

From Figure 5.9, we can see that the response time increases at a slower rate with the number of instances of the 16-VM program. It is because faster VMs (lower CPU demand VMs) need to wait for slower VMs (higher CPU demand VMs) and cannot fully utilize its resources. In this experiment, the CPU time utilization for the program is 56%. Therefore, we can overbook the cluster with 17 instances of the program running simultaneously and the response time is roughly 10 times of running on a dedicated cluster, which is what the cluster user expecting when submitting the program with 0.1 CPU share for each VM. We use the bank-assisted share exchange in this simulation.

When there are 8 or fewer VMs on each physical host, the response time is the same with or without DPS. It is because the CPU is not fully allocated (only 0.1–0.8 CPU is booked) and a work-conserving scheduler is being used, each VM actually gets more than

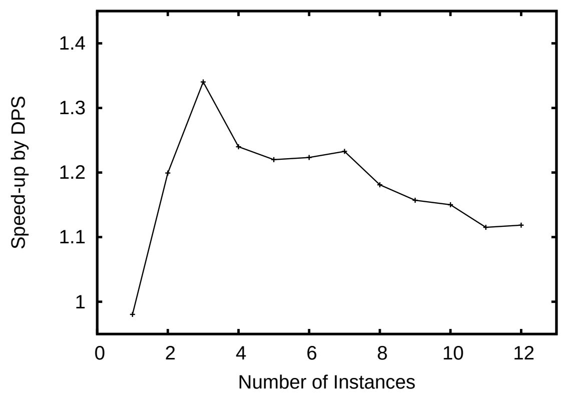


Figure 5.10 The speed-up of program response time by utilizing DPS.

its share and the need of DPS is nullified. When there are more than 8 instances of the program, we can see that DPS is able to reduce program response times by up to 15%, or speed up by up to 17%. Maximum benefits are achieved when there are 10–11 instances of the program, *i.e.*, 0–10% overbooking or about 60–70% actual CPU utilization. We note that if each instance is run one after another on a dedicated cluster, for the same amount of time only 6–7 instances can be completed instead of 10–11 instances.

The benefit of DPS reduces as the number of instances continues to increase. It is because there is less and less idle time in the cluster after extensive overbooking. However, one cannot simply rely on overbooking to improve utilization as the capability of a program using CPU time can also change from time to time. If a program suddenly can utilize every CPU cycle it is given, it cannot recoup enough CPU time it is entitled to. Penalties might be resulted if resource provider cannot meet the agreement. The appropriate amount of overbooking depends on the workload and allocation of each VM. Finding the optimal extent of overbooking is out of the scope of this work.

In another experiment, we show how the levels of multiplexing affecting DPS. The same program is used but each CPU is always fully allocated no matter how many instances of the program are running. In other words, when N instances are running, each VM is entitled $1/N$ CPU share initially. After excess shares are transferred, VMs on overbooked physical hosts are migrated to underloaded physical hosts to balance workload distribution.

From Figure 5.10, we can see that the benefit of DPS reaches a maximum when there are 3 instances of the program running with 0.33 CPU share initially for each VM. DPS is able to speed up the execution by 34%. Similarly, as the number of instances increases, the benefit of DPS is diluted because of simple multiplexing. DPS performs better when there are a fewer number of programs with larger chunks of CPU allocations instead of many programs with little CPU allocations.

5.6 Summary

Enhancing utilization as high as possible has always been a goal for computing cluster operators. While some parallel programs may place balanced workloads on computing nodes, most of workloads do not, leaving the allocated resources idle (hence wasted) for an extended period of time. An important source of this mismatch between resource allocation and consumption is found to be the over-simplified program submission model, where program and resource specifications are bound *together*. The proposed DPS scheduler makes it possible to decouple these two aspects, and is experimentally shown to yield considerable benefits.

The use of VM enables each hosted parallel program to be executed in a dedicated environment while the operator can easily move resources around. In addition to creating VM disk images for a specific execution environment, kernel configuration can also be customized to best fit the hosted application. With VM migration, the flexibility of allocating and utilizing resources in a VM hosting cluster greatly improves the limits of a traditional time-sharing cluster. With emerging hardware virtualization, this is also expected to benefit even I/O-intensive programs.

Another benefit of DPS scheduling is the provision of explicit resource guarantees, which is the fundamental requirement of scheduling real-time applications in computing clusters. Once a program is submitted for execution, it is guaranteed to receive the specified amount of resource, even with migration during its execution. For a utility cluster, such an underlying characteristic is essential; otherwise, no service-level agreement can be realized. The DPS scheduler is also distributed and scalable. We believe DPS can be extended to other resources as well.

Chapter 6

Accurate Packet Timestamping for Game Servers

6.1 Introduction

Virtualization has been used extensively for server consolidation in order to improve hardware utilization and reduce operating cost. While some resources in a computer system are space-partitioned and allocated to each virtual machine (VM), other resources are time-shared between VMs, incurring an additional scheduling delay that can reduce the responsiveness of services running in VMs. Although the prolonged response time generally does not affect throughput-oriented services such as web servers, online gaming, an emerging Internet application, is very sensitive to such delays.

Different genres of online games have different requirements on server responsiveness and first-person shooter (FPS) is the type of games that is most sensitive to the latency between a game server and its clients. In FPS games, a dead reckoning mechanism is typically employed to hide the latency by predicting the movement of all entities in a game. Previous studies have shown that players are able to notice sluggishness when the network latency is above 75ms and the hit accuracy of precision shooting decreases steadily when the latency is 100ms or more [12]. Similarly, gamers are also affected by long input lags on certain LCD monitors, which can be as high as 70ms in some instances [1]. In other words, by reducing the latency, online gamers can have a better gaming experience.

