UNIVERSITY OF CALIFORNIA, SAN DIEGO

Flexible and Efficient Resource Management in a Virtual Cluster Environment

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science

by

Marvin McNett II

Committee in charge:

Professor Geoffrey M. Voelker, Chair
Professor Rene L. Cruz
Professor Ramesh Rao
Professor Stefan Savage
Professor Amin Vahdat

2008

The dissertation of Marvin McNett II is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I dedicate this dissertation to my beautiful wife Karin, who sacrificed more for its completion than most could imagine.

Chapter 2, in part, is a reprint of material as it appears in the USENIX LISA Conference, 2007, McNett, Marvin; Gupta, Diwaker; Vahdat, Amin; Voelker, Geoffrey M.. The dissertation author was the primary investigator and author of this paper.

VITA

2008	Doctor of Philosophy in Computer Science University of California, San Diego San Diego, CA, USA
1999	Master of Arts in Mathematics University of Kansas Lawrence, KS, USA
1994	Bachelor of Science in Aerospace Engineering University of Kansas Lawrence, KS, USA

PUBLICATIONS

"Usher: An Extensible Framework for Managing Clusters of Virtual Machines,"
Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker, In *Proceedings of the 21st Large Installation System Administration Conference*, pp. 169-183, November 2007.

"To Infinity and Beyond: Time-Warped Network Emulation,"
Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker, In *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, pp. 87-100, May 2006.

"Access and Mobility of Wireless PDA Users,"
Marvin McNett and Geoffrey M. Voelker, In *ACM Mobile Computing and Communications Review*, Volume 9, Number 2, pp. 40-55, April 2005.

FIELDS OF STUDY

Computer Systems and Networking

Cluster Management

ABSTRACT OF THE DISSERTATION

Flexible and Efficient Resource Management in a Virtual Cluster Environment

by

Marvin McNett II
Doctor of Philosophy in Computer Science
University of California, San Diego, 2008
Professor Geoffrey M. Voelker, Chair

Virtual machine (VM) use in a cluster environment imposes many challenges upon a cluster administrator. As the number of VMs across a site grows, manually tracking transient site state information such as resource availability and VM locations and status while enforcing policies for running large numbers of VMs across a cluster becomes increasingly difficult. In this dissertation I focus on VM use in clusters and consider the management and efficiency problems that arise in this unique environment. I present Usher, a virtual machine management system designed to impose few constraints upon the computing environment under its management. Usher enables administrators to choose how their virtual machine environment will be configured and the policies under which it will be managed. Usher cluster administrators can push basic virtual cluster management tasks such as VM start and stop out to the users (virtual cluster creators) themselves, reducing administrator workload and allowing users to create virtual clusters on demand. The modular design of Usher allows for alternate implementations for authentication, authorization, infrastructure handling, logging, and virtual machine scheduling. The design philosophy of Usher is to provide an interface whereby users and administrators can request virtual machine operations while delegating administrative tasks and policy enforcement for these requests to modular plugins. I present an Usher scheduling plugin designed to map virtual machines onto physical

machines such that an arbitrarily defined utility is optimized. I discuss possible cluster scheduling goals and present a representative cluster scheduling problem called Fair Maximum Utilization (FMU). Exploration of scheduling heuristics of varying levels of sophistication applied to FMU suggest that those which make better VM resource demand predictions and only slight schedule adjustments work well in general.

Chapter 1

Introduction

The proliferation of clusters of powerful, multi-processor and multi-core servers has many cluster administrators looking for new ways to harness the full potential of their computing environments. These administrators often look to virtualization as a means to increase their cluster utilization in a flexible manner. Though promises of increased hardware utilization and flexibility in resource provisioning frequently motivate virtual machine (VM) use in clusters of commodity computers, many administrators influenced by these promises are not prepared for the difficulty of deploying and managing such a complex computing environment. As the number of VMs across a site grows, manually tracking transient site state information such as resource availability and VM locations and status while enforcing policies for running large numbers of VMs across a cluster becomes increasingly onerous. Consequently, to the cluster administrator, hypervisor features and performance have become less important than the ability to effectively manage large numbers of virtual machines across a site. To this end, I designed and implemented Usher, a virtual cluster management system which substantially reduces the administrative burden of managing virtual machines across a physical cluster. In this dissertation I present Usher, an extensible framework for managing clusters of virtual machines, and demonstrate its ability to significantly reduce management and efficiency problems inherent in virtual cluster environments.

1.1 Motivation

Cluster administrators often turn to VMs as a remedy for rigid, course grained resource sharing models and low resource utilization. VMs provide a convenient mechanism for partitioning computing resources among users at nearly any granularity. Once partitioned, VM priority and placement is dictated by the administrator, providing the flexibility to support a wide range of cluster operational goals. In addition, the ability to multiplex several VMs on each physical machine enables tapping otherwise idle resources, greatly increasing overall utilization.

Indeed, my own experience led me to seek better solutions to sharing cluster computers among members of the Systems and Networking (SysNet) group at UCSD; a model of distributing computing resources to users at the granularity of physical machines led to very poor resource utilization and starved users. Due to the nature of research in the SysNet group, many required clusters of machines to carry out their research. A physical machine checkout system providing remote console and power cycle abilities for each machine quickly exhausted all computing resources. At the same time, cluster-wide resource utilization was less than five percent.

With approximately 50 active users, over 200 physical machines were insufficient for the SysNet group. As surprising as this may seem, the problem is a result of researchers' reluctancy to return machines which they have spent significant time setting up. Cluster setup and experimentation can be quite time consuming. Researchers want to ensure they are completely finished with their work before releasing machines back to the group.[1] Returning machines sometimes took years.[2] Unfortunately, acquiring additional hardware did not alleviate the starvation problem. New machines were typically claimed before arriving.

Though just an example, other highly experienced persons in industry and academia have confirmed [Eus07, Ham07, Kis07, O'H08] that they have very similar

[1] A system I designed to pull disk images from checked out machines (in about five minutes) when researchers would not be using them and put the images back (again in about five minutes) when ready to resume their experiments went unused.

[2] Often, only because the student graduated.

problems to those faced by the SysNet group. Whether by researchers or groups in a larger company, these resource sharing issues arise wherever clusters are shared between self-interested parties. All of these individuals have expressed interest in virtualization as a possible solution to their resource allocation problems.

As for SysNet, most users in need of a cluster (or even a single machine) do not actually need physical machines, but rather the abstraction of physical machines, which virtual machines provide. Therefore, SysNet chose to use VMs to alleviate their resource sharing woes. In making that decision, however, I learned that manually managing more than a few dozen VMs across a cluster is extremely time consuming and arduous to track. Furthermore, proper monitoring of resource availability is difficult, making VM placement hit-or-miss (e.g., placing two CPU hungry VMs on the same node while other nodes remain nearly idle is likely to eventually happen). Clearly, such an environment requires a VM management system. Similar to traditional operating systems, this system must support a wide range of management policies and operational goals in an efficient manner. I implemented Usher to meet these requirements.

1.2 Contributions

In this dissertation I focus on VM use in clusters and present solutions to management and efficiency problems that arise in this unique environment. To address manageability, I designed and implemented Usher, a virtual cluster management system designed to substantially reduce the administrative burden of managing cluster resources while simultaneously improving the ability of users to request, control, and customize their resources and computing environment. Based upon feedback from users and administrators from another Usher installation, I refactored Usher to be easily extensible. This refactoring greatly improved the flexibility of Usher to adapt to different computing environments. The current version of Usher imposes few constraints upon the computing environment under its management. No two sites have identical hardware and software configurations, user and application requirements, or service infrastruc-

tures. To facilitate its use in a wide range of environments, Usher combines a core set of interfaces that implement basic mechanisms, clients for using these mechanisms, and a framework for expressing and customizing administrative policies through extensible modules, or plugins.

Efficiency encompasses both virtual cluster setup time and operation. For setup time, I consider the time devoted to virtual cluster creation and destruction. Usher provides a single control point from which administrators can manage their entire virtual cluster environment while pushing many of the mundane tasks involved with virtual cluster administration out to the users (cluster creators) themselves. In this way, a minimal amount of setup time is required by Usher administrators. Usher makes pushing setup responsibility out to users possible through an extensible plugin system. The plugin system enables administrators to enforce arbitrary management policies or support arbitrary operational goals. This design places the responsibility of enforcing these policies on Usher itself. The administrator is not required to approve all user requests. These features (i.e., central control point and setup delegation) act to increase administrator efficiency in managing virtual cluster installations.

From my experience, deploying Usher transformed cluster management from taking well over two hours to create a cluster of 20 VMs, complete with DNS entries and user LDAP authentication, to around two minutes for a cluster of approximately 100 VMs.[3] Usher enabled users to experiment with much larger personal clusters. Many users created long running virtual clusters of over 100 nodes.[4] The largest physical cluster by a SysNet user was 50 nodes for only a few days.[5] Several months saw over 500 virtual machines created by SysNet users on a 25 node Usher cluster.

To address operational efficiency, I implemented an Usher scheduling plugin. This plugin seeks to provide a mechanism for enforcing the operational goals under which administrators wants their sites to run. Enforcing these operational goals is ac-

[3] Actually, this is the time required for all the VMs to be up and running. Administrator time is only a few seconds since creating such a cluster in Usher can be handled with a single command.

[4] Many of these would have been larger had more physical machines been dedicated to the Usher installation.

[5] That involved several days of asking other SysNet users to “borrow” their checked out physical machines for a few days to meet a paper deadline.

complished by scheduling VMs across a site so as to maximize an administrator defined utility of the mapping of VMs to physical machines. Utility is expressed as a mathematical function over the set of feasible assignments of VMs to physical machines. This design enhances flexibility as well. Administrators can support nearly arbitrary operational goals by expressing those goals as utility functions. Usher's scheduling plugin then works to maintain those goals as the system runs via monitoring and transparently migrating VMs when goals are not being met. This design pushes the job of maintaining a site's operational goals onto Usher. Administrators are not required to continuously monitor their site's status to keep it in line with these operational goals, greatly decreasing the time they spend maintaining their sites while increasing their ability to enforce a wide range of operational goals.

