2008-12-10

Dynamic Load Balancing of Virtual Machines Hosted on Xen

Terry Clyde Wilcox
Brigham Young University - Provo

DYNAMIC LOAD BALANCING OF VIRTUAL MACHINES HOSTED ON XEN

by

Terry C. Wilcox Jr.

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Terry C. Wilcox Jr.

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Phillip J. Windley, Chair
Date	Kelly J. Flanagan
Date	Tony R. Martinez

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Terry C. Wilcox Jr. in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Phillip J. Windley Chair, Graduate Committee

Accepted for the Department

Date	Kent E. Seamons Graduate Coordinator

Accepted for the College

Date	Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

DYNAMIC LOAD BALANCING OF VIRTUAL MACHINES
HOSTED ON XEN

Terry C. Wilcox Jr.
Department of Computer Science
Master of Science

Currently systems of virtual machines are load balanced statically which can create load imbalances for systems where the load changes dynamically over time. For throughput and response time of a system to be maximized it is necessary for load to be evenly distributed among each part of the system. We implement a prototype policy engine for the Xen virtual machine monitor which can dynamically load balance virtual machines. We compare the throughput and response time of our system using the cpu2000 and the WEB2005 benchmarks from SPEC. Under the loads we tested, dynamic load balancing had 5%-8% higher throughput than static load balancing.

ACKNOWLEDGMENTS

There are several people who helped me finish my thesis. I'd like to thank Loren for fixing the hardware I was using while I was in Texas. I'd also like to thank Shannon for helping me schedule my defense and giving me a clear roadmap for what I need to get done. Since I was out-of-state until right before the defense, I wouldn't have been able to schedule my defense without her help. I'm also grateful for the help given to me by the other graduate students in my lab. In particular, Devlin made my life much easier by lending me a laptop to use to prepare for the presentation. I am also grateful for the help, feedback, and assistance given to me by Dr. Windley. Finally, I want to thank my wife. She took extra responsibilities on herself to give me time to work. She also helped with the editing. She has read my thesis possibly more than anyone else (including me), and probably more than anyone ever will. Oh, and I can't forget Caleb. He helped! Thanks again to everyone.

Contents

Chapter 1

Introduction

During the past few decades, virtual machines have changed the way people use computers. Users are no longer restricted by the maxim that each physical machine will host only one operating system. By having multiple operating systems simultaneously running on the same hardware, users can tap into more of the physical resources of the machine. Hence, a machine's hardware has not changed, but its ability to do work has increased.

Virtual machine technology has many benefits. One key component of virtual machines is that they allow data centers to use their hardware more effectively. Organizations that use virtual machines are able to reduce the number of physical machines needed to run their operations. The consolidation is accomplished by transforming each physical machine into a virtual machine and then running multiple virtual machines on the same physical hardware, as illustrated in Figure 1.1. In this Figure we see two physical machines converted into virtual machines which both run on a single physical host.

Server consolidation is particularly useful for data centers. Oftentimes, many computers in data centers are underutilized. The computers are underutilized for two reasons. First, unrelated software is run on different machines in order to minimize the risks associated with program interactions. That way, if one of the programs is unstable or exposes a security problem, the damage would be minimized because no other software is running on that machine. The second reason for server consolida-

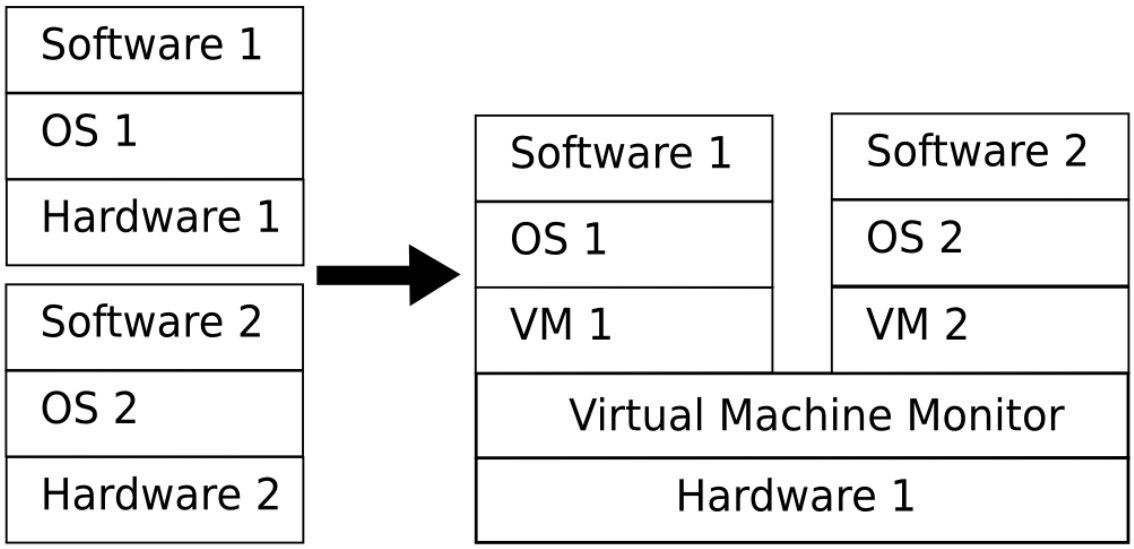

Figure 1.1: Two physical machines are converted into virtual machines which run on the same host.

tion is that with how fast computers are becoming, most programs will not use any significant portion of the resources available on a computer. Using virtual machines as part of server consolidation allows all underutilized machines to run on the same hardware while minimizing the above risks.

However, if consolidation is not done carefully, the hardware could become overloaded. This problem is difficult because the physical resource that will become the bottleneck may not be evident during the planning stages. Some physical components that could easily become overloaded include the CPU, network cards, and other I/O devices. When a load is limited by the physical resources of the computer it is running on, but sufficient physical resources are available and unused on another computer in the same system, we say that load is imbalanced. When the resources required by a running virtual machine change dramatically over time, load imbalances will likely occur.

Load imbalances are a concern because of decreased throughput and system response.[21]. In a large organization with numerous servers, the cumulative effect of load imbalances could be substantial. Because load imbalances reduce throughput

and increase response time of a system, more hardware will be needed to meet the same level of service when loads are imbalanced.

Our goal in this work is to explore the benefits of dynamically load balancing virtual machines. When we started our research, there was no published work on load balancing virtual machines dynamically. Other researchers published a paper on the same topic after we finished gathering our data. The benefits and challenges of dynamic load balancing have been studied by computer scientists for many years, and we seek to add to this work by applying published techniques.

1.1 Virtual Machine Introduction

Virtual machines are not new to the field of computer science. Virtual machines have been around for a long time and presently encompass a wide variety of technologies and abstractions. Despite the different uses for virtual machines, however, all virtual machines have at least one thing in common. Rosenblum explains the link between all virtual machines when he says, "In all definitions the virtual machine is a target for a programmer or compilation system. In other words, software is written to run on the virtual machine" [15].

On a theoretical level, a virtual machine is simply an isomorphism that maps a virtual set of hardware to physical hardware. For each transition in the virtual machine, there is a series of transitions in the host to get to an equivalent state. Often, the transitions inside of virtual machines are so primitive that there is a one-to-one mapping between the state of the virtual machine and the state of the physical host that is being emulated.

On a more practical level, a virtual machine is a program which runs on a virtual machine monitor. A virtual machine monitor exposes an interface that can be used to run programs and often mirrors the underlying hardware. The physical machine is called the host, and virtual machines are referred to as guests.

For our research work, we use the Xen Virtual Machine Monitor [4]. The Xen Virtual Machine Monitor runs directly on the hardware. There is a special virtual machine in Xen called Domain0 which has management functionality that other virtual machines do not have. Every running Xen installation will include Domain0, so every Xen installation has at least one virtual machine.

1.1.1 A Brief History of Virtual Machines

IBM introduced virtual machines in the 1960's to multiplex hardware between different users. Each user had a virtual machine which appeared to be running directly on the hardware. During that time, operating systems had limited capability for supporting multiple concurrent users. Thus, IBM's virtual machines were a huge step forward in the field of computer science. These virtual machines helped pave the way for the complex multi-user operating systems that are ubiquitous today.

Virtual machines were popular in both the academic and the business world in the 1960's and 1970's. The virtual machines monitors of that period ran directly on the hardware of expensive main frames. The hardware assisted the virtual machine monitor by trapping privileged instructions and passing control back to the virtual machine monitor.

In 1966, not long after IBM released its implementation of virtual machines, researchers created an intermediate language produced by a BCPL compiler. This intermediate language, called O-code, was executed by a virtual machine much like java byte-code executes on the JVM. This was the first virtual machine for a programming language. Pascal popularized the concept of a programming language virtual machine when it introduced the p-code machine in the early 1970's.

During the 1980's and early 1990's, however, few advancements were made related to virtual machines. Virtual machines did not receive the attention they had in previous decades because there were fewer economic reasons to use virtual machines.

Virtual machines had been used to partition the resources of main frames, but during the 1980's and 1990's, for the first time, a person could have their own dedicated computer. Also, the hardware that became popular for desktop machines did not support the hardware primitives that had been previously used for virtualization.

In the late 1990's interest in virtual machines was rekindled. Researchers and businesses became interested in virtual machines as a way to conserve server space and consolidate many applications onto one machine. One group of researchers at Stanford found a method to run commodity operating systems on hardware that did not support virtualization [5]. In 1998 these researchers left academia to start VMware. Then in 2003, researchers at Cambridge introduced the Xen virtual machine monitor [4]. Xen requires that the host operating system be specially compiled to run on Xen, but offers better performance than VMware's products. The goal of many of these researchers is to increase the effectiveness of virtual machine use. We seek to further that work by showing the circumstances under which dynamic load balancing is more beneficial than the static load balancing that is currently favored.

1.1.2 Taxonomy of Virtual Machines

There are many types of "virtual machines". In fact, if one were to look at the purpose and implementation of virtual machines, many common virtual machines have more differences than similarities. Many different programs that are used for many different purposes are often all categorized as virtual machines. One reason that dissimilar things are all known as virtual machines is because the meaning of the word *machine* changes based on context. In some circumstances it is correct to think of a machine as a combination of some of the hardware and the interface presented by the operating system. In other circumstances, however, a machine is a complete system containing CPU's, memory, disk drives, and I/O devices.