[1]Dell 2408WFP Review. http://www.tftcentral.co.uk/reviews/dell_2408wfp.htm

There are many game server hosting companies on the Internet competing to provide the lowest cost game servers for hardcore online game players who look for short network response time, fast hardware, and the ability to customize a game server installation. With the aforementioned response time requirement, game server hosting service providers are on the horns of a dilemma. They want to reduce their operating cost by consolidating as many game servers in each physical host as possible without violating the service quality requirement. Virtualization should not only bring the ease of consolidation to service providers, but also provide an adequate interface for the hosted application. In this work, we look into game servers hosted in a virtualized environment and augment the VM abstraction to improve game server performance.

When user inputs arrive at a game server as incoming packets, a typical game server takes a timestamp immediately after receiving a packet and uses the timestamp as the time that the user makes the move. The time is then used for dead reckoning calculation to determine the state of the entities involved before the user input is applied to the game state. Periodically, the game server sends updates of the game state to all clients so all users have a consistent view of the game world. Obviously, an accurate timestamp reflects the user's intention better, provides higher fidelity, and results in better gaming experience.

In a virtualized environment, however, a guest OS generally has no or very limited access to the host platform. For example, a guest OS can access a virtual disk device provided by the host platform, but disk I/O scheduling in guest OS is generally ineffective as the physical disk head location is unavailable to the guest OS. Likewise, while all VMs can synchronize to the host clock, the system time maintained in each guest OS becomes non-contiguous as timestamps can only be taken if the VM is actually running on at least one CPU.

The coarse-grained virtual time creates a couple of problems for game servers. First, the latency increases as it includes the time the host OS delivers the packet to a guest OS and the scheduling delay that a VM experiences. As our experiment shows, in a loaded system the latency can be as high as 40ms in the worst case. Second, the inter-arrival time between user inputs becomes obscure. For example, while two packets may arrive at the physical host 10ms apart, they may arrive at a VM virtually at the same time. Although the arrival order can be preserved, their effect on the game world can be very different.

To bridge the gap between the physical host and a VM, we propose to augment the virtual network interface used in VMs such that the packet arrival time can be taken in the host OS and passed to the guest OS if an application requests it. We modified the Xen virtual network device driver in the Linux kernel. Our evaluation result shows that the timestamp reflects the user input time better and is much more accurate than taking timestamps in the guest OS because the timestamp is taken in the very early stage of network packet processing.

The contribution of this work includes: (1) restore the capability of taking an accurate timestamp of packet arrival in VMs; (2) provide a prototype in Xen/Linux along with a game model for evaluation; and (3) investigate its applications beside online game servers.

The rest of the chapter is organized as follows. We first provide an overview of an FPS game model and an architecture of game-server hosting in Section 6.2. The packet delivery path in a virtualized system is described in Section 6.3. Our augmented virtual network interface design and implementation is detailed in Section 6.4 and 6.5, respectively. Evaluation results are presented in Section 6.6 before we summarize the chapter in Section 6.7.

6.2 Model

In this section, we first provide a brief history of FPS games and the need for hosting game servers inside VMs. An architecture of a game server hosting service is then described which is following by a problem statement.

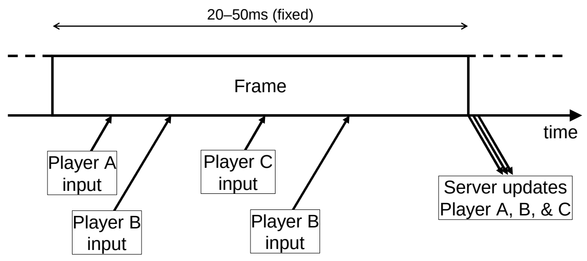


Figure 6.1 The interaction between game players and a server in a time frame.

6.2.1 Overview of FPS Games

While the history of FPS games can be traced back to 1973, its current form became well-known after *id Software* introduced *Doom*, the first multiplayer FPS game, in 1993. Doom allowed two to four players against each other over a local-area network (LAN) and became very popular. Many other popular FPS games, including *Quake*, *Unreal*, and *Battlefield* series, followed the same track where the newer ones allowed up to 64 players in a game session simultaneously.

When a game session is being held over a LAN, one of the player's computer (a PC or a game console) is usually assigned an additional role as the game server. Other players connect to the game server using TCP/IP, IPX, or other protocols. As this is the most common game-server setup, many game-server programs only support one session at a time. Geographically-dispersed gamers can also connect to a dedicated game server on the Internet and play remotely. More recently, emerging massively multiplayer online FPS (MMOFPS) games support more than 100 players in a battlefield.

The main function of a game server is to provide a consistent view of the game world to all players. Typically, The game time is partitioned into time frames. The length of each time frame is fixed during the game play and usually ranges 20–50ms (i.e., 20–50 frames per second.) The interaction between game players and a server is depicted in Figure 6.1. During each time frame, the server receives packets that represent player movements from clients. At the end of each frame, the server sends out updates to each client to inform them the current game state, essentially a snapshot of the game. Since updating the game world does not involve any graphics rendering, the server process itself does not demand much computation resources. When a game session is hosted over a LAN, a relatively powerful computer is usually chosen to take the additional workload.

Between two updates of the game world, each game client employs a dead reckoning mechanism to bring a continuously evolving game world to players. A dead reckoning mechanism uses a pre-defined model and the current game state, such as an entity's position, heading, velocity, acceleration, etc., to predict the entity's state and its interaction with user inputs. However, when all users input arrive and reconcile at the game server, some locally generated game state may be nullified.