To demonstrate the effectiveness of the Usher scheduling plugin, I present several possible cluster scheduling goals and define a representative cluster scheduling problem called Fair Maximum Utilization (FMU). The goal of FMU is to fairly maximize overall cluster resource usage in a best-effort fashion (i.e., no guarantees, or service level agreements (SLAs)). In FMU, fairness is measured as the sum over all resources of Jain's Fairness Index of resource allocations to VMs wanting more of that particular resource. In other words, FMU tries to give all VMs as much of a resource as it desires. When that is not possible, it fairly divides the resource among those VMs wanting more of that resource. This is an appropriate scheduling goal for SysNet, where users are allowed to create clusters of any size and use them as desired.

FMU is an instance of a multiple objective combinatorial optimization problem. I prove that FMU is in the class of NP-hard problems. As a result, deterministic, polynomial-time algorithms for solving this problem are not known to exist. Therefore, solution strategies which approximate optimal solutions are required.

I designed heuristics of varying levels of sophistication for optimizing arbitrarily defined utility functions over the set of all feasible assignments of VMs to physical machines. I then experimentally evaluated these heuristics with FMU scheduling to establish each heuristic's suitability to scheduling in environments with a range of VM

dynamism levels. My results indicate that schedulers which make better VM resource demand predictions and only slight schedule adjustments work well in general.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the Usher system, details its design and implementation, and discusses its use in two different cluster environments. Chapter 3 presents the virtual machine scheduling problem, defines our canonical FMU scheduling problem, proves FMU's inclusion in the set of NP-hard problems, and presents heuristics for finding good solutions to the FMU scheduling problem. Chapter 4 presents experimental results for my heuristics applied to a range of virtual cluster environment workload characteristics. Chapter 5 discusses future directions and Chapter 6 concludes. There are also three appendices. Appendix A presents an overview of the Usher source code, Appendix B lists all core Usher events, and Appendix C discusses writing Usher extensions (clients, plugins, and hypervisor wrappers).

Chapter 2

Usher: An Extensible Framework for Managing Clusters of Virtual Machines

Here, we present the Usher virtual machine management system. Usher provides an extensible framework and client API enabling plugins and custom applications to control virtual clusters. In addition to managing VMs across a site, Usher enables us to experiment with complex virtual machine management policies. In Chapter 3, we use Usher to study VM scheduling heuristics for efficient resource utilization.

In this chapter, we provide an overview of the Usher system architecture and implementation. We then present specific examples of Usher plugins and applications. The chapter ends with a discussion of two Usher installation with very different management goals.

2.1 Introduction

Usher is a virtual cluster management system designed to substantially reduce the administrative burden of managing cluster resources while simultaneously improving the ability of users to request, control, and customize their resources and computing environment. System administrators of cluster computing environments face a number of imposing challenges. Different users within a cluster can have a wide range of

computing demands, spanning general best-effort computing needs, batch scheduling systems, and complete control of dedicated resources. These resource demands vary substantially over time in response to changes in workload, user base, and failures. Furthermore, users often need to customize their operating system and application environments, substantially increasing configuration and maintenance tasks. Finally, clusters rarely operate in isolated administrative environments, and must be integrated into existing authentication, storage, network, and host address and name service infrastructure.

Usher balances these imposing requirements using a combination of abstraction and architecture. Usher provides a simple abstraction of a logical cluster of virtual machines, or virtual cluster. Usher users can create any number of virtual clusters (VCs) of arbitrary size, while Usher multiplexes individual virtual machines (VMs) on available physical machine hardware. By decoupling logical machine resources from physical machines, users can create and use machines according to their needs rather than according to assigned physical resources.

Architecturally, Usher is designed to impose few constraints upon the computing environment under its management. No two sites have identical hardware and software configurations, user and application requirements, or service infrastructures. To facilitate its use in a wide range of environments, Usher combines a core set of interfaces that implement basic mechanisms, clients for using these mechanisms, and a framework for expressing and customizing administrative policies in extensible modules, or plugins.

The Usher core implements basic virtual cluster and machine management mechanisms, such as creating, destroying, and migrating VMs. Usher clients use this core to manipulate virtual clusters. These clients serve as interfaces to the system for users as well as for use by higher-level cluster software. For example, an Usher client called *ush* provides an interactive command shell for users to interact with the system. We have also implemented an adapter for a high-level execution management system [ABD+07], which operates as an Usher client, that creates and manipulates virtual clusters on its own behalf.

Usher supports customizable modules for two important purposes. First, these modules enable Usher to interact with broader site infrastructure, such as authentication, storage, and host address and naming services. Usher implements default behavior for common situations; e.g., newly created VMs in Usher can use a site's DHCP service to obtain addresses and domain names. Additionally, sites can customize Usher to implement more specialized policies; at UCSD, an Usher VM identity module allocates IP address ranges to VMs within the same virtual cluster.

Second, pluggable modules enable system administrators to express site-specific policies for the placement, scheduling, and use of VMs. As a result, Usher allows administrators to decide how to configure their virtual machine environments and determine the appropriate management policies. For instance, to support a general-purpose computing environment, administrators can install an available Usher scheduling and placement plugin that performs round-robin placement of VMs across physical machines and simple rebalancing in response to the addition or removal of virtual and physical machines. With this plugin, users can dynamically add or remove VMs from VCs at any time without having to specify service level agreements (SLAs) [AFF+01, RRX+06, WSVY07], write configuration files [Beg06], or obtain leases on resources [CIG+03, GIYC06]. With live migration of VMs, Usher can dynamically and transparently adjust the mapping of virtual to physical machines to adapt to changes in load among active VMs or the working set of active VMs, exploit affinities among VMs (e.g., to enhance physical page sharing [Wal02]), or add and remove hardware with little or no interruption.

Usher enables other powerful policies to be expressed, such as power management (reduce the number of active physical machines hosting virtual clusters), distribution (constrain virtual machines within a virtual cluster to run on separate nodes), and resource guarantees. Another installation of Usher uses its cluster to support scientific batch jobs running within virtual clusters, guarantees resources to those jobs when they run, and implements a load-balancing policy that migrates VMs in response to load spikes [CGK+06].

Usher is a fully functional system. It has been installed in multiple cluster computing environments including UCSD, University of Hawaii, and the Russian Research Center in Kurchatov, Russia. At UCSD, Usher has been in production use since January 2007. It has managed up to 400 virtual machines in virtual clusters across 62 physical machines.

2.2 Related Work

Since the emergence of widespread cluster computing over a decade ago [ACPtNT95, Mer], many cluster configuration and management systems have been developed to achieve a range of goals. These goals naturally influence individual approaches to cluster management. Early configuration and management systems, such as Galaxy [VD00], focus on expressive and scalable mechanisms for defining clusters for specific types of service, and physically partition cluster nodes among those types.

More recent systems target specific domains, such as Internet services, computational grids, and experimental testbeds, that have strict workload or resource allocation requirements. These systems support services that express explicit resource requirements, typically in some form of service level agreement (SLA). Services provide their requirements as input to the system, and the system allocates its resources among services while satisfying the constraints of the SLA requirements.

For example, Océano provides a computing utility for e-commerce [AFF+01]. Services formally state their workload performance requirements (e.g., response time), and Océano dynamically allocates physical servers in response to changing workload conditions to satisfy such requirements. Cluster-on-Demand (COD) performs resource allocation for computing utilities and computational grid services [CIG+03]. COD implements a virtual cluster abstraction, where a virtual cluster is a disjoint set of physical servers specifically configured to the requirements of a particular service, such as a local site component of a larger wide-area computational grid. Services specify and request resources to a site manager and COD leases those resources to them. Finally, Emulab

provides a shared network testbed in which users specify experiments [WLS+02]. An experiment specifies network topologies and characteristics as well as node software configurations, and Emulab dedicates, isolates, and configures testbed resources for the duration of the experiment.

Rocks and Rolls provide scalable and customizable configuration for computational grids [BKSP04, SCB04]. Though these systems have eased the pain of deploying and managing clusters, the decision to use such a system comes at the cost of much flexibility. While many groups purchase cluster hardware for tasks amenable to traditional static installation and configuration of a site's computing infrastructure (e.g., batch processing) or horizontal scale out of services, others require more flexible computing models.

The recent rise in virtual machine monitor (VMM) popularity has naturally led to systems for configuring and managing virtual machines. For computational grid systems, for example, Shirako extends Cluster-on-Demand by incorporating virtual machines to further improve system resource multiplexing while satisfying explicit service requirements [GIYC06], and VIOLIN supports both intra- and inter-domain migration to satisfy specified resource utilization limits [RRX+06]. Sandpiper develops policies for detecting and reacting to hotspots in virtual cluster systems while satisfying application SLAs [WSVY07], including determining when and where to migrate virtual machines, although again under the constraints of meeting the stringent SLA requirements of a data center.

On the other hand, Usher provides a framework that allows system administrators to express site-specific policies depending upon their needs and goals. By default, the Usher core provides, in essence, a general-purpose, best-effort computing environment. It imposes no restrictions on the number and kind of virtual clusters and machines, and performs simple load balancing across physical machines. We believe this usage model is important because it is widely applicable and natural to use. Requiring users to explicitly specify their resource requirements for their needs, for example, can be awkward and challenging since users often do not know when or for how long they will

need resources. Further, allocating and reserving resources can limit resource utilization; guaranteed resources that go idle cannot be used for other purposes. However, sites can specify more elaborate policies in Usher for controlling the placement, scheduling, and migration of VMs if desired. Such policies can range from batch schedulers to allocation of dedicated physical resources.

In terms of configuration, Usher shares many of the motivations that inspired the Manage Large Networks (MLN) tool [Beg06]. The goal of MLN is to enable administrators and users to take advantage of virtualization while easing administrator burden. Administrators can use MLN to configure and manage virtual machines and clusters (*distributed projects*), and it supports multiple virtualization platforms (Xen and User-Mode Linux). MLN, however, requires administrators to express a number of static configuration decisions through configuration files (e.g., physical host binding, number of virtual hosts), and supports only coarse granularity dynamic reallocation (manually by the administrator). Usher configuration is interactive and dynamic, enables users to create and manage their virtual clusters without administrative intervention, and enables a site to globally manage all VMs according to cluster-wide policies.