To help differentiate between various virtual machines, Smith and Nair created a taxonomy [18]. This taxonomy divides virtual machines into families based on the type of program the virtual machine runs. The four VM families from this taxonomy are listed below. An example from each of the four families is also discussed.

- Process VMs with an interface the same as the underlying machine.
- Process VMs with an interface different from the underlying machine.
- System VMs with an interface the same as the underlying machine.
- System VMs with an interface different from the underlying machine.

HP Dynamo is a process VM with an interface the same as the underlying machine. The Dynamo system dynamically optimizes executing machine code. Typically, even programs produced by an "optimizing" compiler will speed up around 20% when running under HP Dynamo. One reason for the performance gains is that the HP Dynamo knows exactly what type of machine the program is running on. This allows some instructions to be replaced with alternative instructions that run more quickly. Another reason for the performance gains is that HP Dynamo has actual run time information which can be used to optimize the parts that matter most [3].

The Java Virtual Machine (JVM) is a process VM with an interface different from the underlying machine. One attractive feature of the JVM is portability. Since the JVM has been ported to many different platforms, programs designed to run on the JVM are much more portable than programs designed to run on a specific hardware/operating system combination. The JVM has many other features that make it an attractive run-time environment. For example, JVM provides a safe environment for untrusted applications, it automatically reclaims unused resources, and it provides a rich execution environment.[12].

VMware ESX Server is a system virtual machine that uses the same interface as the underlying machine. VMware ESX Server allows multiple underutilized phys-

ical machines to be consolidated into one machine. Also, because most instructions run directly on the hardware, ESX Server is nearly as efficient as running on bare hardware. [20].

Finally, an example of a system VM with an interface different from the underlying machine is Transmeta's Crusoe. The Crusoe uses dynamic code translation to run IA-32 code on a proprietary VLIW instruction set. Compared to other chips that run IA-32 code, the Crusoe is energy efficient. This is because the Crusoe doesn't implement the full IA-32 instruction set in hardware.

1.1.3 The Xen Virtual Machine Monitor

Not all virtual machines fit perfectly into the categories suggested by Smith and Nair. Created in 2003 by a team of researchers at Cambridge, the Xen Virtual Machine Monitor is one such example. The Xen Virtual Machine Monitor is unique because it uses a technique known as *paravirtualization* [4]. Paravirtualization presents an interface that is similar, but not identical to, the underlying hardware. In particular, some privileged instructions that make virtualization difficult are dropped. In many cases those instructions are replaced by calls into the Xen Virtual Machine Monitor. This means that operating systems must be specifically ported to run on Xen, but application programs do not need to be modified. Because the Xen Virtual Machine Monitor is not hampered by instructions that are difficult to virtualize, Xen's performance is better than other system VMs.

One other feature of Xen that is extremely important to our work is that Xen supports the live migration of virtual machines between hosts. A running virtual machine can be moved to a new virtual machine with downtime as low as 60 milliseconds. This is accomplished by doing a pre-copy of RAM. Before the virtual machine is migrated, RAM is copied from the source to the destination. Usually, the virtual machine will only be using a relatively small amount of its total memory, so

even though some pages of memory will have to be sent again, many pages of memory only need to be sent once. This minimizes the total time needed for the virtual machine to be turned off [7].

Rosenblum, a researcher at VMware, talked about live migration and some of the things this technology will make possible.

> Today, manual migration is the norm, but the future should see a virtual machine infrastructure that automatically performs *load balancing*, detects impending hardware failures and migrates virtual machines accordingly, and creates and destroys virtual machines according to the demand for particular services. (Emphasis not in original)

It is our hope that this work will contribute in a small way to the future of virtual machines described by Rosenblum.

1.1.4 Recent Work Involving Virtual Machines

Some recent work with virtual machines has attempted to establish standards for how virtual machines should interact with other things. One example of this is the work by Russell [17]. Russell establishes an API for how virtualization systems interact with devices. His goal is to save future hypervisor implementors the trouble of implementing their own drivers.

Virtual machines are also being applied as solutions to a wide variety of problems. For example, Sotomayor et. al. recently used virtual machines to help schedule workloads on clusters.

These are just two examples of work currently being done with virtual machines. Hopefully virtual machines will be beneficial to researchers for many years to come.

1.2 Load Balancing Introduction

Load balancing, or scheduling, is the problem of finding a mapping between jobs to physical resources. In other words, load balancing is the problem of deciding which jobs should be allocated to which physical resources.

In this work we compare static and dynamic load balancing. In static load balancing, the mapping of jobs to resources is not allowed to change after the load balancing has begun. On the other hand, dynamic load balancing will change the mapping of jobs to resources at any point, but there may be a cost associated with the changes. There are two main types of dynamic load balancing algorithms: preemptive and non-preemptive. In a preemptive algorithm, the mapping of jobs to physical resources may change while a job is running. In contrast, with non-preemptive algorithms, once a job has started the physical resources assigned to that job cannot change.

Load balancing is essentially a resource management problem. Stated simply, the load balancing problem is knowing how to divide work between available machines in a way that achieves performance goals. Common goals of load balancing include maximizing throughput, minimizing response time, maximizing resource utilization, minimizing execution time, and/or minimizing communication time. System designers should know the system's specific load balancing goals before choosing an algorithm.

Load balancing is also a scheduling problem. The operating system on each processor performs local scheduling. Local scheduling involves deciding which processes should run at a given moment. Global scheduling, on the other hand, is the process of deciding where a given process should run. Global scheduling may be performed by a central authority or it can be distributed among all of the processors. The two main types of scheduling are static scheduling and dynamic scheduling.

1.2.1 Static and Dynamic Scheduling

Static scheduling assigns tasks to processors before execution begins. Tasks will not change processors while the system is executing any given task. One challenge of static scheduling is predicting the execution behavior of a system. Another challenge of static scheduling is partitioning tasks to processors in a way that maximizes resource utilization and minimizes communication time.

Perhaps the greatest obstacle related to static scheduling, however, is that researchers have shown that finding the optimal solution is NP-complete. Because finding the optimal solution is NP-complete, much of the research done for static scheduling has involved developing heuristics. One common heuristic in static scheduling is to give priority to long tasks.

In addition to the limitation of being NP-complete, it is also challenging to estimate task execution times before a task runs. This is difficult because the run-time behavior of a task is often affected by things that cannot be predicted beforehand, such as conditional constructs or network congestion.

Dynamic scheduling redistributes tasks during execution. The redistribution typically moves tasks from heavily loaded processors to lightly loaded processors. A dynamic scheduling algorithm typically has three parts.

- Information Policy: This explains how much information is available to the decision-making components.
- Transfer Policy: This determines when tasks will be redistributed.
- Placement Policy: This decides where a task will be placed.

A dynamic scheduling algorithm can be centralized or distributed. In centralized dynamic scheduling, all information is sent to a single decision-making agent. The decision-making agent decides when load balancing will occur and which tasks will be moved. In distributed dynamic scheduling, load information is not shared

globally. All processors take part in deciding when load-balancing occurs and which tasks will be migrated.

When compared with static scheduling, one major advantage of dynamic scheduling is that the run-time behavior of the system does not need to be known in advance. Dynamic scheduling algorithms are very good at maximizing utilization of all resources. Unfortunately, dynamic scheduling has run time overhead associated with gathering information and transferring tasks to different processors. If the run time overhead is greater than the extra work that is accomplished by moving tasks from heavily loaded processors to lightly loaded processors, than dynamic load balancing may slow execution.

1.2.2 Load Rebalancing

Load rebalancing is a more specialized version of the load balancing problem. In load rebalancing, given a suboptimal assignment of jobs to processors, we are interested in reducing the load on the heaviest loaded processor. This is accomplished by migrating jobs from the heaviest loaded processor to other processors. Aggarwal et. al. provide a good formal definition of the load rebalancing problem [2].

The Load Rebalancing Problem Given an assignment of the n jobs to m processors, and a positive integer k, relocate no more than k jobs so as to minimize the maximum load on a processor. More generally, we are given a cost function c_i which is the cost of relocating job i, with the constraint that the total relocation cost be bounded by a specified budget B.

Aggarwal et. al. also prove many interesting features of load rebalancing. In particular, they classify different load rebalancing problems, such as load rebalancing with constraints on which machines a job may be assigned to, or constraints on which jobs may be on the same machine. The theorems are too numerous to list here, but in general they show that load rebalancing is an NP-hard problem. Further, many

of the subproblems they studied cannot have an approximate solution in polynomial time unless $P = NP$ [2].

Some researchers dispute the effectiveness of dynamic load balancing. In particular Eager et. al. argue that dynamic load balancing by migrating jobs will never yield major performance benefits compared to balancing load through initial placement of jobs [8]. They argue that only under extreme condition can migration offer modest performance benefits. However, this point is disputed, and many researchers continue to argue that dynamic load balancing can be effective, even in circumstances that aren't extreme. Also, dynamic load balancing can also have many benefits besides performance. For example, when load across a system is low dynamic load balancing could allow for many virtual machines to be combined and for hardware to be turned off. This could lead to the system using significantly less power which would probably save money.

1.3 Thesis Statement

Applying published dynamic load balancing algorithms to virtual machines can yield improvements in both throughput and response time. Please note that the dynamic load balancing algorithms we examine in this thesis have never been applied to virtual machines in any published work. In fact, when we started this thesis, there were no published papers about dynamically load balancing virtual machines. We are primarily interested in answering these three questions:

- "What is the overhead of load balancing using our policy engine?"
- "Can dynamic load balancing be practically applied to virtual machines?"
- "How does the length of time that the workload runs effect the performance increase due to dynamic load balancing?"

Chapter 2

Policy Engine

We create a policy engine to implement our dynamic load balancing algorithm. Each host will send state information to the policy engine. The policy engine will then make all migration decisions based on the information it receives and the policy that is currently running. By changing the policy used by the policy engine, we can implement different dynamic load balancing algorithms or even migrate for reasons other than load balancing. For example, the policy engine could be used to migrate virtual machines when hardware fails.

In Figure 2.1 we show the relationship between the host machines running Xen and the policy engine. In this Figure, there are two host machines running the Xen Virtual Machine Monitor. Each of the hosts is sending state data to the policy engine. The policy engine will tell the Xen hosts when a virtual machine should migrate to a new host depending on the state of the hosts and the policy that is being executed. In Figure 2.1, one virtual machine is currently being moved to a new host.

In the rest of this chapter, we will discuss our policy engine in greater detail.