A game server is also highly configurable. For example, it can support different game types (death-match, capture-the-flag, etc.) and scenarios (including third-party/customized ones), tweak update frequency, limit the number of players and the selection of weapons, anti-cheat plug-in, etc. A group of players may develop their preferred settings and only play accordingly. Unless two groups of players can all agree on a set of settings, two different servers are needed.

6.2.2 Game Server Hosting

As the number of players in a game session increases, it is more and more difficult to bring all players together and connect their equipment to a LAN. Playing games over the Internet becomes an attractive alternative. Game servers may be provided by the original game producer and hosted in the company's facility. A group of game players can also rent a server in a data center and install their customized server program. When a player wants to join a game, a list of available game servers is shown on the screen along with their latency, server configuration, etc. The player then joins a server with a short latency and preferred game settings.

In order to reduce the operating cost of running game servers, data center operators want to use as few physical servers as possible to host game-server programs. As many existing game-server programs can support only one game session at a time, multiple copies of a certain game may need to run on a hosting platform concurrently. However, a game may only listen to a certain TCP/UDP port and load its configurations from a certain fixed path. While modifying game-server source code and game protocols can relax these restrictions, all gamers must download new client software in order to continue playing the game, resulting in a prohibitive redistribution cost. Game developers may also find themselves a lack of incentives to supporting legacy games.

A simpler approach is to host each game server program in a virtual machine (VM) and consolidate multiple VMs on top of a physical server. Virtualization provides an illusion that each game-server program is running on a dedicated server so no redesign is required. A group of game players can rent a game server VM and apply their customizations and settings on it. Using virtualization also provides us the flexibility to migrate a game server from one physical server to another. Game server hosting service providers can use VM migration to balance the workload on its physical servers. A game server VM can also migrate to a data center where all players in that session have equal network delays. When a game server is not currently being used, the data center operator can also suspend the VM and resume it when needed.

However, any consolidation effort must also meet the timing requirements imposed by the application. FPS game players are very sensitive to the game-server's responsiveness because a short delay can easily influence the result of a duel. While the latency in a LAN is very short ($<$1ms), the latency of Internet game playing is much higher. For example, the network latency over a coast-to-coast (US) link is around 30–50ms. On top of the network latency is the latency incurred within the game server, including the delivery of incoming packets to VM and the VM scheduling delay. The goal of this study is to understand these delays and improve game-server performance in a virtualized infrastructure.

6.2.3 Problem Statement

Given the popular demand of Internet gaming and the need of game-server hosting, the goal of data centers is to host as many FPS game servers as possible on top of a virtualized infrastructure. The consolidation effort is constrained by a service-level agreement that limits the latency a user input may experience inside a data center. We also assume that the workload model of each server is identical and the host platform has a fixed amount of resources. Therefore, the number of hosted game servers is used as the metric in our evaluation.

6.3 Timing in Game Servers

As a game server and its clients are geographically dispersed, it is usually difficult to have their clocks synchronized with each other. Therefore, the gold master state of a game world must depend on a single clock source, the server clock, to serialize events generated from all clients. In this section, we first discuss various latencies in a virtualized system and how timestamps are taken in a guest OS.

6.3.1 Packet Delivery Delay

In current architecture, all network I/Os to and from VMs are handled by device drivers in the host OS. The packet delivery path is illustrated in Figure 6.2. For each incoming packet, the host OS typically uses its layer-2 MAC address, layer-3 IP address, or layer-4 port number to switch or route a packet to a VM. The packet header is overwritten on-the-fly to reflect different network settings. The host OS then uses the virtualization interface provided by the hypervisor to make the packet available to the destination VM. Once the packet is delivered, the host OS sends an event to wake up the destination VM. If the VM is not currently scheduled to run, the scheduler will schedule its execution.

We define the delay between the time that a packet is received by the host OS and the

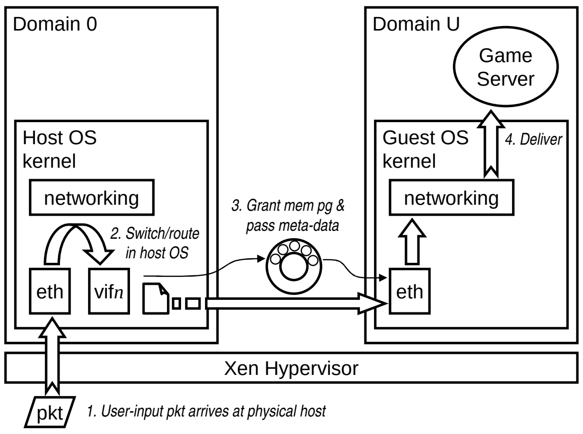


Figure 6.2 The packet delivery path for an incoming user-input packet.

time that a packet is received by a guest OS as the *packet delivery delay*. From the above discussion, the packet delivery delay includes the time a packet spent in the protocol stack in host OS, the mechanism that passes the packet from the host OS to a VM, and a scheduling delay before the guest OS can pick it up. Among these components, the scheduling delay is the most non-deterministic and is discussed below.

6.3.2 Scheduling Delay

In a non-virtualized environment, a game-server process is competing with other processes in an operating system (OS) for CPU time. The process scheduler in the OS has full control of CPU time allocation. The *process scheduling delay*, which is defined as the time between a process is ready to run and the time that it actually begins to run, is a function of the scheduling algorithm in use, other processes in the system, timer frequency, and so on. Typically, a game server only co-exists with some background daemon processes so the process scheduling delay is very short.