XenEnterprise [xen] from XenSource and VirtualCenter [vmwa] from VMware are commercial products for managing virtual machines on cluster hardware from the respective companies. XenEnterprise provides a graphical administration console, Virtual Data Center, for creating, managing, and monitoring Xen virtual machines. VirtualCenter monitors and manages VMware virtual machines on a cluster as a data center, supporting VM restart when nodes fail and dynamic load balancing through live VM migration. Both list interfaces for external control, although it is not clear whether administrators can implement plugins for enforcing policy, integrating the systems into existing infrastructure, or controlling VMs in response to arbitrary events in the system.

VirtualCenter does provide a mechanism to call arbitrary scripts in response to a limited set of events. However, these run as separate processes from VirtualCenter itself, so they have no access to its internal state. Also, VMWare's Infrastructure Management SDK provides functionality similar to that provided by the Usher client API.

However, this SDK does not provide the tight integration with VMWare's centralized management system that plugins do for the Usher system.

In addition, these commercial offerings are all tied to managing a single VM product, whereas Usher is designed to interface with any virtualization platform that exports a standard administrative interface. Furthermore, they are closed source, proprietary, and infinitely more expensive than the free, open source Usher system.

Enomaly [eno], formerly known as Enomalism, was developed in parallel with Usher. Originally, Enomalism was a free, open source system designed to manage virtual machines using Red Hat's libvirt toolkit [lib]. It has since evolved into a platform for elastic cloud computing. Enomaly aims to provide a system which would enable businesses to offer their own elastic cloud services similar to that of the Amazon Elastic Compute Cloud (EC2) [ec2]. Enomaly is still free and open source. Their revenue is generated by paid support and custom extensions to their product.

Recently, a number of both free and commercial VM management products have been introduced.

Eucalyptus [euc] is an open source system for implementing cloud computing on private clusters. This is similar in spirit to Enomaly. Eucalyptus aims at being compatible with Amazon's EC2 interface.

Kodiak [kod] from Bluebear is a commercial (although free and open source) hypervisor agnostic VM management system with support for "many" hypervisors out of the box. Bluebear claims that Kodiak is "the industry's only application that's both hypervisor-agnostic and cross-platform". However, Kodiak only supports a single hypervisor at the time of this writing. In addition, Bluebear claims that "Kodiak is also extensible through a soon to be published open framework". Apparently, the folks at Bluebear believe in the Usher approach to VM management, but neglect to acknowledge Usher's predating of their system.

ConVirt [conb], originally called "XenMan", is another hypervisor agnostic VM management system released under the GNU Public License. ConVirt provides a convenient graphical management console for managing VMs across multiple PMs. At

the time of this writing, it appears that Convirt does not support plugins or provide a client API to integrate with third-party applications.

Tashi [tas] is a cluster management project proposed by Intel Research. Their goal is to develop a layer of utility software to convert clusters into cloud computers enabling remote users to operate on large, locally stored data sets. Tashi will manage both virtual and physical machines to operate on massive internet-scale datasets without the overhead of moving the data between sites.

Sun Microsystems very recently announced availability of a VM management system named xVM Ops Center [xvm]. This system was designed to manage their Xen based hypervisor called xVM Server across x64 and SPARC systems. The feature set of xVM Ops Center is similar to that of VMware's VirtualCenter.

Finally, Microsoft entered the VM management space with their System Center Virtual Machine Manager [sys]. This system enables configuration and management of virtualized infrastructures running on their Hyper-V hypervisor or VMware ESX Server. Again, the feature set of this commercial offering is very similar to that of VMware's VirtualCenter.

2.3 System Architecture

This section describes the architecture of Usher. It begins with a brief summary of the goals guiding the Usher system design, followed by a high-level overview of the core Usher system. It then describes the purpose and operation of each of the various core system components, and how they interact with each other to accomplish their tasks. It ends with a discussion of how the Usher system accommodates software and hardware failures.

2.3.1 Design Goals

As mentioned, no two sites have identical hardware and software configurations, user and application requirements, or service infrastructures. As a result, Usher

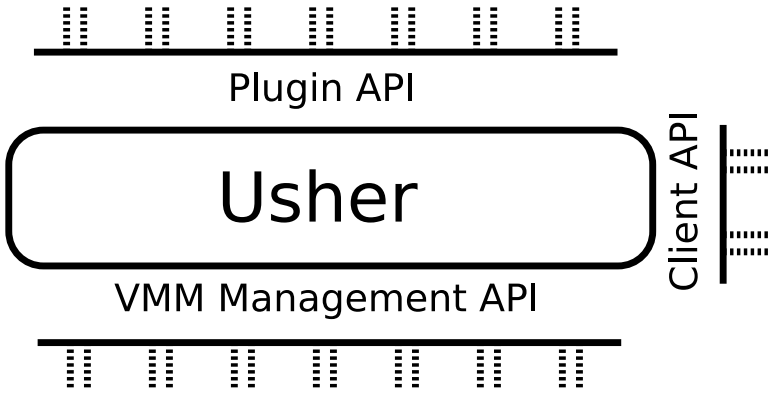

Figure 2.1 Usher Interfaces

was designed as a flexible platform for constructing virtual machine management installations customized to the needs of a particular site.

To accomplish this goal, there were two design objectives for Usher. First, Usher maintains a clean separation between policy and mechanism. The Usher core provides a minimal set of mechanisms essential for virtual machine management. For instance, the Usher core has mechanisms for starting, migrating, and stopping virtual machines, as well as, maintaining and querying the global state of the system.

Second, Usher is designed for extensibility. The Usher core provides three ways to extend functionality, as illustrated in Figure 2.1. First, Usher provides an interface to integrate with different virtual machine managers (VMMs). For instance, while Usher provides a reference implementation for use with the Xen VMM, it is straightforward to implement this adapter for other VMMs. Second, developers can use a Plugin API to enhance Usher functionality. For example, plugins can provide database functionality for persistently storing system state using a file-backed database, or provide authentication backed by local Unix passwords. Plugins can also extend the operations API exposed to connected clients for cases where the core API does not provide functionality desired or required. Third, Usher provides a Client API for integrating with user interfaces and third-party tools, such as the Usher command-line shell (Section 2.5.1) and the Plush execution management system (Section 2.5.2).

2.3.2 System Overview

A running Usher system consists of three main components: local node managers (LNMs), a centralized controller, and clients. Figure 2.2 depicts the core components of an Usher installation.

One LNM runs on each physical node and interacts directly with the VMM to perform management operations such as creating, deleting, and migrating VMs on behalf of the controller. Since Usher was designed to be VMM agnostic, one key role of the LNM is to wrap the administrative API of the underlying VMM, so that VMs of that type can be managed by Usher. The local node managers also collect resource usage data from the VMMs and monitor local events. LNMs report resource usage updates and events back to the controller for use by plugins and clients.

The controller is the central component of the Usher system. It receives authenticated requests from clients and issues authorized commands to the LNMs. It also communicates with the LNMs to collect usage data and manage virtual machines running on each physical node. The controller provides event notification to connected clients and plugins registered to receive notification for a particular event (e.g., a VM has started, been destroyed, or changed state). Plugin modules can perform a wide range of tasks, such as maintaining persistent system-wide state information, performing DDNS updates, or doing external environment preparation and cleanup.

The Usher client library provides an API for applications to communicate with the Usher controller. Essentially, clients submit requests to the controller when they need to manipulate their VMs or request additional VMs. The controller can grant or deny these requests as its operational policy dictates. One purpose of clients is to serve as the user interface to the system, and users use clients to manage their VMs and monitor system state. More generally, arbitrary applications can use the client library to register callbacks for events of interest in the Usher system.

Typically, a few services also support a running Usher system. Depending upon the functionality desired and the infrastructure provided by a particular site, these services might include a combination of the following: a database server for maintaining

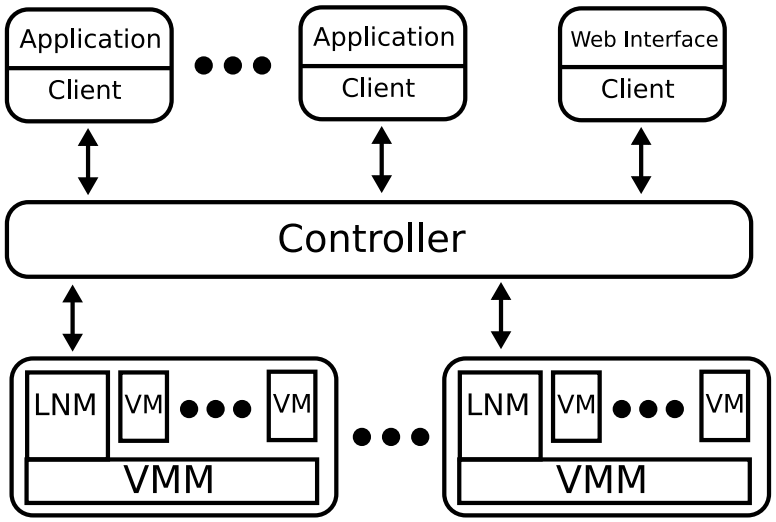

Figure 2.2 Usher Components

state information or logging, a NAS server to serve VM filesystems, an authentication server to provide authentication for Usher and VMs created by Usher, a DHCP server to manage IP addresses, and a DNS server for name resolution of all Usher created VMs. Note that an administrator may configure Usher to use any set of support services desired, not necessarily restricted to the preceding list.

2.3.3 Local Node Managers

The local node managers (LNMs) operate closest to the hardware. As shown in Figure 2.2, LNMs run as servers on each physical node in the Usher system. The LNMs have three major duties: i) to provide a remote API to the controller for managing local VMs, ii) to collect and periodically upload local resource usage data to the controller, and iii) to report local events to the controller.

Each LNM presents a remote API to the controller for manipulating VMs on its node. Upon invoking an API method, the LNM translates the operation into the equivalent operation of the VM management API exposed by the VMM running on the node (i.e., the LNM is essentially an adapter for the VM management API exposed by the underlying VMM). Note that all LNM API methods are asynchronous so that the

controller does not block waiting for the VMM operation to complete. We emphasize that this architecture abstracts VMM-specific implementations — the controller is oblivious to the specific VMMs running on the physical nodes as long as the LNM provides the remote API implementation. As a result, although our implementation currently uses the Xen VMM, Usher can target other virtualization platforms. Further, Usher is capable of managing VMs running any operating system supported by the VMMs under its management.