2.1 The Specifics of Our Policy Engine

A policy engine is a software program which executes commands based on a set of rules known as a policy. In our work, the policy engine makes decisions about when to migrate virtual machines between hosts. Using a policy engine, a user can automatically migrate virtual machines to new hosts in a variety of circumstances,

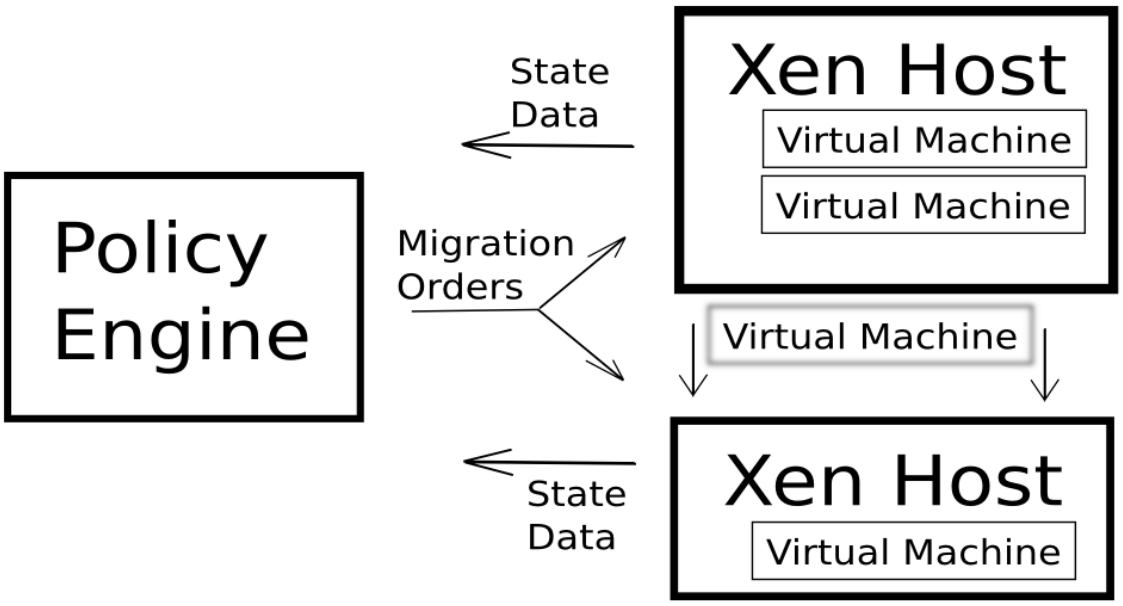

Figure 2.1: Block diagram showing relationship between policy engine and the Xen hosts.

including when hardware fails, to balance load, to place virtual machines on certain hosts in ways that might optimize performance, or to move virtual machines that should not be turned off from a host that is being shut down.

Our policy engine, which runs inside of a virtual machine using a linux kernel, queries each host for information that is relevant to the policy that is running. Relevant information could include, but is not limited to, CPU usage, memory usage, and the state of the operating system. This information is relayed through a simple web service running on each Xen host to the policy engine. Based on the information the hosts report, the policy engine will act within the parameters of its policy to decide when a virtual machine should migrate. The policy engine will also determine which host will accept the migrating virtual machines.

We configured our policy engine to dynamically load balance virtual machines based on CPU thresholds. We currently require that policies be written in Ruby, though a domain specific language may be better suited for future work. For more

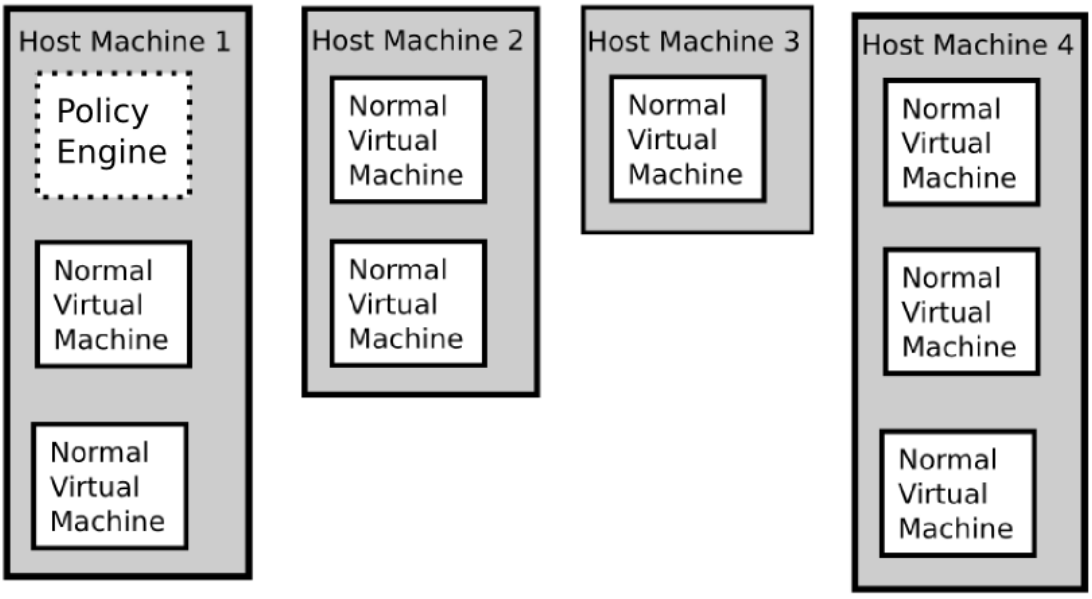

Figure 2.2: For our prototype, the policy engine runs inside of a virtual machine separate from everything else.

details about the dynamic load balancing algorithm we used for our policy, please see Section 3.

In Figure 2.2 we see an example system. Here we have four hosts, with the policy engine running on host machine 1. Please note that the policy engine is running as a normal virtual machine. From an external perspective it looks like any other virtual machine. In fact, the policy engine could even move itself to a different host like any other virtual machine.

In Figure 2.3 we see the communication links between the policy engine and the host machines. Notice that the policy engine communicates with all of the hosts. The policy engine will periodically query the hosts for information. Based on the reported information, the policy engine will instruct hosts to migrate virtual machines.

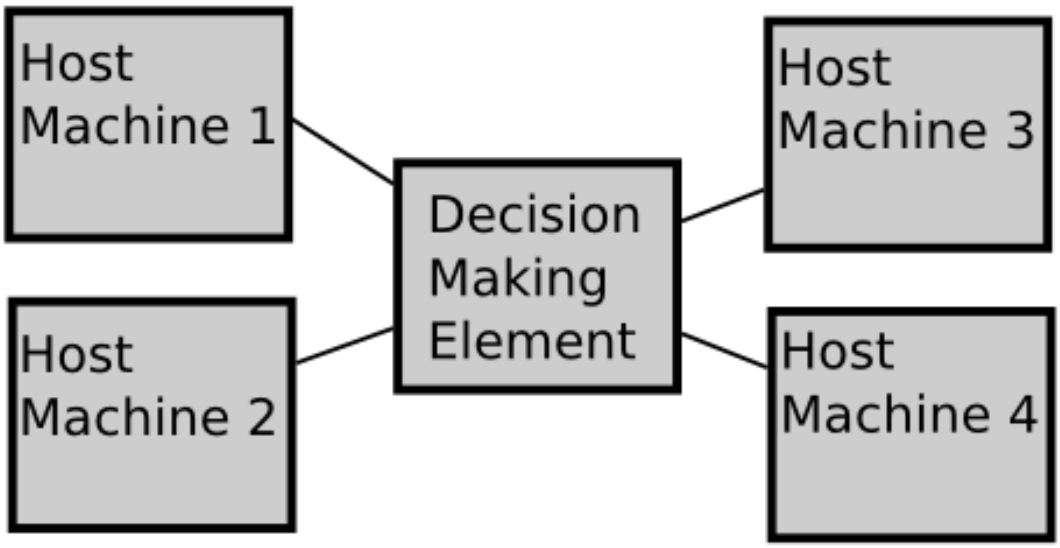

Figure 2.3: The prototype policy engine communicates with all hosts to decide when virtual machines should be migrated and to initiate migration when necessary.

Chapter 3

Load Balancing Algorithm

The algorithm we implemented inside of our policy engine is a modified version of the Central Scheduler Load Balancing (CSLB) algorithm developed by Lan and Yu [11]. We selected the CSLB algorithm because it met our requirement of being published in an academic venue while also being easy to understand and implement. Also, because the CSLB uses a central node that makes all load balancing decisions, it fits our current implementation of our policy engine better than a distributed scheduling load balancing algorithm. The differences between our algorithm and the algorithm presented by Lan and Yu will be discussed after we introduce their algorithm.

3.1 Defining the CSLB Algorithm

We will describe the CSLB algorithm using the five phases of dynamic load balancing algorithms introduced by Watts and Taylor [21]. One critical difference between the work of Watts and Taylor and our work is that Watts and Taylor were working with processes; they were not working with virtual machines. These phases are load evaluation, profitability determination, work transfer vector calculation, task selection, and task migration.

1. **Load Evaluation:** As its name implies, the CSLB is a central scheduling algorithm; all load information is sent to a central node. In the CSLB algorithm, the information to evaluate the work load can either be sent at a set interval or

when load information changes substantially. The CSLB algorithm also includes two global thresholds: one light threshold and one heavy threshold. These thresholds change based on the average CPU load of the nodes. The central scheduler computes the thresholds and broadcasts them to the other nodes. Each node determines load by using a heuristic to estimate how much time is needed to complete all of the jobs in its buffer. Each node puts itself into one of three groups based on the two thresholds and its load. Each node will then inform the central scheduler which group it is a part of. Figure 3.1 illustrates how nodes divide themselves into groups. Darker colors represent heavier loads. Based on the two thresholds created by the central scheduler, three groups are created. One group is for lightly loaded nodes; one group is for moderately loaded nodes, and the final group is for heavily loaded nodes.

2. **Profitability Determination:** Migration will be considered profitable if there is at least one host in the heavily loaded group and at least one host in the lightly loaded group.
3. **Work Transfer Vector Calculation:** Lan and Yu do not describe how to calculate the ideal amount of work to transfer in their work about the CSLB algorithm.
4. **Task Selection:** The CSLB paper also does not describe how to decide which tasks to migrate.
5. **Task Migration:** The CSLB paper does not describe how the jobs are actually transfered.