In a virtualized environment, a game server also needs to compete with other VMs in the physical server for CPU time, incurring an additional *VM scheduling delay*. The VM scheduling delay is defined as the time between a VM, or a process in the VM, is ready to run and the time that VM actually begins to run. Similar to the process scheduling delay, the VM scheduling delay is also a function of the underlying scheduling algorithm, the workload of other VMs, timer frequency, and so on.

The CPU scheduler in a hypervisor can increase the priority of a VM that was waiting for incoming packets or I/O operations to complete. This design improves the responsiveness of I/O-intensive VMs . However, if there are many I/O-ready VMs that need to be scheduled, a penalty is still imposed on the response time. Our evaluation result shows that the VM scheduling delay can be as high as 40ms in a game-server hosting scenario.

6.3.3 Timestamping Methods

Modern CPUs have a built-in timestamp counter that can supply high-resolution timestamps to a OS. During boot-up, the OS first initializes the system time with the hardware clock and also calibrates the timestamp counter using hardware timers. When current system time is needed, the OS reads the current value of the timestamp counter and converts the value to the system time in a certain format, such as `struct timeval` or `ktime_t`. Time resolution might be limited in the conversion process, e.g., 1μs for `struct timeval` and 1ns for `ktime_t`.

In a POSIX-compliant system, application programmers can use the `gettimeofday` to learn about current system time. Therefore, one can call the function immediately after receiving a packet from the OS and associate the timestamp with the packet. However, this method suffers from both packet delivery and scheduling delays. It also generates an excessive number of system calls and results in higher CPU demand as timestamps are required for most input packets. On the other hand, it is compatible to many systems and is the most portable approach.

In many recent OSes, one can specify the `SO_TIMESTAMP` option to an opened socket using the `setsockopt` system call. Timestamps are taken in the kernel and available to user-space applications via the `recvmsg` system call as ancillary data (also called control messages.) In Linux kernel, the timestamp is taken at the very beginning of device-independent network code. Although the time spent in network device drivers and the scheduling of kernel tasklets are still not taken into account, the timestamp is much closer to the packet arrival time. Both the network time protocol (NTP) daemon and the *tcpdump* network packet capturing program use this option to obtain timestamps.

While the option is enabled per socket, Linux kernel timestamps all incoming packets if any opened socket requests timestamps. This design values timestamp accuracy as one does not need to wait until a packet is associated with a socket to determine whether a timestamp is needed. Given that a large portion of game traffic requires timestamps, the extra time spent in timestamping all incoming packets becomes insignificant for game servers.

However, in a VM, the thus-obtained timestamp only reflects the time that a packet is handled by a guest OS. The packet delivery latency within a virtualized platform is not taken into account. This presents an information gap between a VM and its host system. For a game server, a longer latency and inaccurate timestamps reduce the fidelity of game playing.

6.3.4 Effects of Inaccurate Timestamps

When the packet delivery delay occurs between a client and a server, the dead reckoning mechanism uses an inaccurate timestamp to estimate game state before applying the player's command delivered in the packet. The net effect appears that the client took longer to make the decision. When the total latency is large, game players will feel sluggish as their inputs are not processed promptly.

Furthermore, the inter-arrival time between two user inputs, whether or not they come from the same user, also becomes obscure. While the two inputs may be 10ms apart when they arrived the host OS, they may be delivered to a VM in the same batch and handled by a guest OS almost at the same time. Although the order of packet delivery can be preserved, the result can be very different. If the two inputs were from the same user, the difference in

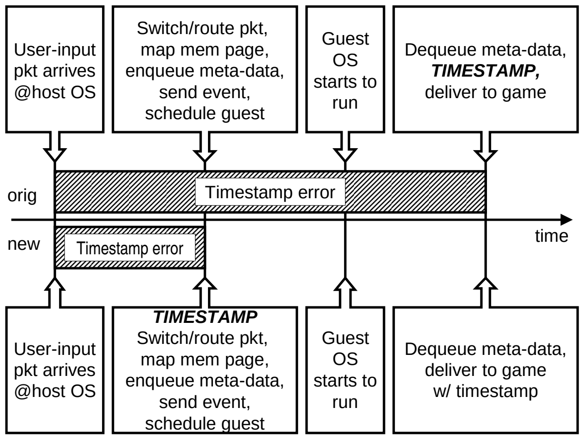

User-input pkt arrives @host OS
Switch/route pkt, map mem page, enqueue meta-data, send event, schedule guest
Guest OS starts to run
Dequeue meta-data, *TIMESTAMP,* deliver to game
orig
Timestamp error
new
Timestamp error
time
User-input pkt arrives @host OS
TIMESTAMP Switch/route pkt, map mem page, enqueue meta-data, send event, schedule guest
Guest OS starts to run
Dequeue meta-data, deliver to game w/ timestamp

Figure 6.3 A comparison between two different timestamping approaches.

inter-arrival time may not reflect the user's original intention. If the two inputs were from two users aiming each other with a gun, the difference in inter-arrival time may determine whether a gunshot hits a dodging opponent or not. Previous research also shows that relative delays between gamers may also impact gaming experience [59]. Because of these consequences, we need a better timestamping mechanism for game servers in VM.

6.4 Augmented Virtual Network Interface

In order to reduce the packet delivery delay, one can control the number of VMs hosted on a host platform so the VM scheduling delay can be minimized. An admission control mechanism can be used to determine if new game-server VMs can be started or running game servers need to be migrated to another host platform. However, this approach simply manages the additional latency but not fills the information gap.