As the Usher system runs, VM and VMM resource usage fluctuates considerably. The local node manager on each node monitors these fluctuations and reports them back to the controller by request. The report includes resource usage of CPU utilization, network receive and transmit loads (for both bits and packets per second), and disk I/O activity (for both blocks and operations per second) in 1, 5, and 15-minute averages. Typically, these statistics are not readily available from the VMM, leaving the LNM writer charged with the job of collecting and maintain these values.

In addition to changes in resource usage, VM state changes sometimes occur unexpectedly. VMs can crash or even unexpectedly appear or disappear from the system. Detecting these and other related events requires careful monitoring by the local node managers, possibly assisted by VMM support for internal event notification. Administrators can set a tunable parameter for how often the LNM scans for missing VMs or unexpected VMs. If available, the LNM will register callbacks with the VMM platform for other events, such as VM crashes. Otherwise, the LNM will periodically scan to detect these events.

2.3.4 Controller

The controller is the center of the Usher system. It can either be bootstapped into a VM running in the system, or run on a separate server. The controller provides the following:

- User authentication

- VM operation request API
- Global state maintenance
- Event notification

User authentication: Usher uses SSL-encrypted user authentication. All users of the Usher system must authenticate before submitting requests to the system. Administrators are free to use any of the included authentication modules for use with various authentication backends (e.g., LDAP), or implement their own. Information on implementing and registering an authentication module with the Usher system can be found in Appendix C.2.

An administrator can register multiple authentication modules, and Usher will query each in turn. Authentication module registration is specified in the controller's configuration file. This support is useful, for instance, to provide local password authentication if LDAP or NIS authentication fails. Upon receiving a user's credentials, the controller checks them against the active authentication module chain. If one succeeds before reaching the end of the chain, the user is authenticated. Otherwise, authentication fails and the user must retry.

VM operation request API: A key component of the controller is the remote API for Usher clients. This API is the gateway into the system for VM management requests (via RPC) from connected clients. Typically, the controller invokes an authorization plugin to verify that the authenticated user can perform the operation before proceeding. The controller may also invoke other plugins to do preprocessing such as checking resource availability and making placement decisions at this point. Usher calls any plugin modules registered to receive notifications for a particular request once the controller receives such a request.

Usher delegates authorization to plugin modules so that administrators are free to implement any policy or policies they wish and stack and swap modules as the system runs. In addition, an administrator can configure the monitoring infrastructure to

automatically swap or add policies as the system runs based upon current system load, time of day, etc. In its simplest form, an authorization policy module openly allows users to create and manipulate their VMs as they desire or view the global state of the system. More restrictive policies may limit the number of VMs a user can start, prohibit or limit migration, or restrict what information the system returns upon user query.

Once a request has successfully traversed the authorization and preprocessing steps, the controller executes it by invoking asynchronous RPCs to each LNM involved. As described above, it is up to any plugin policy modules to authorize and check resource availability prior to this point. Depending upon the running policy, the authorization and preprocessing steps may alter a user request before the controller executes it. For example, the policy may be to simply "do the best we can" to honor a request when it arrives. If a user requests more VMs than allowed, this policy will simply start as many VMs as are allowed for this user, and report back to the client what action was taken. Finally, if insufficient resources are available to satisfy an authorized and preprocessed request, the controller will attempt to fulfill the request until resources are exhausted.

A powerful feature of the Usher system is the ability for plugins themselves to extend the API exposed to clients by the Controller. For example, one might imagine a plugin which allows for a connected client with sufficient privileges to restart selected LNMs, the controller, or even the entire Usher system. The controller exposes a method which allows for connected clients to call methods exposed by a plugin (see Section 2.4.2).

Global state maintenance: The controller maintains a few lists which constitute the global state of the system. These lists link objects encapsulating state information for running VMs, running VMMs, and instantiated virtual clusters (VCs). A virtual cluster in Usher can contain an arbitrary set of VMs, and administrators are free to define VCs in any way suitable to their computing environment.

In addition to the above lists, the controller maintains three other lists of VMs:

lost, *missing*, and *unmanaged* VMs. The subtle distinction between lost and missing is that lost VMs are a result of a LNM or VMM failure (the controller is unable to make this distinction), whereas a missing VM is a result of an unexpected VM disappearance as reported by the LNM where the VM was last seen running. A missing VM can be the result of an unexpected VMM error (e.g., We have encountered this case upon a VMM error on migration). Unmanaged VMs are typically a result of an administrator manually starting a VM on a VMM being managed by Usher; Usher is aware of the VM, but is not itself managing it. The list of unmanaged VMs aids resource usage reporting so that Usher has a complete picture of all VMs running on its nodes.

Having the controller maintain system state removes the need for it to query all LNMs in the system for every VM management operation and state query. However, the controller does have to synchronize with the rest of system. I discuss synchronization further in Section 2.3.6.

Event notification: Usher often needs to alert clients and plugin modules when various events in the system occur. Events typically fall into one of three categories:

- VM operation requests
- VM state changes
- Errors and unexpected events

Connected clients automatically receive notices of state changes of their virtual machines. Clients are free to take any action desired upon notification, and can safely ignore them. Plugin modules, however, must explicitly register with the controller to receive event notifications. Plugins can register for any type of event in the system (see Appendix B for a complete list of events for which a plugin can be registered). For example, a plugin may wish to receive notice of VM operation requests for preprocessing, or error and VM state change events for reporting and cleanup.

2.3.5 Clients and the Client API

Applications use the Usher client API to interact with the Usher controller. This API provides methods for requesting or manipulating VMs and performing state queries. We refer to any application importing this API as a client.

The client API provides the mechanism for clients to securely authenticate and connect to the Usher controller. Once connected, an application may call any of the methods provided by the API. All methods are asynchronous, event-based calls to the controller (Section 2.4). As mentioned above, connected clients also receive notifications from the controller for state changes to any of their VMs. Client applications can register to have specific callbacks invoked for these notifications.

2.3.6 Component Interaction

Having described each of the Usher components individually, we now describe how they interact in more detail. We first discuss how the controller and LNMs interact, and then describe how the controller and clients interact. Note that clients never directly communicate with LNMs; in effect, the controller "proxies" all interactions between clients and LNMs.

Controller and LNM Interaction

When an LNM starts or receives a controller recovery notice, it connects to the controller specified in its configuration file. The controller authenticates all connections from LNMs, and encrypts the connection for privacy. Upon connection to the controller, the LNM passes a capability to the controller for access to its VM management API.

Using the capability returned by the LNM, the controller first requests information about the hardware configuration and a list of currently running virtual machines on the new node. The controller adds this information to its lists of running VMs and VMMs in the system. It then uses the capability to assume management of the VMs running on the LNM's node.

The controller also returns a capability back to the LNM. The LNM uses this capability for both event notification and periodic reporting of resource usage back to the controller.

When the controller discovers that a new node already has running VMs (e.g., because the node's LNM failed and restarted), it first determines if it should assume management of any of these newly discovered VMs. The controller makes this determination based solely upon the name of the VM. If the VM name ends with the domain name specified in the controller's configuration file, then the controller assumes it should manage this VM. Any VMs which it should not manage are placed on the *unmanaged* list discussed above. For any VMs which the controller should manage, the controller creates a VM object instance and places this object on its running VMs list. These instances are sent to the LNMs where the VMs are running and cached there. Whenever an LNM sees that a cached VM object is inconsistent with the corresponding VM running there (e.g., the state of the VM changed), it alerts the controller of this event. The controller then updates the cached object on the LNM. In this way, the update serves as an acknowledgment and the LNM knows that the controller received notice of the event.

Similarly, the controller sends VM object instances for newly created VMs to an LNM before the VM is actually started there. Upon successful return from a start command, the controller updates the VMs cached object state on the LNM. Subsequently, the LNM assumes the responsibility for monitoring and reporting any unexpected state changes back to the controller.

One scenario which must be resolved by the Controller and LNM is failed migration. It is difficult to distinguish between a failed and very slow VM migration. In this case, a tunable timeout value is set. Migrations which do not complete withing that time are expected to have failed. At this point, the controller notifies the LNM which returns the VM back to its run state for monitoring purposes.

In the event that a migration assumed to have failed subsequently succeeds, one of two events will occur: i) the destination LNM detects the VM during its periodic scan of its VMM state and reports back to the controller, or ii) the source LNM reports

the VM as missing. In the first case, the controller notifies the source LNM that the VM is no longer there, and the LNM deletes its cached copy of the VM. In both cases, the controller updates its VM object to reflect the VMs new location and sends a cached copy of the VM to the new LNM.

Controller and Client Interaction

Clients to the Usher system communicate with the controller. Before a client can make any requests, it must authenticate with the controller. If authentication succeeds, the controller returns a capability to the client for invoking its remote API methods. Clients use this API to manipulate VMs.

Similar to the local node managers, clients receive cached object instances corresponding to their VMs from the controller upon connection. If desired, clients can filter this list of VMs based upon virtual cluster grouping to limit network traffic. The purpose of the cached objects at the client is twofold. First, they provide a convenient mechanism by which clients can receive notification of events affecting their VMs, since the controller sends updates to each cached VM object when the actual VM is modified. Second, the cached VM objects provide state information to the clients when they request VM operations. With this organization, clients do not have to query the controller about the global state of the system before actually submitting a valid request. For example, a client should not request migration of a non-existent VM, or try to destroy a VM which it does not own. The client library is designed to check for these kinds of conditions before submitting a request. Note that the controller is capable of handling errant requests; this scheme simply offloads request filtering to the client.

The controller is the authority on the global state of the system. When the controller performs an action, it does so based on what the controller believes is the current global state. The cached state at the client reflects the controller's global view. For this reason, even if the controller is in error, its state is typically used by clients for making resource requests. The controller must be capable of recovering from errors due to inconsistencies between its own view of the global state of the system and the actual

global state. These inconsistencies are typically transient (e.g., a late event notification from an LNM), in which case the controller may log an error and return an error message to the client.

2.3.7 Failures

As the Usher system runs, it is possible for the controller or any of the local node managers to become unavailable. This situation could be the result of hardware failure, operating system failure, or the server itself failing. Usher has been designed to handle these failures gracefully.

In the event of a controller failure, the LNMs will start a listening server for a recovery announcement sent by the controller. When the controller restarts, it sends a recovery message to all previously known LNMs. When the LNMs receive this announcement, they reconnect to the controller. As mentioned in Section 2.3.6, when an LNM connects to the controller, it passes information about its physical parameters and locally running VMs. With this information from all connecting LNMs, the controller recreates the global state of the system. With this design, Usher only requires persistent storage of the list of previously known LNMs rather than the entire state of the system to restore system state upon controller crash or failure.