3.2 Our Modified Algorithm

Because the CSLB algorithm is not perfectly suited to our test environment, we have modified it to better meet our purposes. We describe our modified CSLB algorithm

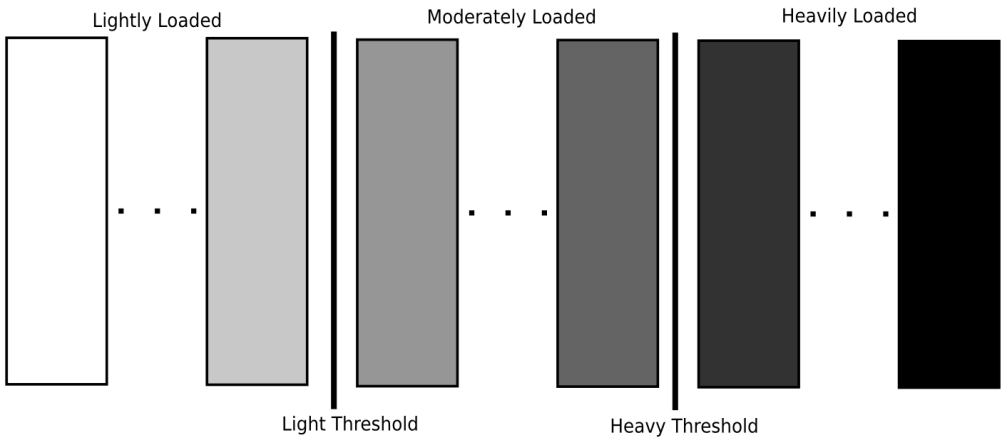

Figure 3.1: Each host machine is assigned to one of three groups based on CPU utilization and two thresholds. A darker color indicates that the host is more heavily loaded.

below using the five phases of a dynamic load balancing algorithm previously described by Watts and Taylor.

1. **Load Evaluation:** Unlike the jobs used in the original CSLB, virtual machines do not terminate. Rather, virtual machines often run indefinitely. Thus, load is determined by summing the percentage of the CPU used by each virtual machine. By changing the method that we use to compute load, we could balance load across other resources. For example, if we measured load by the amount of reads and writes to disk, we could put virtual machines that frequently use the hard disk on the same physical host as virtual machines that use the hard disk relatively infrequently. In any case, all of the workloads that we use to evaluate the dynamic load balancing of virtual machines are CPU bound. Thus, using the CPU to measure load is appropriate. Each host then puts itself into one of three groups based on two thresholds. Each host informs the central scheduler which group it is in.

2. **Profitability Determination:** Like the original CSLB algorithm, migration is considered profitable if there is at least one host in the highly utilized group and at least one host in the lightly utilized group.

3. **Work Transfer Vector Calculation:** Hosts from the lightly utilized group and heavily utilized group are paired. The ideal amount of work to transfer is equal to the difference between the middle value for the moderately utilized group and the current CPU utilization of the lightly utilized host.

4. **Task Selection:** The virtual machine from the highly utilized host with the CPU load closest to, but not exceeding, the amount specified in the work transfer vector calculation is selected for migration. If the CPU load of all virtual machines on the source host are greater than the amount specified in the work transfer vector, then the virtual machine with the smallest CPU utilization is selected.

5. **Task Migration:** The selected virtual machine is transfered to the lightly utilized host. This migration takes place using migration algorithm described Clark et. al [7].

To limit the number of variables in our test environment, our policy engine only balances load based on CPU usage. We recognize that other factors such as memory usage or I/O affect load and can contribute to load imbalances.

Chapter 4

Experiment Design

4.1 Hardware

We run all of our tests using two quadcore Dell 6650's. Both of these machines are connected to a storage area network, which fulfills the hardware requirements needed to migrate virtual machines between hosts. We also use a third machine to run the tests on the Dell 6650's and to record the results. When looking at the results, note that hyper-threading is enabled on both of the Dell 6650's. Thus, for each given task on a machine, the totals for CPU utilization sometimes sum to slightly more than 100%.

4.2 Test Harness

Our test harness runs all of our tests and also records the results of the tests. The test harness takes a script which has an entry for each test to be run. The entry describes which benchmark should run, how long the test should run, how many virtual machines should run the benchmark, and how often new virtual machines should be selected for running the benchmark. By varying how many virtual machines should run the benchmark, as well as how often new virtual machines should be selected, we can programmatically change whether the load is static or dynamic. This also affects how quickly load becomes unbalanced. The test harness randomizes the order of the entries before running the tests.

The test harness also collects data. For instance,the test harness records the percentage of the CPU being used by each host and each virtual machine. The test harness also records when migrations start and end, and which virtual machines and hosts were involved. Finally, the test harness records when each benchmark run started and finished. Each entry includes all test details in case we wish to repeat the test at a later point.

4.3 Static Versus Dynamic Load Balancing

Virtual machines are typically load balanced statically. In order to show improvements in run time and throughput, we will compare dynamic load balancing using our policy engine to static load balancing. In these tests we will use static load balancing as a baseline for performance. We will also place the virtual machines in a test environment where they are known to run favorably so that it is a fair comparison of static and dynamic load balancing.

4.4 Three Key Components

To determine the effectiveness of static load balancing versus dynamic load balancing, we are primarily concerned with three components: throughput, response time and overhead. These three variables will help us define the circumstances when dynamic load balancing is most effective.

We measure throughput by running benchmarks from SPEC's CPU2000 benchmark suite [9]. CPU2000 is used to measure the throughput of a processor. In particular, we run the benchmark in enough virtual machines to saturate the CPU of the host machine. For each benchmark from the CPU2000 suite that we investigate, we count how many times that benchmark can run during a given time period. We also vary the length of the benchmark run by having a virtual machine run the benchmark some set number of times before reporting completion to the test har-

ness. This way, we can have longer workloads that are identical to shorter workloads, except for the length.

We measure response time by running benchmarks from SPEC's WEB2005 benchmark suite [1]. WEB2005 is designed to measure the response time of a web server. We run the WEB2005 benchmark suite for multiple virtual machines at the same time. Again, all of our tests are CPU intensive. We measure the performance of the CPU by comparing throughput and response time. Both the CPU2000 and the WEB2005 are standard benchmark suites that are widely available, making it easier for other groups to repeat our experiments.

To measure the overhead of our policy engine doing dynamic load balancing, we run workloads that are static. Since the workloads are static, they derive no benefit from the dynamic load balancing algorithm being run by the policy engine. To determine the overhead of our policy engine, we will run workloads that will not benefit from dynamic load balancing with and without the policy engine and compare performance.

4.5 Three Important Questions

The purpose of our benchmarks related to throughput, response time and overhead is to help us answer some key questions related to load balancing. We are primarily interested in answering these three questions:

- "What is the overhead of load balancing using our policy engine?"
- "Can dynamic load balancing be practically applied to virtual machines?"
- "How does the length of time that the workload runs effect the performance increase due to dynamic load balancing?"

To answer the first question, we ran static CPU intensive workloads on both machines. We tried to find workloads that would completely saturate the CPU,

because if idle cycles were included then we would not be able to measure the overhead of our policy engine. By comparing the benchmarks with the policy engine turned on to benchmarks with the policy engine turned off, we get a worst case estimate of the policy engine's load in our environment. For the purposes of our research, we ran benchmarks simultaneously on nine virtual machines across two host machines.

We chose to run our static load benchmarks with nine virtual machines for the sake of consistency. We needed to have nine virtual machines for the dynamic tests, and so we decided to use nine virtual machines for the other benchmarks as well. Because we were using CPU bound benchmarks for the static workloads, as long as we used at least eight virtual machines, or enough to saturate the CPU, the results will be similar.

To answer the second and third questions, we ran a series of benchmarks with varying lengths. The benchmarks needed to be dynamic and random. The amount of load was also an important aspect of our benchmarks. There needed to be enough load so that a load imbalance would force migration to occur. Thus, one host would have more work than it could handle based on its CPU. The other host would have significantly less work, with enough room left to accommodate some of the work from the overloaded host. When load was balanced, both hosts had sufficient CPU cycles to perform their workloads.

Because of the above conditions, there is no static assignment of virtual machines to hosts that would not result in occasional load imbalances. In particular, we ran nine virtual machines across two hosts with eight virtual machines running the benchmark at any given time. Because we had nine virtual machines total and eight would run the benchmark, sometimes the load would be imbalanced. We varied the length of the benchmark to evaluate how the throughput and response time reported by the benchmarks would change as a function of the assigned workload. We

took particular note of length of time needed by workloads of the migrating virtual machine.

We expected that dynamic load balancing using our policy engine could reduce throughput and increase response time for very short workloads. This seems possible because of the large overhead created when transferring a virtual machine image between hosts. In the most extreme case, if the benchmarks are very short, then a virtual machine migration may cause a new load imbalance. This could create a cycle where one virtual machine migration causes a load imbalance which creates the need for another migration, and so forth.

4.6 Confounding Variables

To reduce the effect of confounding variables, tests were run in batches. The individual tests within a batch were run in a randomly assigned order. Some confounding variables that might effect the results are discussed below:

- **Network traffic**. The virtual machine's state is transfered over the network, but our test machines share a gigabit ethernet port with several other machines. Some of these machines are not under our control, so it is possible that another lab may do network-intensive tests and the switch could become a bottleneck. To counter this, we have randomized the order of our tests and tried to get multiple data points for every combination of tests.
- **Characteristics of the benchmark**. Each benchmark has different memory access patterns which may effect the results. The benchmark for each run is an independent variable and is recorded with the results.

Chapter 5

Results

5.1 Determining Policy Engine Overhead

First we discuss the benchmarks that we used to determine the overhead of the policy engine. For this set of benchmarks, we ran static workloads that would not benefit from load balancing. We ran two sets of tests with static workloads: one with the policy engine and one without the policy engine. The difference between these benchmark results will tell us how much overhead we had while running our policy engine.

We start with the gap benchmark from SPEC's CPU2000 benchmark suite because this benchmark that we ran the most times. The averages for these tests are shown in Table 5.1. The numbers show how much of the CPU was being utilized for different tasks while the benchmarks were being run. The numbers do not sum to 100% because hyperthreading was turned on while the benchmarks were running. To determine the odds that the overhead of both groups is identical, we analyzed the raw data using an ANOVA test. The ANOVA test, which stands for the Analysis of Variance test, is used to statistically compare multiple groups. The test will also determine if variation between two groups is due to random chance. In terms of results, a high ANOVA percentage indicates that there is no difference between the two groups in the aspects being measured. A low percentage indicates that there are differences between groups. The odds that the differences between these two groups is due completely to random chance is under 0.0001%.