To make accurate packet arrival timestamps available to VMs, we decide to record packet arrival times in the host OS and deliver the timestamp to the destination guest OS. A comparison of the timestamp error in these two approaches can be seen in Figure 6.3. While the packet delivery delay is not shortened with this approach, the timestamp error — the difference between the reported timestamp and the actual packet arrival time — is significantly reduced, so the effects of inaccurate timestamps can be reduced.

Similar to the `SO_TIMESTAMP` socket option that a process can use to request timestamps from kernel, we allow the guest OS to proxy that request to the host OS and take timestamps in the host OS instead. The timestamp is then passed to a VM along with other meta-data of a packet, such as the physical memory address that the packet resides, whether the packet has been check-summed in host OS, and so on. When the guest OS receives a timestamp-associated packet, it simply trusts the timestamp and passes the timestamp to the application — the game server. While the game server may process multiple user inputs in a batch, their timestamps faithfully represent their arrival times in the host OS and the resulting game state can be constructed with higher fidelity.

For each game packet, the timestamp is only taken once — the design simply moves the work from a guest OS to the host OS. We also value timestamp accuracy more by taking timestamps immediately after each incoming packet enters device-independent network code in the host OS. This approach avoids most of the packet delivery delay but additional work is done for packets that are not destined for game-server VMs. We argue that it is still a better approach as selective timestamping incoming packets can easily incur higher overhead.

6.5 Implementation

We implemented the augmented virtual network interface by modifying the Xen virtual machine monitor (VMM). Specifically, we modified the virtual network interface card (NIC) device driver to allow the host OS to pass timestamps to guest OSes. We also created a model FPS game to emulate the game-server workload without setting up a large number of game servers and clients. The detail of the implementation is described next.

6.5.1 Xen Virtual NIC

In order to reduce the cost of memory copying in delivering packets between VMs (called domains in Xen), the sending Xen virtual NIC typically grants accesses to the memory pages that contain the packet and only passes memory addresses to the receiving party. The memory addresses are passed in a circular buffer shared between a pair of virtual NICs. The circular buffer is implemented as an array of fixed-sized records. The size of each record is 8 bytes. A record may represent a packet that is pending to be received at the other end. Additional records may be used to deliver meta-data. We defined a new record type and use it to deliver the arrival time of a packet.

Unfortunately, only 6 bytes of an 8-byte record can be used to pass the timestamp, the other 2 bytes are header fields. The timestamp in Linux 2.6.18, which is both the host and the guest OS in our implementation, is stored in the `timeval` structure. The structure stores the number of seconds since the UNIX epoch and the number of microseconds passed the second, hence having a resolution of 1μs. In a well-formed timestamp the value in the second field is always between 0 and $10^6 - 1$ (inclusive.)

In order to fit the timestamp into the 6-byte payload, we reduced the timestamp resolution to 16μs ($2^4\mu$s) so the “microseconds” part of a timestamp needs only 16 bits (2 bytes) instead of $\lceil log_2 10^6 \rceil = 20$ bits to represent all possible values. This implementation is effective because only one additional record is needed to deliver a timestamp and there is no need for any prior language or any dependency between packets. The 16μs resolution provides enough details for online gaming. If a higher resolution is needed, one can use multiple records or use a common base timestamp between multiple packets.

6.5.2 Model FPS Game

We first investigate the source code of an open-source network FPS game, Urban-Terror, to understand the logic of a game server. The game is based on the game engine of Quake

3, a very popular FPS game. The game uses UDP/IP for network gaming so players can connect to a game server and play from a remote location. Similar to other games, the game server supports only one session at a time.

When a user makes a move in the game, a packet is sent from the client to the sender to propagate the state change. Every 50ms the server sends an update of the game world to each client. The traffic between an Urban-Terror game server and its clients is not encrypted.

While a large installation of game-server VMs can be set up by cloning VMs, it is much more difficult to set up a large number of clients and generate game playing traffic to game servers. Instead, according to the traffic characteristics of Urban-Terror and other studies, we built a model game server and client that help us evaluate large-scale deployment scenarios.

We expect future games contain more details than current games. Previous studies have shown that both the average packet size and packet rate are increasing in more recent games [44]. As the market size of serious gaming becomes larger, we also expect more sophisticated encryption algorithm will be used in online gaming. There are already security software add-ons for Quake 3 that encrypt game traffic to prevent cheating.

Based on these observations, we employed a fixed 256-byte data block to represent user inputs, which are encrypted before sending to the server. Along with a few header fields and message digest, the final UDP payload size is 296 bytes — similar to the average packet size of 247–270 bytes reported for *Halo 3*, a newer and also popular network FPS game [44]. The emulated user inputs follow the Poisson process and the average rate is set to 20 inputs per second, which is also similar to *Halo 3* (15–30 inputs per second). Game-state updates are sent out periodically and the frequency is set to 20 updates per second (the default value as in Quake 3) in our evaluation.

6.5.3 Measuring Packet Delivery Delay

We measure the packet delivery delay by comparing the packet arrival time on the game server to the user input time on the game client. Before sending each user input packet to the game server, a timestamp is taken on the game client and sent to the game server in a header field along with a sequence number. After the server received the packet, it subtracts the client timestamp from the packet arrival time. Since the two timestamps are based on different clocks, we synchronize the two clocks before each experiment.

The resulting time difference is the time a packet spent in sender's protocol stack, transmission delay, and the packet delivery delay at the receiving end. Of these components, the first two are not affected by different timestamping approaches in the game server so we can measure the difference in the packet delivery delay.