Since the controller does not keep persistent information about which clients were known to be connected before a failure, it cannot notify clients when it restarts. Instead, clients connected to a controller which fails will attempt to reconnect with timeouts following an exponential backoff. Once reconnected, clients flush their list of cached VMs and receive a new list from the controller.

The controller detects local node manager failures upon disconnect or TCP timeout. When this situation occurs, the controller changes the state of all VMs known to be running on the node with the failed LNM to *lost*. It makes no out of band attempts to determine if lost VMs are still running or if VMMs on which LNMs have failed are still running. The controller simply logs an error, and relies upon the Usher administrator or a recovery plugin to investigate the cause of the error.

2.4 Implementation

In this section we describe the implementation of Usher, including the interfaces that each component supports and the plugins and applications currently implemented for use with the system.

The main Usher components are written in Python [pyt]. In addition, Usher makes use of the Twisted[1] network programming framework [twi]. Twisted provides convenient mechanisms for implementing event based servers, asynchronous remote procedure calls, and remote object synchronization. Table 2.1 shows source code line counts of the main Usher components, for total of 4930 lines of code.

Table 2.1 Code size of individual components.

Component	**LoC**
LNM (w/ Xen hooks)	1409
Controller	2076
Client API	786
Utilities	659
Total	4930

2.4.1 Local Node Managers

Local Node Managers export the remote API shown in Table 2.2 to the controller. This API is made available to the controller via a remote object reference passed to the controller when an LNM connects.

This API includes methods to query for VM state information and VM resource usage details using the `get_details` and `get_status` methods, respectively. State information includes run state, memory allocation, IP and MAC addresses, the node on which VM is running, VM owner, etc. Resource usage includes 1, 5, and 15-minute utilizations of the various hardware resources.

The `receive` method creates a cached copy of a VM object on an LNM. An LNM receives the cached copy when it connects to the controller. The LNM periodically

[1] We are using Version 2.5.0 at the time of this writing.

Table 2.2 Local Node Manager remote API.

Method Name	Description
`get_details(vm name)`	get VM state information
`get_status(vm name)`	get VM resource usage statistics
`receive(vm instance)`	receive new cached VM object
`start(vm name)`	start cached VM
`op_on(operation, vm name)`	operate on existing VM
`migrate(vm name, lnm name)`	migrate VM to LNM
`get_node_info()`	get node physical characteristics
`get_current_node_stats()`	get node dynamic and resource usage info
`get_current_vm_stats()`	get resource usage statistics for all VMs

compares the state of the VM object with the actual state of the virtual machine. If the states differ, the LNM notifies the controller which updates the LNM's cached copy of the VM as an acknowledgment that it received the state change notice.

In addition, the cached copy of a VM at its LNM contains methods for manipulating the VM it represents. When a VM manipulation method exposed by the LNM's API is invoked (one of `start`, `op_on`, or `migrate`), the method calls the corresponding method of the cached VM object to perform the operation. This structure provides a convenient way to organize VM operations. To manipulate a VM, a developer simply calls the appropriate method of the cached VM object. Note that the controller must still update the state of its VM object as an acknowledgment that the controller knows the operation was successful.

Most operations on an existing VM are encapsulated in the `op_on` function, and have similar signatures. Table 2.3 shows the list of valid operations to the `op_on` method.

All VM operations invoke a corresponding operation in the VMM's administration API. Though Usher currently only manages Xen VMs, it is designed to be VMM-agnostic. An installation must provide an implementation of Usher's VMM interface to support new virtual machine managers.

The LNM's remote API exposes a few methods that do not operate on VMs. The `get_node_info` method returns hardware characteristics of the phys-

Table 2.3 Operations supported by the `op_on` method.

Operation	Description
`pause`	pause VM execution, keeping memory image resident
`resume`	resume execution of a paused VM
`shutdown`	nicely halt a VM
`reboot`	shutdown and restart VM
`hibernate`	save VM's memory image to persistent storage
`restore`	restore hibernated VM to run state
`destroy`	hard shutdown a VM
`cycle`	destroy and restart a VM

ical machine. The controller calls this method when an LNM connects. The `get_current_node_stats` method is similar to the `get_status` method. Additionally, it reports the number of VMs running on the VMM and the amount of free memory on the node. Finally, the `get_current_vm_stats` method returns resource usage for all VMs running on the physical machine.

2.4.2 Controller

Similar to the local node manager, the controller is an event-based server. Upon startup, the controller listens for connections on two ports, one for client connections and the other for LNM connections.

The remote API exported by the controller to connecting clients closely resembles the interface exported by LNMs to the controller. Table 2.4 lists the methods exported by the controller to Usher clients. This API is made available to clients via a capability passed upon successful authentication with the controller.

Note that most of these methods operate on lists of VMs, rather than single VMs expected by the LNM API methods. Since Usher was designed to manage clusters, the common case is to invoke these methods on lists of VMs rather than on a single VM at a time. This convention saves significant call overhead when dealing with large lists of VMs.

The `start` and `migrate` methods both take a list of LNMs. For `start`,

Table 2.4 Controller remote API for use by Usher clients.

Method Name	**Description**
`list(vm list, status)`	list state and resource usage information for VMs
`list_lnms(lnm list, status)`	list LNMs and resource usage information for VMMs
`start(vm list, lnm list)`	start list of VMs on LNMs
`op_on(operation, vm list)`	operate on existing VMs
`migrate(vm list, lnm list)`	migrate VMs to LNMs
`plugin_method(plugin, method, event, arguments)`	call method exposed by a registered plugin
`set_lnm_acl(lnm list, acl)`	set ACL for a list of LNMs
`register_plugin(plugin, event, configuration dictionary)`	register a plugin for an event
`unregister_plugin(plugin, event, order)`	unregister a plugin for an event

the list specifies the LNMs on which the VMs should be started. An empty list indicates that the VMs can be started anywhere. Recall that this parameter is simply a suggestion to the controller. Policies installed in the controller dictate whether or not the controller will honor the suggestion. Likewise, the LNM list passed to the `migrate` method is simply a suggestion to the controller as to where to migrate the VMs. The controller can choose to ignore this suggestion or ignore the migrate request altogether based upon the policies installed.

The operations supported by the `op_on` method in the controller API are the same as those to the `op_on` method of the remote LNM API (Table 2.3).

As mentioned, plugins provide the ability to extend the operations API exposed by the controller to clients. It is often convenient for plugins to expose a few methods. An SQL plugin may wish to provide a few database management operations, for example. In addition, plugins can be pure API extension modules which register for a pseudo event which never fires (e.g., an LNM restart method). Plugin methods are made available via the controller's `plugin_method` method. Clients specify the plugin and method name, along with the event for which the plugin is registered (a single

plugin can be registered for multiple events) and an argument dictionary to pass to the plugin method. This is an extremely convenient and powerful feature of Usher which allows for the system to provide customized operations for any installation.

Usher also allows for access control at the LNM level. For instance, an administrator may wish to restrict starting and managing VMs on a particular node to only a specified list of users. This level of control is provided through access control lists maintained for each LNM. Connected clients with sufficient privileges can modify these lists through the `set_lnm_acl` method.

Finally, the controller provides methods for clients to register and unregister plugins "on the fly". Providing these methods enables adding functionality or changing policy as the system runs. In this way, administrators can avoid down time due to modifying or changing the active set of plugins. The controller provides the `register_plugin` and `unregister_plugin` methods for this purpose. The `register_plugin` method takes the plugin name to be registered along with the event for which it should be registered. It also takes a configuration dictionary which contains all configuration parameters for the plugin. The `unregister_plugin` takes the name of the plugin to unregister along with the event name and an order parameter to uniquely identify the plugin (since a plugin can be registered for multiple events and even multiple times for the same event).

2.4.3 Client API

The client API closely mirrors that of the controller. An important difference between these two APIs, though, is that the client API signatures contain many additional parameters to aid in working with large sets of VMs. These additional parameters allow users to operate on arbitrary sets of VMs and virtual clusters in a single method call. The API supports specifying VM sets as regular expressions, explicit lists, or ranges (when VM names contain numbers). The client API also allows users to specify source and destination LNMs (i.e., physical machines) using regular expressions or explicit lists.

Another difference between the client and controller APIs is that the client API expands the `op_on` method into methods for each type of operation. Explicitly enumerating the operations as individual methods avoids confusing application writers unfamiliar with the `op_on` method. These methods simply wrap the call to the `op_on` method, which is still available for those wishing to call it directly.

Finally, the client API contains `connect` and `reconnect` methods. These methods contact and authenticate with the controller via SSL. They also start the client's event loop to handle cached object updates and results from asynchronous remote method calls. The `reconnect` method is merely a convenience method to avoid having to pass credentials to the API if a reconnect is required after having been successfully connected. A reconnecting application can use this method upon an unexpected disconnect.

Appendix C.1 contains information on writing Usher clients.

2.4.4 Plugins

Plugins are separate add-on modules which can be registered to receive notification of nearly any event in the system. Plugins live in a special directory (aptly named "plugins") of the Usher source tree. Usher also looks in a configurable location for third-party/user plugins. Any plugins found are automatically sourced and added to a list of *available* plugins.

To register a plugin, an administrator can include a section for it, which includes any configuration parameters for the plugin, in the `plugins` section of its own configuration file. Any plugins listed in the controller's configuration file are loaded at startup. In addition, the controller provides an API call `register_plugin` which can be called from anywhere in the Usher code (e.g., even in other plugins) or by connected clients with sufficient privileges.

Each plugin is required to provide a method named `entry_point` to be called when an event fires for which it is registered. It is possible to add a single plugin to multiple event handler chains, as well as, multiple times to a single event handler

chain.

By default, plugins for each event are called in the order in which they are registered. Therefore, careful consideration must be given to ordering while registering plugins. A plugin's configuration object can optionally take an *order* parameter that governs the order in which plugins are called on the event's callback list. The plugin API also provides a converse `unregister_plugin` call to change event handling at runtime.