Table 5.1: Benchmarks Measuring Overhead #1: Gap Benchmark

	Gap with Policy Engine	Gap without Policy Engine
Virtual Machines	94.514%	96.274%
Domain0	7.107%	6.352%
Idle	1.226%	0.768%
Policy Engine	0.085%	0.000%
Completed Benchmarks	243.406	247.603

Table 5.2: Benchmarks Measuring Overhead #2: GCC Benchmark

	GCC with Policy Engine	GCC without Policy Engine
Virtual Machines	94.370%	94.577%
Domain0	6.543%	6.250%
Idle	0.560%	0.564%
Policy Engine	0.069%	0.000%
Completed Benchmarks	437.250	438.600

We also used an ANOVA test for the GCC benchmark summarized in Table 5.2. As with the other benchmark, this test was used to determine the odds that we would achieve these results assuming no differences between the groups. The result of the ANOVA test was 16.7%. We had fewer total benchmark runs with this test, which explains the lack of certainty in the statistical results.

Table 5.3: Benchmarks Measuring Overhead #3: Mesa Benchmark

	Mesa with Policy Engine	Mesa without Policy Engine
Virtual Machines	95.410%	96.937%
Domain0	6.802%	6.324%
Idle	1.137%	0.523%
Policy Engine	0.058%	0.000%
Completed Benchmarks	206.000	209.000

Table 5.3 summarizes the test results for the mesa benchmark. Assuming that both groups are equal,the ANOVA test results state that the odds of getting results like these is 7.1%.

The averages of the results from the VPR benchmark runs are shown in Table 5.4. Assuming no difference between the groups, the odds that we would get results

Table 5.4: Benchmarks Measuring Overhead #4: VPR benchmark

	Vpr with Policy Engine	Vpr without Policy Engine
Virtual Machines	95.343%	96.097%
Domain0	6.652%	6.321%
Idle	1.287%	1.560%
Policy Engine	0.064%	0.000%
Completed Benchmarks	217.250	218.625

similar to these is 2.0%, according to the the ANOVA test. When a multiple analysis of variance is performed on all of the data, the odds that we would get data similar to this is less than 0.0007%, assuming the groups were the same. Thus, it is clear that the policy engine has statistically significant overhead associated with it. However, in practical terms the overhead looks to be rather low. When the policy engine was running, on average we completed 1% fewer benchmarks.

5.2 Dynamic Load Balancing Benchmarks

This series of benchmarks is designed to measure the performance impact of dynamic load balancing on workloads of various lengths. We ran each benchmark many times with the policy engine turned both off and on, varying the number of times a virtual machine would have to complete the benchmark before another virtual machine was randomly selected to run the benchmark.

We start by explaining the information that is presented in the tables. As mentioned previously, we want to see how the length of the workload effects the results. To lengthen the workload, we would rerun a benchmark multiple times on the same virtual machine. By rerunning a benchmark multiple times on the same virtual machine, we are lengthening the time between points when new virtual machines are selected for work. In our tables, “1 Iteration” means that we only ran each benchmark once before selecting a new virtual machine. “2 Iterations” means that we ran each benchmark two times before selecting a new virtual machine. Also, we recorded some

information about how the CPU was being used while the benchmarks were running. Here we show what percentage of the time the CPU was given to a virtual machine, what percentage of the CPU domain0 used, what percentage of the CPU was idle, and what percentage of the CPU was used by the policy engine. For each benchmark length that we tested, we include all of the above information averaged for all of our tests, and also the average number of completed benchmark runs.

Table 5.5: Benchmarks Measuring Dynamic Workloads #1: Gap Benchmark

	Runs	VM%	Dom0%	Idle%	PE%
1 Iteration	222.7	82.03%	12.58%	8.32%	0.18%
2 Iterations	229.7	86.12%	11.07%	5.76%	0.07%
4 Iterations	236.9	90.38%	9.98%	3.36%	0.09%
8 Iterations	239.9	92.43%	7.97%	2.13%	0.08%
16 Iterations	241.8	93.73%	7.22%	1.52%	0.08%
32 Iterations	242.7	94.03%	7.24%	1.42%	0.09%
64 Iterations	242.4	93.95%	7.16%	1.27%	0.08%
128 Iterations	242.3	93.97%	7.11%	1.53%	0.08%
256 Iterations	242.2	93.83%	7.23%	1.50%	0.08%
512 Iterations	243.5	94.40%	7.04%	1.29%	0.09%
No Policy Engine	234.96	90.49%	8.55%	3.80%	0.00%

Because it is the benchmark we ran most frequently, we start with the gap benchmark. Table 5.5 summarizes the results. Figure 5.2 shows each of the number of benchmark runs that were completed over the log of the workload length for each of the tests that was run without the policy engine. One interesting aspect of this figure is that as the length of the benchmark increases, two groups are created: one group which completes a large number of benchmark runs and another group that completes a noticeably smaller number of benchmark runs. These two groups are caused by load imbalances. For long benchmarks with static load balancing, if the load is imbalanced at the start, it stays imbalanced and throughput suffers. However, if load is balanced at the start, the workload will stay balanced and throughput reaches near optimal levels. In contrast, Figure 5.1 shows that when the policy engine is doing a dynamic load balancing algorithm, the number of completed benchmark runs

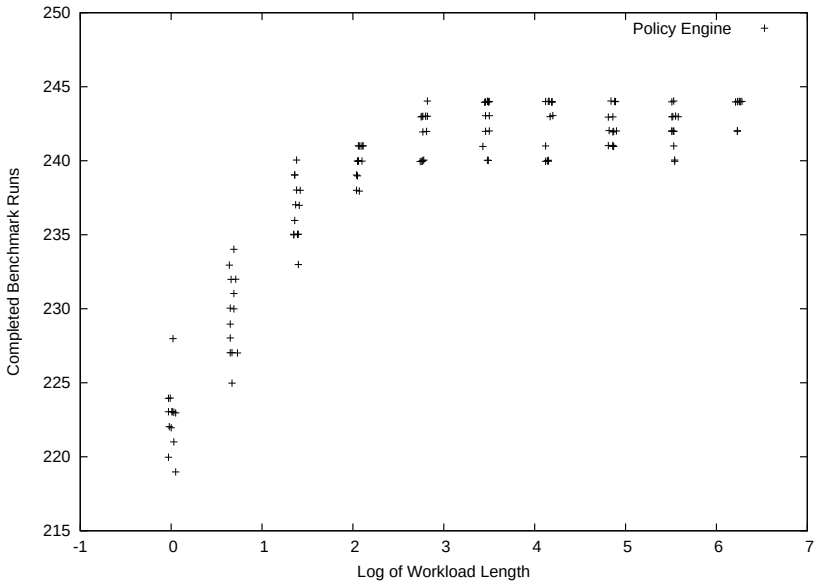

Figure 5.1: This figure shows the results of benchmark runs for the *gap* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. For this set of benchmarks, the policy engine is turned on.

is fairly consistent no matter how long the workload runs. However, the throughput of the system increases as the benchmarks get longer, with diminishing returns after a point. We also see that for short benchmark runs, the policy engine's overhead while doing dynamic load balancing is greater than the benefit derived from load balancing. On the other hand, throughput for dynamic load balancing is higher on average with longer benchmarks.

Table 5.6 summarizes the data from our test runs using the mesa benchmark. Also, each of the individual tests can be seen in Figure 5.5 and Figure 5.6. As can be seen in the figures and tables, this data has many of the same trends that we saw with the gap benchmark. These results are particularly interesting for longer iterations when the policy engine is turned off. Under these circumstances, the number of completed benchmark runs would either be really high or really low. Also, when the

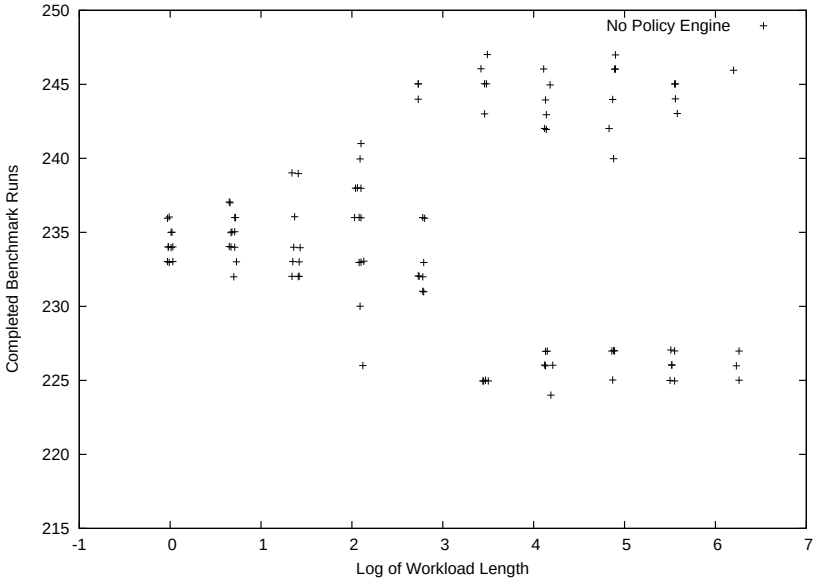

Figure 5.2: This figure shows the results of benchmark runs for the *gap* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. For this set of benchmarks, the policy engine is turned off.