6.6 Evaluation

In this section, we primarily use the model game server and client we created to evaluate the packet delivery delay in a VM-based game server hosting scenario. The testbed setup is described first which is followed by evaluation result.

6.6.1 Testbed

We set up a testbed that consists of two machines. Each machine has two dual-core CPUs and 4GB memory which are connected via gigabit Ethernet. One of them is configured with Xen VMM and our modified virtual NIC driver. Since our model game server doesn't use much memory, each game-server VM is only allocated with 1 virtual CPU and 32MB memory. Only a guest OS and our model game server are running within each VM. The smaller memory footprint allowed us to host many VMs without running out of memory. The other machine runs our model game client and measures the latency to the server.

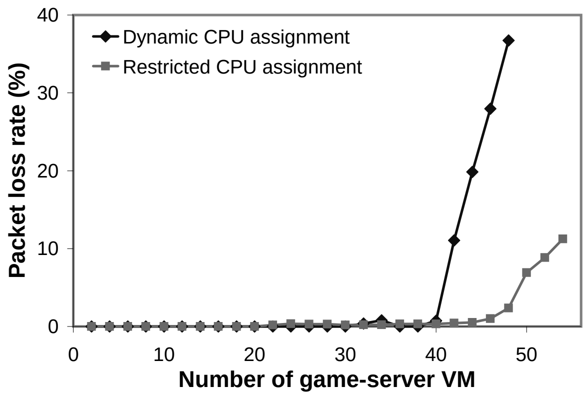


Figure 6.4 Packet loss ratio for different number of game-server VMs in different configurations.

6.6.2 Packet Loss Rate-Based Scalability

We first evaluate the scalability of the game-server hosting infrastructure by hosting as many game-server VMs as possible. We created two slightly different setups and used the packet loss ratio as the metric. In the first setup, the domain 0 (host OS) and all game-server domains (guest OSes) are allowed to use any of the 4 CPU cores in the system. In the second setup, the domain 0 is pinned to one CPU core and all game-server domains are allowed to use any of the 3 other CPU cores.

From Figure 6.4, we see that the packet-loss ratio shoots up when there are more than 40 domains if all 4 CPU cores are dynamically assigned to any domains (labeled *Dynamic CPU assignment* in the figure). On the other hand, if one CPU is dedicated to domain 0 (labeled as *Restricted CPU assignment* in the figure), 46 and 48 game-server domains can be hosted with $<$1% and $<$3% packet loss ratio, respectively.

The restricted CPU assignment scheme works better because all incoming packets need to be switched or routed via domain 0 before they can reach their final destination. When domain 0 could not obtain enough resources, incoming packets are dropped due to limited buffer space. If the packet-loss ratio is the only criteria in a service-level agreement, the hosting service provider may conclude that the system can host 46 active game sessions after this experiment.

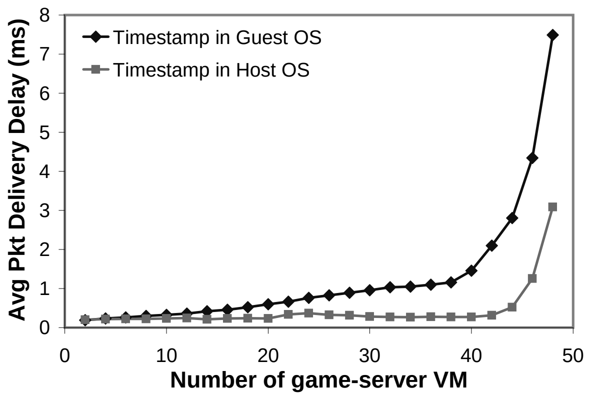


Figure 6.5 Average packet delivery delay for different number of game-server VMs using different timestamping approaches.

6.6.3 Packet Delivery Delay-Based Scalability

With the restricted CPU assignment, we look at the average packet delivery delay obtained by different timestamping approaches. As we can see in Figure 6.5, the average packet delivery delay increases with the number of game-server domains. If the timestamp is taken after the packet arrives in the guest OS, the delay is much higher than taking timestamp in the host OS. When the server is loaded with 46 domains, our design can reduce the delay by 3.1ms. In other words, the obtained timestamp is 3.1ms closer to the actual user input time.

If the game server requires a timestamp to be taken within 1ms after a packet is arrived, 44 domains can be hosted with our new design while only 30 domains can be hosted when timestamps are taken in each guest OS — a 47% increase in scalability.

In addition to the average packet delivery delay, the 95-percentile of packet delivery delay can also be used as an indicator of service quality. We plotted the results in Figure 6.6. We note that it can take more than 20ms to deliver a packet to a guest OS when 46 domains are hosted, the difference between two approaches can be as high as 15ms. When 44 or fewer domains are hosted, the time that a packet spent in the host OS is essentially constant. On the other hand, the 95-percentile of packet delivery delay distribution increases significantly if more than 40 domains are hosted. This shows that the improved timestamp

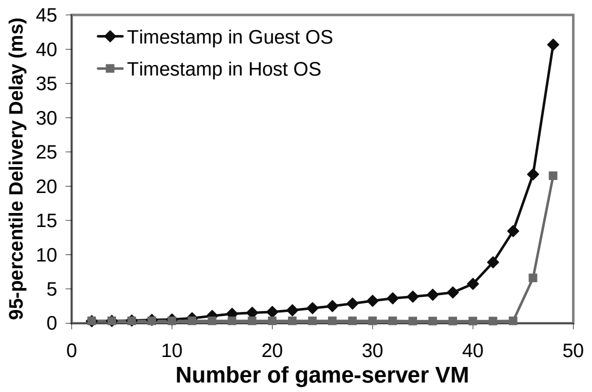


Figure 6.6 95-percentile of packet delivery delay for different number of game-server VMs using different timestamping approaches.

approach not only performs better if a certain service level is required, its performance is only bounded by the speed of the CPU that domain 0 runs on.