When an event occurs in the system, the controller calls the `entry_point` method of each registered plugin in turn. The `entry_point` method receives the event type, as well as the VM or list of VMs involved in the event (for VM related events which are the most common). VM manipulation request events typically involve lists of VMs, whereas VM operation completion events typically involve a single VM since operations are carried out with an asynchronous remote method call for each VM involved.

Plugins can be as simple or complex as necessary. Since the controller invokes plugin callback chains asynchronously (in their own thread), complex plugins should not interfere with the responsiveness of the Usher system (i.e., the main controller event loop will not block waiting for a plugin to finish its task).

Sites can install or customize plugins as desired. The Usher system supports taking arbitrary input and passing it to plugins on request events.[2] For example, a request to start a VM may include information about which OS to boot, which filesystem to mount, etc. Plugin authors can use this mechanism to completely customize VMs at startup.

Appendix C.2 contains information on writing Usher plugins.

Events

There are currently forty events for which plugins can be registered in the core Usher system. These are broken down into five different categories: client, cluster, controller, LNM, and VM events. Appendix B contains a complete list of all events in

[2] A request event is an event generated as a result of receiving a client request to operate on a VM or set of VMs.

the core Usher system at the time of this writing.

In addition, plugins can create additional events for which other plugins may register. For example, a monitoring plugin may create an event named `monitor_sample`. A scheduling plugin may then register for this event to be notified when new data is available for re-scheduling.

Policy Enforcement

Policies in an Usher installation are implemented as plugins. As an example, an administrator may have strict policies regarding startup and migration of virtual machines. To enforce these policies, a plugin (or plugins) is written to authorize start and migrate requests. This plugin gets registered for the `start_request` and `migrate_request` events, either manually using the controllers `register_plugin` command, or by specifying these registrations in the controller's configuration file. Once registered, subsequent start and migrate requests are passed to the plugin (in the form of a Request object) for authorization. At this point, the plugin can approve, approve with modification, or simply reject the request. Once this is done, the request is passed on to any other plugins registered on the `start_request` or `migrate_request` event lists with a higher order attribute.

Figure 2.3 depicts a migrate request traversing the plugin callback chain for the migrate event. Each plugin receives the encapsulated migrate request object, performs its operation, then passes the request object to the `entry_point` method of the subsequent plugin in the chain. Notice that each plugin is free to modify the request object as it sees fit (as shown for Plugin 2). In this way, policy plugins can enforce any rule(s) an administrator wishes to define — including denying the request altogether by converting it into an empty request.

Besides authorization policies, one can imagine policies for VM operation and placement. For example, initial VM placement, VM scheduling (i.e., dynamic migration based on load or optimizing a utility function), or reservations. A policy plugin for initial placement would be registered for the `start_request` event (probably with

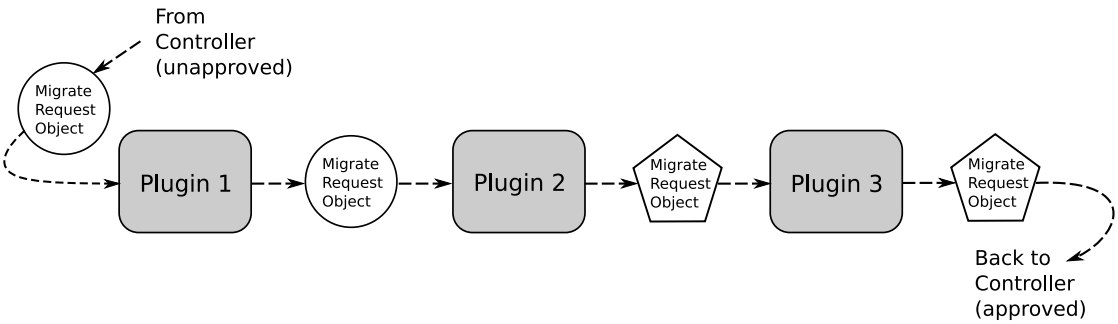

Figure 2.3 Migrate request traversing plugin chain.

a higher order attribute than the startup authorization policy discussed above so that it is called later in the plugin callback chain). Some simple policies such a plugin might support are round-robin and least-loaded. Scheduling and reservation plugins could be registered with a timer to be fired periodically to evaluate the state of the system and make decisions about where VMs should be migrated and which VMs might have an expired reservation, respectively.

2.4.5 Configuration Files

Each component of the Usher system has its own configuration file. The Usher configuration module handles parsing of these configuration files. This module extends ConfigObj [cona] to support additional data types and allow for variables to be defined which apply to all subsections of a nested level. All valid configuration parameters, their type, and default values are specified in the code for each component and documented in an included sample configuration file. Likewise, plugins and clients typically include a sample configuration files with all valid options documented there.

The configuration system tries to read in values from the following locations (in order): a default location in the host filesystem, a default location in the user's home directory, and finally a file indicated by an environment variable. This search ordering enables users to override default values easily. Values read in later configuration files replace values specified in a previously read file.

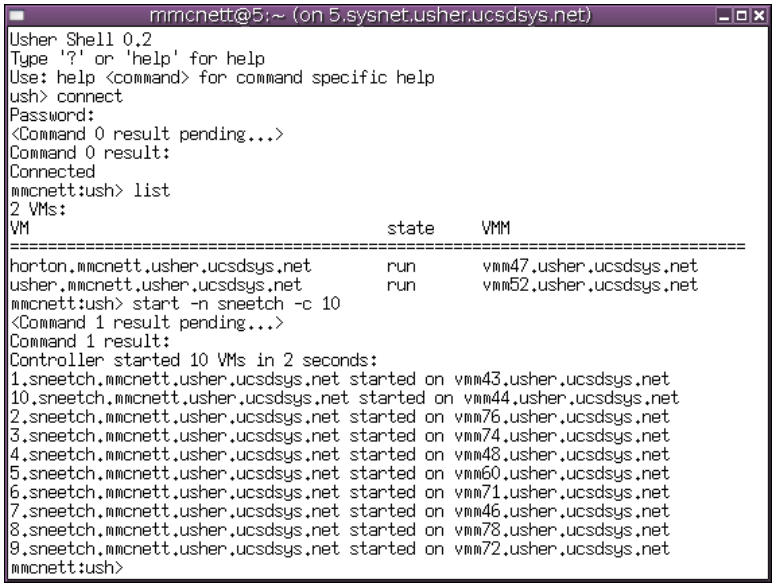

Figure 2.4 Ush Shell

2.5 Application Examples

This section gives a sense for the broad range of applications which can take advantage of Usher's client API to manage virtual machines. Here, we discuss three such applications: a shell named *ush*, an XML-RPC server named *plusher*, and a simple control script for driving VM scheduling experiments.

It is easy to imagine other potential client applications. A current wish list of future clients includes: a web interface for managing VMs via a web browser, a Firefox [fir] extension for managing VMs, a command line suite providing executables callable from a UNIX shell (bash, tcsh, etc.), and a flashy graphical user interface.

2.5.1 Ush Client

The Usher shell *ush* provides an interactive command-line interface to the API exported by the Usher controller. In this regard, the commands available in *ush* mirror the client API. *Ush* provides persistent command line history and comes with extensive online help for each command. If provided, *ush* can read connection details and other startup parameters from a configuration file. *Ush* is currently the most mature

and preferred interface for interacting with the Usher system. *Ush* is implemented in Python and currently consists of 1180 lines of code (over 400 of which is simply online documentation).

Since Usher is extremely extensible, *ush* was written to be extensible as well. First, *ush* gives users the ability to create or replace command options through its configuration file. This ability is convenient since plugins may be in use which extend options for some commands. For example, a DNS plugin may take input to specify whether it should add round robin or reverse DNS entries for a VM. This input could be passed via extra options to the start command.[3] In addition, it is sometimes convenient to replace existing options. For example, an IP address handling plugin may replace the "`ip_addrs`" option to start with its own option. Such command option modifications can easily be made to a site with a few lines in the *ush* site configuration file.

In addition to changing command options, *ush* was written to allow new commands to be easily added. Adding the new command to the dictionary of commands along with a method to invoke is all that is required. If desired, a list of command options may also be specified.

Adding commands is sometimes used for command aliasing. Methods exposed by plugins registered with the controller can be called via the "`plugin_method`" command in *ush*. Using the `plugin_method` command is rather cumbersome in practice, so adding additional commands to *ush* for plugin commands is often desirable.

We now describe a sample *ush* session from the UCSD Usher installation, along with a step-by-step description of actions taken by the core components to perform each request. In this example, user "mmcnett" requests ten VMs. Figure 2.4 contains a snapshot of *ush* upon completion of the start command.

First a user connects to the Usher controller by running the "connect" command. In connecting, the controller receives the user's credentials and checks them against the LDAP database. Once authentication succeeds, the controller returns a capa-

[3] At UCSD, we added an "`--rr`" option to the *ush* start command to indicate that round robin DNS entries should be added for these new VMs.

bility for its remote API and all of user mmcnett's VMs. The somewhat unusual output "`<Command 0 result pending...>`" reflects the fact that all client calls to the controller are asynchronous. When "connect" returns, *ush* responds with the "`Command 0 result:`" message followed by the actual result "`Connected`".

Upon connecting *ush* saves the capability and cached VM instances sent by the controller. Once connected, the user runs the "list" command to view his currently running VMs. Since the client already has cached instances of user mmcnett's VMs, the list command does not invoke any remote procedures. Consequently, *ush* responds immediately indicating that user mmcnett already has two VMs running.

The user then requests the start of ten VMs in the "sneetch" cluster. In this case, the `-n` argument specifies the name of a cluster, and the `-c` argument specifies how many VMs to start in this cluster. When the controller receives this request, it first calls on the authorization and IP management modules to authorize the request and reserve IP addresses for the VMs to be started. Next, the controller calls the initial placement plugin to map where the authorized VMs should be started. The controller calls the `start` method of the remote LNM API at each new VM's LNM. The LNMs call the corresponding method of the VMM administration API to start each VM. Upon successful return of all of these remote method calls, the controller responds to the client (*ush*) that the ten VMs were started in two seconds and provides information about where each VM was started. After completing their boot sequence, user mmcnett can ssh into any of his new VMs by name.

2.5.2 Plusher

Plush [ATSV06] is an extensible execution management system for large-scale distributed systems, and *plusher* is an XML-RPC server that integrates Plush with Usher. Plush users describe batch experiments or computations in a domain-specific language. Plush uses this input to map resource requirements to physical resources, bind a set of matching physical resources to the experiment, set up the execution environment, and finally execute, monitor and control the experiment.