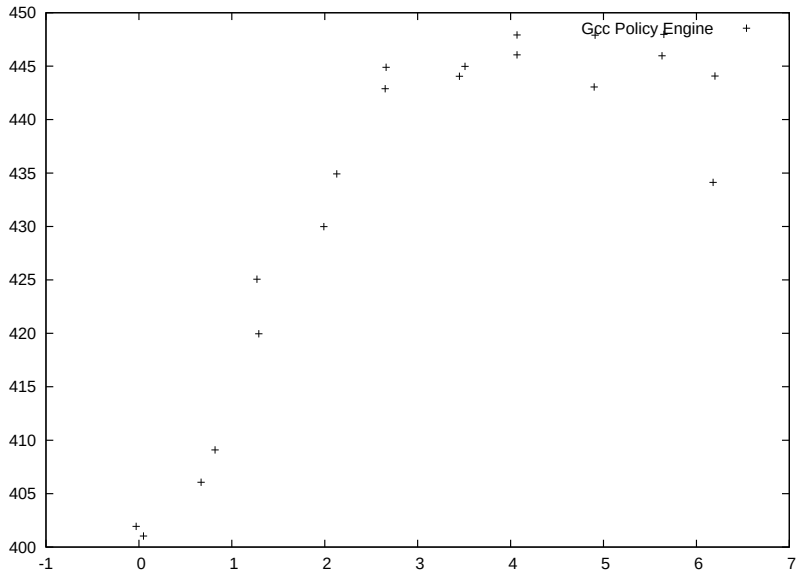

Figure 5.3: This figure shows the results of benchmark runs for the *gcc* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned on.

policy engine was turned on, completed benchmark iterations gradually increased. In this case, though, we stopped seeing benefits at 32 iterations.

A table summarizing the test runs for the benchmarks running VPR is shown in Table 5.7. This data shows similar patterns to what we saw in the past benchmarks.

The results from the GCC benchmark are summarized in Table 5.8. We also show individual benchmark results in Figures 5.3 and 5.4.

5.3 Problems Related to WEB2005 Results

Finally, we also ran the SPEC's WEB2005 benchmark many times. Unfortunately, we had difficulty getting useful information from these tests. These tests were problematic because of the possibility of CPU overload. If the CPU was overloaded, then the benchmark run would simply fail. However, when the CPU was not overloaded, there was no statistical difference between when the policy engine was turned on and

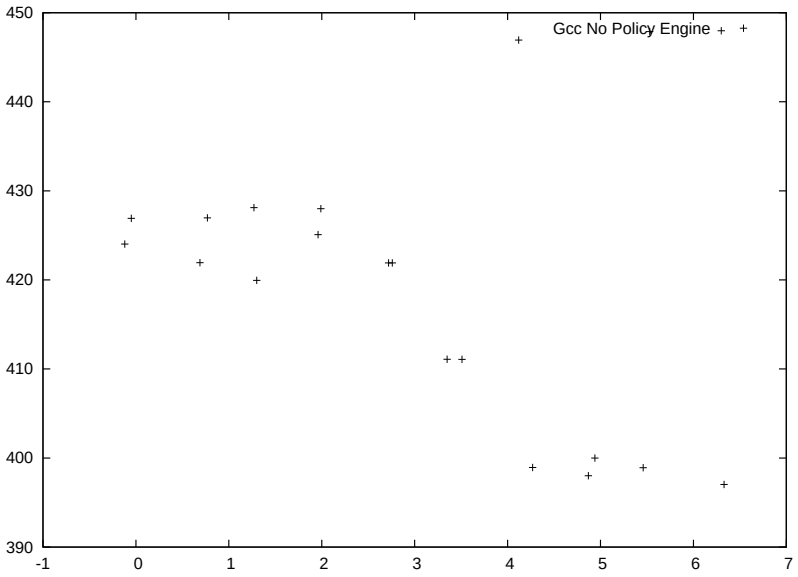

Figure 5.4: This figure shows the results of benchmark runs for the *gcc* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned off.

when it was turned off. The test failed 45% of the time when the policy engine was turned off, and 38% of the time when the policy engine was turned on. However, the difference is not statistically significant. Also, for the tests that completed successfully, we were not able to find any statically significant difference between the response times.

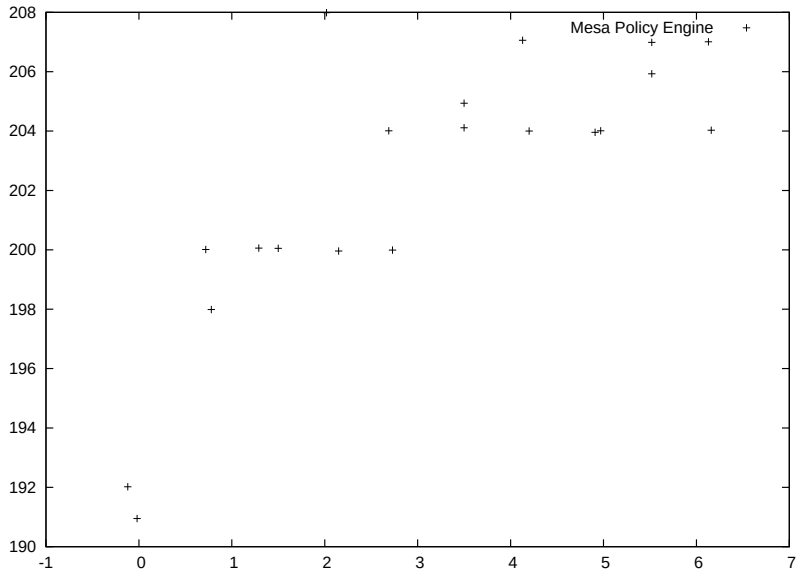

Figure 5.5: This figure shows the results of benchmark runs for the *mesa* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned on.

Table 5.6: Benchmarks Measuring Dynamic Workloads #2: Mesa benchmark

	Runs	VM%	Dom0%	Idle%	PE%
1 Iteration	191.5	85.45%	11.95%	6.94%	0.05%
4 Iterations	199.0	91.05%	8.11%	2.92%	0.05%
8 Iterations	200.0	94.05%	7.33%	1.40%	0.06%
16 Iterations	200.0	92.97%	7.53%	2.49%	0.06%
32 Iterations	204.0	94.67%	6.47%	1.48%	0.06%
64 Iterations	204.0	95.23%	6.64%	1.19%	0.06%
128 Iterations	200.0	94.54%	6.54%	1.12%	0.06%
256 Iterations	202.0	95.54%	6.87%	1.28%	0.06%
512 Iterations	204.0	95.01%	6.81%	1.31	0.06%
No Policy Engine	199.9	92.65%	7.55%	3.29%	0.00%

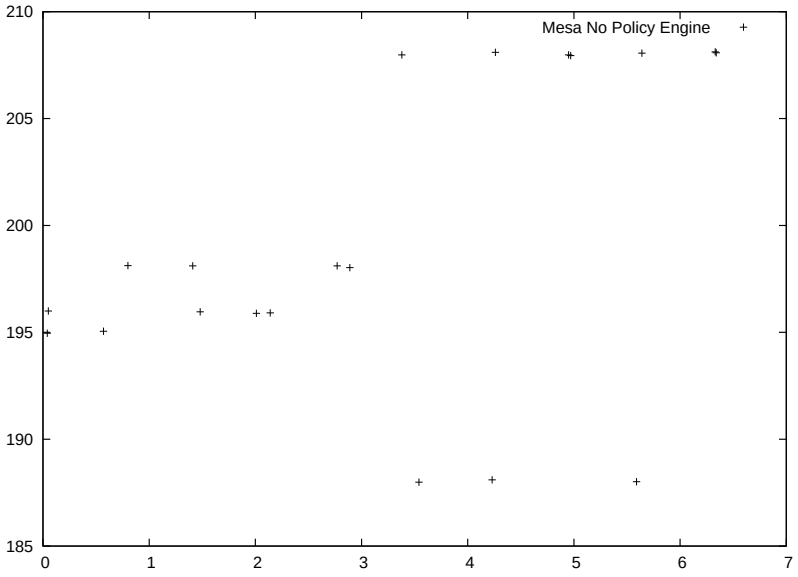

Figure 5.6: This figure shows the results of benchmark runs for the *mesa* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned off.

Table 5.7: Benchmarks Measuring Dynamic Workloads #3: Vpr benchmark

	Runs	VM%	Dom0	Idle%	PE%
1 Iteration	205.5	85.87%	16.81%	6.50%	0.07%
2 Iterations	208.5	88.54%	9.42%	4.81%	0.07%
4 Iterations	212.4	90.41%	8.96%	3.75%	0.10%
8 Iterations	216.3	93.36%	7.40%	1.92%	0.08%
16 Iterations	216.8	94.31%	6.77%	1.00%	0.08%
32 Iterations	216.8	94.76%	6.99%	1.24%	0.08%
64 Iterations	219.0	94.50%	7.01%	1.83%	0.08%
128 Iterations	218.5	94.48%	7.07%	1.67%	0.08%
256 Iterations	218.3	94.77%	6.73%	1.04%	0.08%
512 Iterations	217.5	93.97%	7.30%	1.75%	0.08%
No Policy Engine	210.2	91.21%	7.92%	3.90%	0.00%

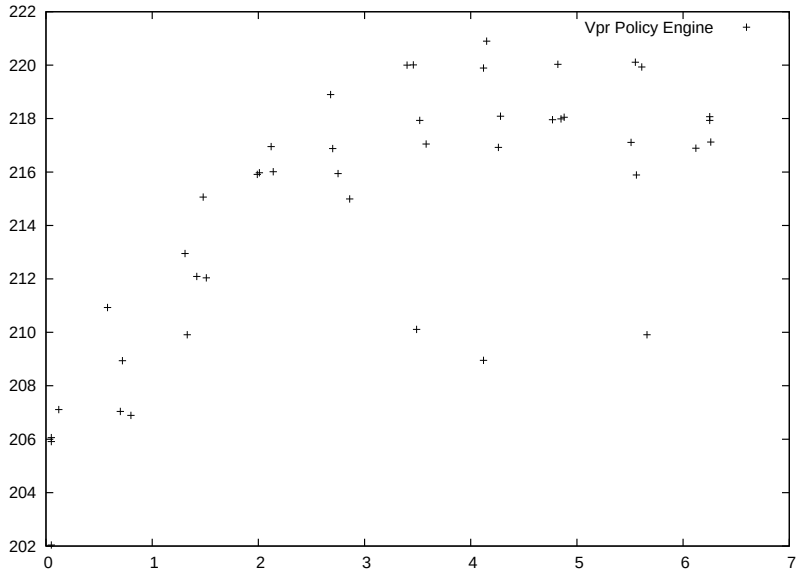

Figure 5.7: This figure shows the results of benchmark runs for the *vpr* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned on.