6.7 Summary

In this chapter, we present a design that bridges an information gap between a physical host and a virtual machine. The timestamps are particularly important to time-sensitive applications such as online gaming, an emerging market in the entertainment sector. We showed that while simple consolidation approaches can host 30 game servers subject to a timestamp accuracy constraint, with our improved interface one can host 44 game servers.

Chapter 7
Conclusion

Virtualization is an old concept that has brought back recently with the primary goal of improving x86 server utilization. In this thesis, a distinct feature of a virtual machine, namely the flexibility in run-time resource allocation adjustment, is utilized to improve the design of software systems on large-scale virtualized infrastructures, such as data centers. We showed that by incorporating this feature in software system designs, better performance can be achieved with the same amount of resources.

We first built Vibra Web, a web server cluster based on a virtualized infrastructure. Instead of balancing the workload of back-end web servers, we used a scalable and locality-aware request dispatcher at the front-end and tackled load imbalance by transferring network bandwidth allocation between back-end server VMs. Our evaluation results showed that average response time is reduced in Vibra Web and it is also more scalable than other locality-aware approaches.

Resource re-allocation schemes can also be integrated with fault-tolerant mechanisms to improve system performance during degraded operations. Surviving servers in a cluster can be granted more resources to compensate for the failed server. We also showed that by bridging a gap between a physical machine and a VM, the system clock can still be available to a VM even when the VM is not running, resulting in more accurate timestamps and better online gaming experience.

We also applied the idea of transferring virtual resources to parallel computing applications and built a distributed proportional-share (DPS) CPU scheduler. By transferring

CPU share allocations between computing nodes, a parallel application can complete its execution faster. It also relieves programmers from balancing computing workload.

In traditional operating system designs, we have seen that many designs were based on certain features at the hardware/software interface. Similarly, in this thesis, we showed that understanding the VM abstraction and design software systems accordingly can also make significant improvements in performance. We expect that there will be more systems that can be designed and perform better in virtualized infrastructures.

In the future, we expect more cores and larger memory in computer systems of all sizes. A more sophisticated hypervisor is needed to schedule hundreds of VMs, each with tens GB of main memory, on a single server with tens of cores and hundreds GB of physical memory. While the number of VMs that need to be scheduled is similar to the number of processes in a typical system, the size of each VM is roughly 2 orders of magnitude larger than a typical process, making memory allocation in NUMA systems a bigger problem.

We also expect the advent of more specialized, virtualization-capable I/O devices, such as newer 1Gbps and 10Gbps NICs. While their introduction will reduce the work that needs to be done in the host OS, the NICs also become stateful and may hamper compatibility of VM migration. New classes of computing resources, such as general-purpose computation graphics processing units (GPGPU) in advanced video cards, should also be made available to VMs while complicating VM abstraction. That is, keeping a small footprint and a small attack surface of a hypervisor will become more and more difficult. With a richer VM abstraction and more types of resources supporting dynamic allocation at run-time, it also brings more opportunities to optimize VM performance in a heterogeneous environment.

Bibliography

[1] Emmanuel Ackaouy. The xen credit cpu scheduler. Presentation in the Xen Summit of Fall 2006, September 2006.

[2] Saurabh Agarwal, Andy B. Yoo, Gyu Sang Choi, Chita R. Das, and Shailabh Nagar. Co-ordinated coscheduling in time-sharing clusters through a generic framework. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pages 84–91, December 2003.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *Proceedings of International Conference on Parallel Processing*, pages 547–554, 2003.

[5] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Labs, Oct. 1999.

[6] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1999. USENIX Association.

[7] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *SIGMETRICS ’00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 90–101, New York, NY, USA, 2000. ACM Press.

[8] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based networks servers. In *ATEC’00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association.

[9] Andrea C. Arpaci-Dusseau and David E. Culler. Extending proportional-share scheduling to a network of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[10] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst.*, 19(3):283–331, 2001.

[11] 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 *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[12] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, New York, NY, USA, 2004. ACM.

[13] Richard B. Bunt, Derek L. Eager, Gregory M. Oster, and Carey L. Williamson. Achieving load balance and effective caching in clustered Web servers. In *Proceedings of the 4th International Web Caching Workshop*, pages 159–169, 1999.

[14] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

[15] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, Oct. 2002.

[16] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 1–14, New York, NY, USA, 2007. ACM.

[17] Wu chang Feng, David Brandt, and Debanjan Saha. A long-term study of a popular mmorpg. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 19–24, New York, NY, USA, 2007. ACM.

[18] Wu chang Feng, Francis Chang, Wu chi Feng, and Jonathan Walpole. A traffic characterization of popular on-line games. *IEEE/ACM Trans. Netw.*, 13(3):488–500, 2005.

[19] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, 2001.

[20] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Understanding the characteristics of internet short video sharing: Youtube as a case study. In *Quality of Service, 2007 Fifteenth IEEE International Workshop on*, June 2008.

[21] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real:resource allocaiton challenges in virtual machine based it environments. Technical Report HPL-2007-25, HP Labs, February 2007.

[22] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, Andy B. Yoo, and Chita R. Das. Coscheduling in clusters: Is it a viable alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)*, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005.

[24] Anthony Cricenti and Philip Branch. Arma(1,1) modeling of quake4 server to client game traffic. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 70–74, New York, NY, USA, 2007. ACM.