Since Usher is essentially a service provider for the virtual machine "resource", it was natural to integrate it with Plush. This integration would allow users to request virtual machines (instead of physical machines) for running their experiments using a familiar interface.

Developing *plusher*[4] was straightforward. Plush already exports a simple control interface through XML-RPC to integrate with resource providers. Plush requires providers to implement a small number of up-calls and down-calls. Up-calls allow resource providers to notify Plush of asynchronous events. For example, using down-calls Plush requests resources asynchronously so that it does not have to wait for resource allocation to complete before continuing. When the provider finishes allocating resources, it notifies Plush using an up-call.

To integrate Plush and Usher in *plusher*, it was only necessary to implement stubs for this XML-RPC interface in Usher. The XML-RPC stub uses the Client API to talk to the Usher controller. The XML-RPC stub acts as a proxy for authentication — it relays the authentication information (provided by users to Plush) to the controller before proceeding. When the requested virtual machines have been created, *plusher* returns a list of IP addresses to Plush. If the request fails, it returns an appropriate error message.

2.5.3 Control Scripts

As an example of a very simple client, we wrote VM control scripts as an Usher client to automate runs of the scheduling experiments of Chapter 4. These scripts proved invaluable to running scheduling experiments which are often long in duration.

The control scripts use the client API to initialize each experiment. Initializing each experiment involves starting VMs with the proper resources, initial placement, and load. Experiments run for each scheduler of interest over a range of parameters (e.g., number of VMs and physical machines, loads, etc.). Upon completion of all experiments, the control scripts save the experimental results and clean up by shutting down

[4]The *plusher* application was written by Diwaker Gupta. Thanks to Diwaker for being the first person (other than myself) to attempt to write an Usher application.

all running VMs.

Though this is a fairly straightforward client, the ability to script VM management operations is extremely powerful. Scripting is a major component of large infrastructure management today. As a result, administrators can conveniently control Usher managed VMs with familiar tools.

2.6 Plugin Examples

I now describe a few existing Usher plugins to demonstrate the adaptability of the Usher system to support a wide range of computing environments. The plugins presented here will be limited to those written by the author for use by the SysNet group at UCSD. Nonetheless, these show the extensibility of Usher, which can be leveraged by any administrator to incorporate Usher into her installation.

The SysNet installation uses the following plugins: an SQL database plugin for IP address management supporting access control on named address ranges and multiple VLANS (`uipmgr`); an LDAP plugin for user authentication for both Usher and VMs created by Usher (`uldap`); a DNS plugin for modifying DNS entries for VMs managed by Usher (`udns`); a monitoring plugin to track VM resource utilization (`umon`); and a scheduling plugin to determine where VMs should run (`usched`). We will defer detailed discussion of the scheduling plugin to Chapter 3.

All plugins for the UCSD installation are written in Python. Table 2.5 contains line counts for these plugins.

Table 2.5 Code size of UCSD plugins.

Plugin	**LoC**
IP Manager	250
LDAP	869
DNS	140
Monitor	85
Scheduler	1,786

2.6.1 IP Address Management

The IP address management plugin uses a PostgreSQL [pos] database to manage and provide access control on named IP address ranges. A useful feature of PostgreSQL is its support for an IPv4 data type via the Indexable IPv4 range [ipv] extension. This extension is very convenient for defining address ranges (arbitrary or CIDR), testing inclusion of addresses in ranges, etc.

When an IP address on a particular range is requested, the plugin finds the next available address on that range and returns it along with other network specific parameters for that address (e.g., netmask, vlan, etc). If an address is not available, or the range does not exist, the request fails.

The SysNet installation does not use DHCP to assign IP addresses to VMs for two reasons. First, the SysNet group manages several subnets,[5] spanning multiple VLANs. Since setting up interfaces on different VLANs is handled by scripts on each physical machine, a VLAN number must be passed to the LNM when starting a VM. Passing of VLAN numbers cannot be done with DHCP.

The second reason for not using DHCP is that it is impossible to assign ownership of arbitrary IP address ranges to specified users. When starting a VM, an address range can be specified by name.[6] Permissions on the specified range are checked so that only authorized users can assign addresses from that range.

Events and Operation

The IP manager plugin registers for the following events: `ctrl_start`, `start_request`, `start`, `start_failure`, `register`, `unregister`, and `state_change`.

Upon receiving a `ctrl_start` event, this plugin cleans up the list of reserved IP addresses and sets the state of all used IP addresses to “unknown”. It also removes addresses which have been in the “lost”, “missing”, or “unknown” state[7] for longer

[5] Some are externally routable, others routable only withing UCSD.
[6] Otherwise, addresses are taken from a default range with open access.
[7] The “lost” and “missing” states correspond to VM states described in Section 2.4.2.

than a tunable expiration duration. Since the controller queries the global state of the system at startup, it regenerates the correct list of used IP addresses at startup (see the discussion of the `register` event below). In this way, we avoid inconsistencies and lost IP addresses.

The `start_request` event will cause the IP manager plugin to reserve IP addresses for the set of VMs being started. Reserving addresses is necessary since a start may fail for some subset of the VMs. In this case, the plugin reclaims any unused reserved address upon receiving a `start_failure` event.

In the event of a successful start, the firing of the `start` event will cause the plugin to move the IP address for the VM (only one VM per `start` event) for the reserved to the used table.

The IP manager plugin sets the state of the IP address to that of the VM (e.g., run) upon receipt of a `register` event object. Setting the address state is necessary since, as mentioned, the `ctrl_start` event sets the state of all used IP addresses to "unknown". Therefore, each VM found during the state regeneration period of controller restart will have its IP address state in the database updated (or added if not there). The `unregister` event causes the plugin to reclaim the IP address used by the VM.

Finally, the IP manager plugin updates the state of an IP address to be that of the VM using that address upon receipt of a `state_change` event. This can happen when a VM goes missing, is lost, or crashes.[8]

2.6.2 LDAP

The LDAP plugin serves two purposes. First, it provides methods for managing and authenticating Usher users. Second, it provides the convenience of creating a branch in the LDAP database for each cluster an Usher user creates. This branch enables each VM the user creates to authenticate its users through the LDAP database.

This functionality provides a convenient authentication service to virtual cluster creators. First, it allows Usher users to use their Usher credentials as their VM login

[8] A crash actually causes the IP address to be reclaimed since the VM will need to be restarted and can get a new address at that time.

credentials since they are automatically added as a user in each cluster created. Since each cluster uses a different branch in the LDAP database, we use aliasing in LDAP to provide Usher users a single set of credentials. In addition, the plugin adds each Usher user to the "admin" group of each cluster the user creates. VM filesystems can then be configured to grant special privileges to this group (e.g., sudo privileges). This approach is convenient when using a read-only NFS root filesystem where no default root password is set.

Second, and more importantly, this arrangement addresses the cluster authentication problem for Usher users in the SysNet group. Authentication for clusters is challenging enough for experienced administrators. Delegating this problem to users is not only time consuming for them, but could lead to insecure VMs.

Creating a separate branch for each cluster allows Usher users to create accounts and groups for their clusters without burdening the Usher administrator with this task. This capability is especially conducive to collaborative work, a common case in a research lab setting. An administrator could easily be overwhelmed with management requests in a setting where users are free to create their own clusters, yet are unable to fully manage them. This approach pushes many mundane administrative tasks out to the users who have the incentive to create accounts on their VMs.

Allowing Usher users to modify the LDAP database requires careful configuration of the LDAP server, however. An LDAP server configuration file that allows Usher users to only manage branches which they own is included with the Usher source code. In addition, the Usher plugin for the LDAP server includes scripts for installation on a user's VM filesystems to modify cluster LDAP entries (i.e., to add, modify, or delete users and groups).

Events and Operation

The LDAP plugin registers for the following events: `cluster_register`, `client_authenticate`, and `lnm_authenticate`. The LDAP plugin creates a new branch for a cluster in its LDAP database upon receiving a `cluster_register`

event object. As mentioned, this branch authenticates users of the new cluster. The cluster creator also has permissions to manage users and groups in this new LDAP branch.

The Usher credential checker module generates `client_authenticate` and `lnm_authenticate` events when a client or LNM attempt to authenticate. The LDAP module then returns whether or not the credentials passed in the event object are valid.

2.6.3 DNS

By default, Usher names VMs using the following naming scheme:

`<requested VM name>.<creator's username>.<Usher system domain name>`

where the Usher system domain name is specified in a configuration file read by the controller at startup. The DNS plugin adds both A and PTR records for each VM into a DNS server specified in the plugin's configuration file. Optionally, a round robin DNS entry can be added for each VM if the keyword parameter "rr" is set in the start request object passed to the plugin.

This plugin registers for the `register` and `unregister` events. The `register` event prompts the DNS plugin to add records (A, PTR, and/or round robin) to its DNS server, whereas the `unregister` event prompts removal of the entries.

2.6.4 Monitor

The monitoring plugin is responsible for consolidating resource usage data for all VMs and PMs into a format convenient for consumption by plugins and clients. This information is maintained by each LNM and up-to-date information is returned to the monitoring plugin upon request. Other plugin modules may use this data, for example, to restrict user resource requests based on the current system load or to make VM scheduling decisions to determine where VMs should run. Clients uses may include triggering arbitrary action when loads reach a threshold or simply reporting the

information back to interested users.

The Monitor plugin registers for a "periodic" event. The controller fires this event at a specified interval. Unlike other events, periodic events are not appended to event callback lists. Rather, they register with Twisted which periodically calls them from its main event loop thread.[9]

In addition, the monitor plugin provides a new event, aptly named "monitor_sample". This new event enables plugins register to be notified when the monitoring plugin has collected and consolidated a new data sample.

Each time the monitor plugin is fired, it collects information about resource usage of each VM and PM as reported by their LNMs. Currently, the LNM is responsible for providing instantaneous load data, as well as, data averaged over 1, 5, and 15 minute intervals for VM CPU cycle consumption, network transmit and receive bandwidth, network transmit and receive packet counts, and disk read and write request counts.

2.6.5 Start Request Scenario

As a concrete example of plugin operation, we outline the sequence of events for a scenario of starting a set of VMs.