Table 5.8: Benchmarks Measuring Dynamic Workloads #4: Gcc benchmark

	Runs	VM%	Dom0	Idle%	PE%
1 Iteration	401.5	85.87%	12.45%	7.83%	0.06%
2 Iterations	407.5	88.54%	11.25%	6.64%	0.06%
4 Iterations	422.5	90.41%	9.88%	4.43%	0.10%
8 Iterations	432.5	93.36%	8.58%	2.74%	0.06%
16 Iterations	444.0	94.31%	7.06%	1.02%	0.07%
32 Iterations	444.5	94.76%	6.83%	1.03%	0.07%
64 Iterations	447.0	94.50%	6.64%	0.68%	0.07%
128 Iterations	445.5	94.48%	6.86%	0.79%	0.07%
256 Iterations	447.0	94.77%	6.60%	0.62%	0.07%
512 Iterations	439.0	93.97%	6.94%	0.99%	0.07%
No Policy Engine	420.2	91.21%	7.96%	4.35%	0.00%

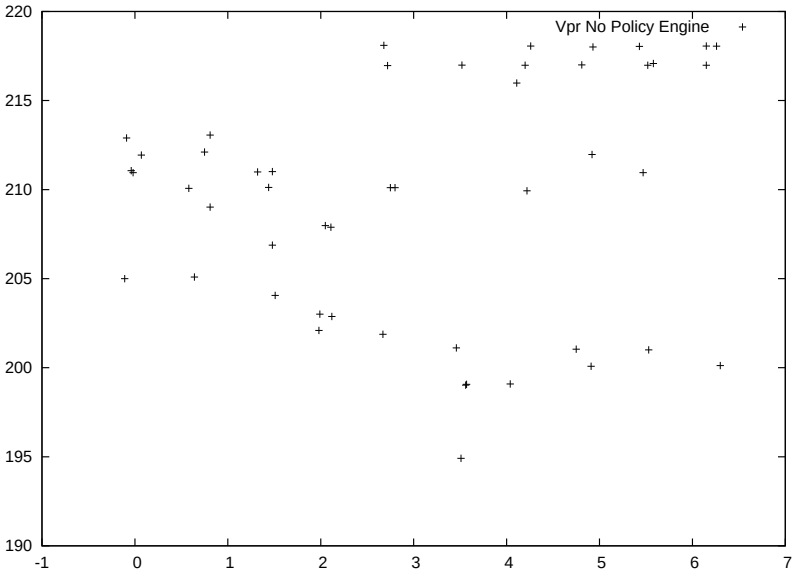

Figure 5.8: This figure shows the results of benchmark runs for the *vpr* benchmark over the log of the number of benchmarks that would run consecutively before randomly selecting a new virtual machine. The policy engine was turned off.

Chapter 6

Analysis

We begin our analysis by building a model of the anticipated benchmark results. This analysis is an original work and not based off of any previous work. This model was created to provide us with a basis for comparison as we examine the benchmarks. When looking at the actual results, we will make note of any interesting results and compare those results to the results predicted by the model.

6.1 Static Throughput Workloads

By definition static workloads are workloads that do not change over time. In other words, the load on each element is constant. We first analyze static throughput workloads for cases when the policy engine is off, and then for cases when the policy engine is turned on.

Let R_{cpu} be the amount of CPU resources available for T units of time. Let O_{vmm} be the overhead from the virtual machine monitor for T units of time. Let O_i be the overhead for running the ith virtual machine, excluding the benchmark, for T units of time. O_i includes things like the overhead of the operating system in the virtual machine. Finally, let each run of the benchmark consume O_b of CPU resources.

Let br be the expected number of finished benchmark runs. The expected number of finished benchmark results is

$$ebr_{off} = (R_{cpu} - O_{vmm} - \sum_{i=1}^{n} O_i)/O_b \tag{6.1}$$

If we assume that all virtual machines have identical values for O_i where the common value is O_{vm}, then the equation can be simplified to

$$ebr_{off} = (R_{cpu} - O_{vmm} - n * O_{vm})/O_b \tag{6.2}$$

This is a reasonable assumption because each virtual machine is running the same operating system and software. When the policy engine is turned on, we must take into consideration the CPU resources used by the policy engine. Because we are only considering static workloads here, there should be no load balancing. Thus, it is not necessary to consider load migrating to or from virtual machines. The expected number of benchmark runs when the policy engine is turned on is equal to the following equation.

$$ebr_{on} = (R_{cpu} - O_{vmm} - (n+1) * O_{vm} - O_{lb})/O_b \tag{6.3}$$

The actual number of completed benchmark runs will be an integer value, whether the policy engine is on or off. If we have n virtual machines and each virtual machine has almost finished an iteration, then the actual number of benchmark runs that are completed will be n less than the result of the above equation rounded down. The greatest number of benchmark runs will happen when the sum total of the CPU resources spent by the virtual machines on benchmarks currently being executed is less than O_b. This means that the actual number of benchmark runs, abr, is described

by the following equation:

$$\lfloor (R_{cpu} - O_{vmm} - \sum_{i=1}^{n} O_i)/O_b \rfloor - n \leq abr \leq \lfloor (R_{cpu} - O_{vmm} - \sum_{i=1}^{n} O_i)/O_b \rfloor \quad (6.4)$$

which can be rewritten as:

$$\lfloor ebr \rfloor - n \leq abr \leq \lfloor ebr \rfloor \quad (6.5)$$

The above equation can be applied for both ebr_{off} and ebr_{on}, yielding a range of values for abr_{off} and abr_{on}.

The difference in the number of completed benchmark runs when the policy engine is turned on and when it is turned off depends on the ratio of $O_{vm} + O_{lb}$ and O_b. When $O_{vm} + O_{lb}$ is much larger than O_b the overhead of running the policy engine keeps the other virtual machines from completing several benchmark runs. On the other hand, when O_b is much larger than $O_{vm} + O_{lb}$ the overhead from running the policy engine probably did not effect the number of completed benchmark runs. Remember that O_b is constant for any given benchmark, but $O_{vm} + O_{lb}$ will continue to increase as long as a benchmark runs. Clearly, for a static workload that does not benefit from load balancing, we should expect a performance penalty for running the policy engine. The magnitude of the performance penalty depends on the size of $O_{vm} + O_{lb}$, as well as the length of time the benchmark was allowed to run.

6.2 Dynamic Throughput Workloads

Now we analyze the case where some computers are overloaded while other computers are underutilized at some periods of time during the benchmark. Our analysis targets this case because it is under these circumstances that dynamic load balancing could be beneficial. The case where all machines are always overloaded is merely a static workload. Further, the case where all machines are always underutilized will derive

no benefit from load balancing. We will also not be hampered by load balancing when the resources used by the policy engine are idle.

There are many relevant pieces of information when analyzing both static and dynamic workloads. Thus, much of the notation for dynamic load balancing will be similar. Differences exist because we must now account for the different workloads of each CPU. We must also take into consideration the overhead of migrating loads between hosts. Let R_i be the CPU resources of the i^{th} CPU. Let O_{vmm} be the overhead of running the virtual machine monitor. The overhead of running a virtual machine is O_{vm}. Let I_i be the number of idle cycles for a given CPU. Finally, let O_b be the CPU resources needed to complete a workload.

Using those symbols,the following equation describes the estimated number of completed benchmark runs when the policy engine is turned off:

$$ebr_{off} = (R_i - O_{vmm} - n * O_{vm} - I_i)/O_b \tag{6.6}$$

Note that the value of I_i is the only difference between this equation and the static workload equation discussed previously. In fact, the magnitude of the differences between I_i for each host determines how unbalanced the workload is. When loads are balanced the differences between I_i for any two hosts will be small; when loads are imbalanced the difference will be large.

However, when the policy engine is turned on, the following equation models the behavior of the system:

$$ebr_{on} = (R_i - O_{vmm} - n * O_{vm} - LB)/O_b \tag{6.7}$$

In the above equation LB is the overhead for migrating a virtual machine between hosts when the load is imbalanced. When the load is balanced it is 0. Comparing the two equations makes it clear that dynamic load balancing of virtual machines will

only be effective when the cost of balancing the system is less than the idle cycles that were wasted because the system was imbalanced.

6.3 Actual Results

Again we show a summary of our data in Table 6.1. One trend shown in the data is related to overhead when a static workload is running. It seems that the overhead is mostly caused by the increased CPU usage of Dom0. In every case with the policy engine turned on, the increase in CPU usage of Dom0 was much greater than the total CPU usage of the virtual machine running the policy engine.

We tested to see if the increase in usage in Dom0 was being caused by spurious virtual machine migrations. To test this, we statistically compared virtual machine migrations between hosts and the overhead of Dom0. Even with static workloads, the policy engine will periodically detect a "load imbalance", with multiple workloads ending at the same time on the same host. When that happens, work is wasted because the virtual machine must be migrated back or the load will be imbalanced. Future work could address these problems by coming up with load balancing algorithms for virtual machines that are more resistant to random noise.

For our data, spurious migrations had no statistical impact on the percentage of the CPU used by Dom0. Without further experimentation, it is not possible to narrow down what is causing the increased CPU usage on the Dom0 virtual machines. However, when examining the figures, it is clear that the increase in CPU usage on Dom0 is around an order of magnitude greater than the CPU usage of the policy engine. For the purposes of our work it is not critically important that we identify the sources of all overhead. Our thesis is to demonstrate the feasibility of dynamic load balancing with virtual machines. Future work on this subject may be able to better identify some of the sources of overhead that are not identified at this time.

Granted, our model is simple. Our results roughly fit the model that we created, however. We predicted two sources of overhead for the static workloads: resources used directly by the policy engine and resources used by Dom0 to run another virtual machine. It appears that the resources used by Dom0 to run another virtual machine vastly exceed the resources used by the policy engine itself.

Table 6.1: Benchmarks Measuring Static Workloads

	Runs	VM%	Dom0%	Idle%	PE%
Gap - Policy	243.4	94.51%	7.11%	1.23%	0.09%
Gap - No Policy	247.6	96.27%	6.35%	0.77%	0.00%
Gcc - Policy	437.3	94.37%	6.54%	0.56%	0.07%
Gcc - No Policy	438.6	94.58%	6.25%	0.56%	0.00%
Mesa - Policy	206.0	95.41%	6.80%	1.14%	0.06%
Mesa - No Policy	209.0	96.94%	6.32%	0.52%	0.00%
Vpr - Policy	217.3	95.34%	6.65%	1.29%	0.06%
Vpr - No Policy	218.6	96.10%	6.32%	1.56%	0.00%

The data from the dynamic test runs also had some interesting patterns. One observation is that throughput increases for stable workloads using dynamic load balancing. Every single benchmark that we ran had higher throughput for longer benchmark runs than static load balancing. The exact point at which dynamic load balancing helped rather than hurt differed from benchmark to benchmark. Based on the profiling numbers we collected, it appears that the increase in performance is due to a decrease in the amount of processing needed by Dom0. Another contributing factor is a decrease in the amount of time the processor was idle.