[25] Francisco Matias Cuenca-Acuna and Thu D. Nguyen. Cooperative caching middleware for cluster-based servers. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[29] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *COMPCON '96: Proceedings of the 41st IEEE Int'l Computer Conference*, Washington, DC, USA, Feb 1996. IEEE Computer Society.

[30] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM symposium on Operating systems principles (SOSP'99)*, pages 261–276, New York, NY, USA, 1999. ACM Press.

[32] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS'96)*, pages 25–36, New York, NY, USA, 1996. ACM Press.

[33] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 2003. USENIX Association.

[34] Dror G. Feitelson. Packing schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'96)*, pages 89–110, London, UK, 1996. Springer-Verlag.

[35] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[36] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, New York, NY, USA, 1997. ACM Press.

[37] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang scheduling with lightweigth user-level communication. In *Proceedings of 2001 International Conference on Parallel Processing (ICPP'01), Workshop on Scheduling and Resource Management for Cluster Computing*, pages 339–348, Valencia, Spain, September 2001. IEEE Computer Society.

[38] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, New York, NY, USA, 2005. ACM.

[39] W. Gentzsch. Sun grid engine: towards creating a compute power grid. In *Proc. First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[40] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, New York, NY, USA, 2007. ACM.

[41] Ajay Gulati, Arif Merchant, Mustafa Uysal, and Peter J. Varman. Efficient and adaptive proportional share i/o scheduling. Technical Report HPL-2007-186, HP Labs, November 2007.

[42] A. Gupta and P. Dinda. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Policies for Parallel Processing*, June 2004.

[43] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, Oct. 2005.

[44] Szabolcs Harcsik, Andreas Petlund, Carsten Griwodz, and Pål Halvorsen. Latency evaluation of networking mechanisms for game traffic. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 129–134, New York, NY, USA, 2007. ACM.

[45] Yousuke Hashimoto and Yutaka Ishibashi. Influences of network latency on interactivity in networked rock-paper-scissors. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 23, New York, NY, USA, 2006. ACM.

[46] Eric Van Hensbergen and Athanasios E. Papathanasiou. Knits: Switch-based connection hand-off. In *IEEE Infocom*, 2002.

[47] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly efficient gang scheduling implementation. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM) (Supercomputing'98)*, Washington, DC, USA, 1998. IEEE Computer Society.

[48] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[49] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating system (ASPLOS-XII)*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[50] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[51] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[52] Jaecheol Kim, Jaeyoung Choi, Dukhyun Chang, Taekyoung Kwon, Yanghee Choi, and Eungsu Yuk. Traffic characteristics of a massively multi-player online role playing game. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8, New York, NY, USA, 2005. ACM.

[53] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[54] Bin Lin and Peter A. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[55] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (VEE'05)*, pages 13–23, New York, NY, USA, 2005. ACM Press.

[56] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. Alternatives to coscheduling a network of workstations. *Journal of Parallel and Distributed Computing*, 59(2):302–327, November 1999.

[57] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[58] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 205–216, New York, NY, USA, 1998. ACM.

[59] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 152–156, New York, NY, USA, 2004. ACM.

[60] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[61] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[62] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM.

[63] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[64] Debanjan Saha, Sambit Sahu, and Anees Shaikh. A service platform for on-line games. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 180–184, New York, NY, USA, 2003. ACM.

[65] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36:377–390, 2002.

[66] Anees Shaikh, Sambit Sahu, Marcel Rosu, Michael Shea, and Debanjan Saha. Implementation of a service platform for online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 106–110, New York, NY, USA, 2004. ACM.

[67] Michael Shirts and Vijay S. Pande. Screen savers of the world unite! *Science*, 290(5498):1903–1904, 2000.

[68] S. Singhal, M. Arlitt, D. Beyer, S. Graupner, V. Machiraju, J. Pruyne, J. Rolia, A. Sahai, C. Santos, J. Ward, and X. Zhu. Quartermaster - a resource utility system. In *Proc. 9th IFIP/IEEE International Symposium on Integrated Network Management IM 2005*, pages 265–278, 2005.

[69] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, pages 234–244, New York, NY, USA, 2000. ACM Press.

[70] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic coscheduling on workstation clusters. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'98)*, pages 231–256, London, UK, 1998. Springer-Verlag.

[71] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *ACM SIGOPS EUROSYS 2007*, March 2007.

[72] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[73] David G. Sullivan and Margo I. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 337–350. USENIX Association, 2000.

[74] Chang-Hao Tsai, Kang G. Shin, John Reumann, and Sharad Singhal. Online web cluster capacity estimation and its application to energy conservation. *IEEE Transactions on Paralled and Distributed Systems*, 18(7):932–945, July 2007.

[75] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, 2002.

[76] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[77] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, 1995.

[78] Kun-Lung Wu and Philip S. Yu. Load balancing and hot spot relief for hash routing among a collection of proxy caches. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 536, Washington, DC, USA, 1999. IEEE Computer Society.

[79] Takahiro Yasui, Yutaka Ishibashi, and Tomohito Ikedo. Influences of network latency and packet loss on consistency in networked racing games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8, New York, NY, USA, 2005. ACM.

[80] Ronghua Zhang, Tarek F. Abdelzaher, and John A. Stankovic. Efficient tcp connection failover in web server clusters. In *INFOCOM '04: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1219–1228, March 2004.

[81] Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, and Margo Seltzer. Hacc: an architecture for cluster-based web servers. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 16–16, Berkeley, CA, USA, 1999. USENIX Association.

[82] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.