When you consider that the overhead of migrating a virtual machine to a new host is part of Dom0, it is interesting that Dom0 would have less overhead for dynamic load balancing when compared to static load balancing for any benchmark run. Our current hypothesis is that Dom0 would be less for some benchmark runs because having an overloaded CPU must trigger some action in Dom0, which has a large overhead.

We also noted that for benchmark runs that would pick new virtual machines frequently, dynamic load balancing hurt throughput. However, when the benchmark was long and stable, the overhead of Dom0 was much higher and the amount of time the CPU spent idling increased. Most likely the increase in overhead from Dom0 is directly caused by the overhead of migrating virtual machines between hosts.

6.4 Summary of Analysis

We built a model of our expected benchmark results. We expect to see the following:

- Small performance penalty for running policy engine with static workloads

For dynamic workloads:

- Performance increases when overhead of policy engine and migrating workload is low
- Performance increase will be small or nonexistent if load imbalances cause idle cycles.
- Running the policy engine causes a performance penalty if the overhead of migrating the workload and the running the policy engine is greater than the amount of wasted work from idle cycles.

From the actual results:

- The overhead of running our policy engine on a static workload is quite small.
- Though statistically significant, the practical difference was only 1% on average.
- If load changes quickly, static load balancing provides the best performance.
- Dynamic load balancing was most effective for longer benchmarks.

Chapter 7

Related Work

Much work has been published about virtual machines. We read numerous papers whose work helped us define our research parameters and design our experiments. These papers are particularly worth mentioning:

Vachon and Teresco recently published work on automated redistribution of virtual machines running under Xen [19]. They benchmark two load balancing algorithms, both of which have significant differences from the CSLB algorithm that we use in this work. One major difference between our modified CSLB algorithm and the algorithms used by Vachan and Teresco is that their algorithm is distributed.

Vachon and Teresco named their first algorithm the *Simple Balancing Algorithm*. This algorithm is decentralized, and works in the sender-initiated case by migrating the virtual machine with the highest CPU demands to one of two locations. A virtual machine will be migrated either to a host where the total load is below some threshold or to a host with fewer total virtual machines.

The second algorithm used by Vachon and Teresco is called the *P-loss Balancing Algorithm*. This algorithm is based on Bayesian utility theory and attempts to measure how much "productivity" is lost if a virtual machine were to be migrated compared to a hypothetical scenario where each virtual machine has as much processing power as it needs. Before migration occurs, the load balancing agent computes the expected state of the system for each possible migration. The lost productivity

for each possibility is then measured and the choice with the lowest amount of lost productivity is selected.

Vachon and Teresco's work also included an attempt to create a simulator for load balancing on Xen. Unfortunately, though the simulator appeared to be accurate for small runs, the numbers returned by the simulator for large runs were not realistic. Vachon and Teresco acknowledge that they probably have a subtle bug in their simulator's code that they were unable to find.

Because we gathered our results before Vachon and Teresco's work was published, we did not design our tests in a way that makes a comparison of our work and their work easy. In their work they are more interested in balancing CPU load, while we are more interested in increasing CPU throughput. In fact, in their paper they do not print any raw performance numbers, and instead print graphs showing CPU load over time. Based on a description of the algorithms used by Vachon and Teresco, it appears that their "simple" algorithm is probably similar to the algorithm that we used; their "p-loss" algorithm is probably inferior. We assume that this is so because their simple algorithm is similar to the algorithm we used, and it outperformed their p-loss algorithm.

The Xen team announced plans to add a cluster management tool that would allow for load balancing based on processor, network, or memory resources [7]. The tool would also accommodate automatically 'evacuated' virtual machines in the event that a host is being turned off. Virtual machines would then be automatically reassigned to other machines nearby. Unfortunately, based on messages posted to the Xen mailing list, the Xen team has announced they are no longer working in this direction.

Ganeti is another virtual server management tool built on top of the Xen virtual machine monitor. Ganeti oversees the management of virtual machines on the host by taking over the creation of virtual disks. It also oversees the installation

of operating systems, as well as startup, shutdown, and failover between physical hosts. Ganeti specifically does not support live migration between hosts.

Another notable paper relating to virtual machines was written by Pelleg et al. Their research combines monitoring software on the host with machine learning to detect failures inside of virtual machines [14]. The benefit of this method is that the monitoring does not occur inside the virtual machine. Thus, the monitoring programs are isolated from errors caused by software running inside of the virtual machine.

Chen et al have recently created a scheme using virtual machine monitors to add another layer of protection for data in commodity operating systems [6]. Their approach uses 'multi-shadowing' to show the data when accessed by the operating system, or an encrypted page when accessed by a user-mode application.

Jones et al add a security service to the virtual machine monitor which attempts to detect stealth rootkits [10]. They combine implicit information available to the virtual machine monitor with statistical inference techniques, like hypothesis testing and linear regression, in an attempt to accurately determine when a rootkit is present in a virtual machine.

Rosenblum et al use virtual machine monitors to protect against buffer overrun attacks [16]. They extend previous work related to context-sensitive page mappings. Knowing whether page mappings depend on memory accessed for an instruction or for data can prevent buffer overrun attacks. In particular, Rosenblum et al extend this work by implementing it inside of a virtual machine monitor.

Ongaro et al evaluate the Xen scheduling algorithm by measuring performance while running multiple guest domains that are running different types of applications [13]. Some of the applications are processor intensive, some are bandwidth intensive, and others are latency sensitive. Their benchmarks are run with each of the 11 different scheduler configurations available within Xen. These configurations identify some of the scheduling problems that exist in the Xen virtual machine monitor.

These papers are a small sample of the work that has been recently published about virtual machines.

Chapter 8

Future Work and Conclusions

8.1 Future Work

Using our work as a starting point, future research could go in many directions. One such direction would be to make changes to the policy engine so that it is a service running on each Xen host, instead of an external entity running in a virtual machine by itself. If the policy engine was running as a local process on each Xen host, the policy engine would work a little differently than our current implementation. The user would create a policy and decide which hosts would participate. When the user starts the policy engine, the policy would be pushed onto all of the other Xen hosts. The algorithm would be distributed and run on each of the hosts running Xen. It is likely that this approach would have better performance than the policy engine that we created because a process is ighter weight than a full virtual machine.

Future work could also run our benchmark on more than two machines. Environments with more than two machines will be more complicated than the simple environments that we tested on here. We expect the results would be similar to those we saw with just two machines, but it is possible that with more than two machines we will uncover other interesting results.

Another direction researchers could go would be rerunning our benchmarks with a different dynamic load balancing algorithm. Our current policy engine design makes it a little difficult to change out one policy for another. Running our results

with different algorithms or changing the architecture of the policy engine so that it is easier to change policies may unearth some interesting results.

It is possible to balance load using different resources. Our work only load balanced with respect to the CPU. Future work may explore dynamic load balancing algorithms that take other resources The researchers could then evaluate the effectiveness of those algorithms in load balancing virtual machines.

Also, future work could compare dynamic load balancing of virtual machines under different workloads. We ran many different benchmarks and we noticed some differences between the results; it may be interesting to explore what is causing those differences. In particular, it would be interesting to rerun our experiments with workloads that more closely resemble the loads of servers in data centers.

The algorithms used to migrate virtual machines to new hosts will be most efficient when virtual machines write on a small number of pages. An interesting direction for future work might investigate how the memory access patterns of the virtual machines effects the overhead of migrating the virtual machine to a new host. Or researchers might modify the kernel of an operating system that runs on Xen so that the operating system will migrate more easily to new hosts.

8.2 Conclusions

In this work we constructed a policy engine so we could apply dynamic load balancing algorithms to virtual machines. When we started this project, there was no published work on dynamic load balancing of virtual machines. We evaluated the performance of our policy engine in multiple ways. We ran static workloads with the policy engine turned both on and off to determine the overhead of running our policy engine. We also ran multiple dynamic workloads of different lengths to determine the performance of dynamic load balancing on virtual machines. We also wanted to see how performance is effected by the lengths of the workloads. We analyzed our workload

and wrote equations to tell what we generally expected of our results. Finally, we analyzed our actual results.

We found several interesting results. The first is that running our policy engine along side a static workload had only a small impact on throughput. The second is that for workloads that do not change quickly, dynamic load balancing seems to always outperform static load balancing for virtual machines.

Bibliography

[1] Specweb2005. World Wide Web electronic publication, 2005.

[2] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, Jun 2003.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Aug 2000.

[4] 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. *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, Oct 2003.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5), Dec 1997.

[6] Xiaoxin Chen, Tal Garfinkel, E Lewis, Pratap Subrahmanyam, Carl Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Mar 2008.

[7] C Clark, K Fraser, S Hand, J Hansen, and E Jul. Live migration of virtual machines. *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273—286, 2005.

[8] D Eager, E Lazowska, and J Zahorjan. The limited performance benefits of migrating active processes for load sharing. *ACM SIGMETRICS Performance Evaluation Review*, 16(1), May 1988.

[9] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[10] Stephen Jones, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Mar 2008.

[11] Youran Lan and Ting Yu. A dynamic central scheduler load balancing mechanism. *Computers and Communications, 1995*, pages 734–740, May 1995.

[12] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[13] Diego Ongaro, Alan Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Mar 2008.

[14] Dan Pelleg, Muli Ben-Yehuda, Rick Harper, Lisa Spainhower, and Tokunbo Adeshiyan. Vigilant: out-of-band detection of failures in virtual machines. *ACM SIGOPS Operating Systems Review*, 42(1), Jan 2008.

[15] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5), Jul 2004.

[16] Nathan Rosenblum, Gregory Cooksey, and Barton Miller. Virtual machine-provided context sensitive page mappings. *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Mar 2008.

[17] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5), Jul 2008.

[18] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[19] Travis F Vachon and James D Teresco. Automated dynamic redistribution of virtual operating systems under the xen virtual machine monitor. *Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference: parallel and distributed computing and networks*, pages 190–195, May 2007.

[20] Carl Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI), Dec 2002.

[21] Jerrell Watts and Stephen Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235—??, Feb 1998.