When a request to start a list of VMs arrives, the controller calls the modules registered for the "start request" event. The IP address module is in the callback list for this event. This module receives the request object and reserves IP addresses for each of the new VMs.

The controller generates a separate VM start command for each VM in the start list. Prior to invoking the start command, the controller triggers a "register VM" event for each VM. The IP management and DNS plugin modules are registered for this event. The IP management module adds changes the state of the address from "reserved" to "init" to reflect the fact that this is now a VM included in the controller's view of the global state. The DNS plugin simply sends a DDNS update to add A and PTR records for this VM in our DNS server.

[9] Similarly, one-off "timer" events are registered with Twisted and called by its event loop as well.

Finally, upon return from each start command, a “start” event fires. The IP manager plugin is registered to receive this event. This plugin checks the result of the command, then either marks the corresponding IP address as in use by changing its state from “reserved” to “init” (upon success) or releases it (upon failure). Eventually, the IP module changes the state of the address to match that of the VM using it when it receives a “state changed” event for this VM (from “init” to “run” if all goes well).

2.7 Usher Installations

This section presents details of two early deployments of Usher in a production environment. The first deployment is for the UCSD CSE Systems and Networking research group, and the second deployment is at the Russian Research Center, Kurchatov Institute (RRC-KI). The two sites have very different usage models and computing environments. In describing these deployments, our goal is to illustrate the flexibility of Usher to meet different virtual machine management requirements and to concretely demonstrate how sites can extend Usher to achieve complex management goals.

Usher does not force one to setup or manage their infrastructure as done by either of these two installations. These installations demonstrate the flexibility Usher offers in setting up a virtual cluster computing environment.

2.7.1 UCSD SysNet

The UCSD CSE Systems and Networking (SysNet) research group has been using Usher experimentally since June 2006 and for production since January 2007. The group consists of nine faculty, 60 graduate students, and a handful of research staff and undergraduate student researchers. The group has a strong focus on experimental networking and distributed systems research, and most projects require large numbers of machines in their research. As a result, the demand for machines far exceeds the supply of physical machines, and juggling physical machine allocations never satisfies all parties. However, for most of their lifetimes, virtual machines can satisfy the needs

of nearly all projects: resource utilization is bursty with very low averages (1 percent or less), an ideal situation for multiplexing; virtualization overhead is an acceptable trade-off to the benefits Usher provides; and users have complete control over their clusters of virtual machines, and can fully customize their machine environments. Usher can also isolate machines, or even remove them from virtualization use, for particular circumstances (e.g., obtaining final experimental results for a paper deadline) and simply place them back under Usher management when the deadline passes.

At the time of this writing, the SysNet group has staged up to 62 physical machines from their hardware cluster into Usher. On those machines, Usher has multiplexed over 400 virtual machines in dozens of virtual clusters. The SysNet Usher controller runs on a Dell PowerEdge 1750 with a 2.8 GHz processor and 2 GB of physical memory. This system easily handles their workload. Although load is mostly dictated by plugin complexity, using the plugins discussed in Section 2.6, the Usher controller consumes less than 2 percent CPU on average (managing over 300 virtual machines) with a memory footprint of approximately 20MB. The Usher implementation is sufficiently reliable that SysNet is now migrating the remainder of their user base from dedicated physical machines to virtual clusters, and Usher will soon manage all physical nodes ($>$200) in the SysNet cluster.

Usage

The straightforward ability to both easily create arbitrary numbers of virtual machines as well as destroy them has proved to be very useful, and the SysNet group has used this capability in a variety of ways. As expected, this ability has greatly eased demand for physical machines within the research group. Projects simply create VMs as necessary. Usher has also been used to create clusters of virtual machines for students in distributed systems and networking courses; each student or student group can create a cluster on demand to experiment with distributed protocol implementation, etc.

Often, projects require a machine or set of machines visible to the world on which services are run. This situation is easily handled with Usher. The SysNet group

uses the IP manager plugin (see Section 2.6.1) to provide externally routable addresses to such projects. These projects then specify at VM startup to use an address on the range containing their externally routable addresses.

The group also previously reserved a set of physical machines for general login access (as opposed to reserved use by a specific research project). With Usher, a virtual cluster of convenience VMs now serves this purpose, and an alias with round-robin DNS provides a logical machine name for reference while distributing users among the VMs upon login. Even mundane tasks, such as experimenting with software installations or configurations, can benefit as well because the cost of creating a new machine is negligible. Rather than having to undo mistakes, a user can simply destroy a VM with an aborted configuration and start from scratch with a new one.

The SysNet group currently uses a simple policy module in Usher to determine the scheduling and placement of VMs. This module relies upon monitoring data collected by a monitoring plugin to make its decisions. It uses heuristics to place new VMs on lightly loaded physical machines, and to migrate VMs when a particular VM imposes sustained high load on a physical machine. Users are reasonably self-policing; they could always create large numbers of VMs to fully consume system resources, for example, but in practice do not. Eventually, as the utilization of physical machines increases to the point where VMs substantially interfere with each other, the group will interpret it as a signal that it is time to purchase additional hardware for the cluster.

This policy works well for the group, but of course is not necessarily suitable for all situations, such as the RRC-KI deployment described in Section 2.7.2.

Usher Filesystems at UCSD

The UCSD SysNet group's installation of Usher uses read-only NFS root filesystems for both VMs and VMMs with a separate writable NFS server for persistent VM filesystem customizations. There are a few reasons for this setup. First, live migration of virtual machines requires a filesystem accessible by the VM at both the source and destination VMM. Since migration is a requirement of the SysNet installation, Sys-

Net VMs must have their root filesystems provided via network-attached storage.

In addition, serving the root filesystem read-only has multiple benefits. First, it is straightforward to keep filesystems across all running VMs synchronized and updated using read-only NFS root filesystems. Furthermore, an experienced administrator can manage this filesystem to ensure that it is secure (e.g., default firewall rules, minimal services started by default, latest security patches, etc.).

Since all VMs mount this filesystem, it is important that it be as responsive as possible. Ensuring that the NFS server serving this filesystem is read-only helps improve performance. Furthermore, an administrator can configure a read-only NFS server to cache the entire filesystem in main memory. As a result, reads go to disk infrequently.

One issue with using a read-only root filesystem is that some files and directories on the filesystem must be writable at system startup. We solve this problem using a ramdisk for any files and directories which must be writable. Early in the boot process, these files and directories are copied into the ramdisk, then mounted using the `--bind` flag to make them writable.

Since the SysNet installation serves its root filesystems read-only, another NFS server provides persistent writable storage. The Usher VM filesystems are configured to initialize their NFS mounted filesystems at boot time. The VMs create the following directories on the group's read-write NFS server:

- **/net/global:** This directory is where users install or store anything they would like to have globally accessible by all of their clusters. The contents of `/net/global` is the same for all VMs a user creates.

- **/net/cluster:** This directory is where users can store files they want accessible by the current cluster only. The contents of `/net/cluster` is the same for all VMs in the same cluster.

- **/net/local:** This directory is unique to the current VM only. The contents of `/net/local` is different for every VM a user creates. Users can use this direc-

tory to set up services and configuration files specific to particular VMs.

Finally, all SysNet users are given a home directory. Automount takes care of mounting these directories upon login. Alternatively, Usher users can choose an alternate URI (stored in LDAP) for their home directory.

In each of `/net/global`, `/net/cluster`, and `/net/local`, there exists a System V init style directory structure in the `etc` directory. Startup scripts in the VM filesystems have been modified to run scripts in the directory for the appropriate runlevel from these three locations after the regular system startup scripts run. With this configuration, even though users cannot write to the root filesystem to change startup scripts, they can have services started for their VMs at VM boot.

Finally, as mentioned above, the VMM filesystems are also served read-only NFS root. This has the same advantages as those listed above, with one additional advantage. It eliminates a common excuse for not placing checked out machines into the Usher system during periods when groups are not using them. The excuse referred to is that users do not want to loose the extensive modifications they have made to the filesystem on the local disk. Since our VMMs do not run from local disk (the disk is never even mounted), these users can get their machines back in the exact state they were in before being put into the Usher system.

2.7.2 RRC-KI

Usher has also been deployed at the Russian Research Center, Kurchatov Institute (RRC-KI). The RRC-KI deployment demonstrates the flexibility of Usher to integrate with different computing environments, and to employ different resource utilization policies. Whereas the UCSD SysNet Usher deployment targeted a general-purpose computing environment, the RRC-KI Usher deployment targets a batch job execution system that provides guaranteed resources to jobs.

RRC-KI contributes part of its compute infrastructure to the Large Hadron Collider (LHC) Grid effort [lcg]. Scientists submit jobs to the system, which are scheduled via a batch job scheduler. Jobs are assigned to physical machines, and one machine

only runs a single job at any time.

Measurements spanning over a year indicated that the overall utilization of machines in this system is fairly low [CGK+06]. While there were some long, compute intensive jobs, there was a large fraction of short, I/O driven jobs. Motivated by these measurements, the goal was to build a flexible job execution system that would improve the aggregate resource utilization of the cluster.

A straightforward approach is to multiplex several jobs on a single machine, and power down the unused machines. However, conventional process-based multiplexing on commodity operating systems is infeasible for a variety of reasons, some social and some technical: scientists want at least the appearance of absolute resource guarantees for their jobs; jobs often span multiple processes, which makes resource accounting and allocation challenging; and the number of physical machines needed depends on the workload and cannot be assigned *a priori*.

Virtual machines are a natural solution to this problem. Since each job gets its own isolated execution environment, resource accounting becomes easier for multi-process jobs. VMs also provide much stronger isolation guarantees than conventional processes. Each job can be given guaranteed resource reservations while still maintaining the abstraction of a physical machine. A trace-driven simulation showed that a VM-based infrastructure would enable significant savings [CGK+06].

One of the biggest challenges to this approach is management. For a VM-based infrastructure to scale, the RRC-KI team need an automated system for deploying and managing virtual machines, a system that can schedule VMs in an intelligent manner, and migrate and place VMs to optimize utilization without sacrificing performance. A prototype system is currently being used at RRC-KI with Usher as the core management framework.

Central to this infrastructure is the *Policy Daemon* responsible for job scheduling and dynamically managing virtual machines (creation, migration, destruction) as a function of the current workload. The Policy Daemon uses the Usher client API to monitor VM status and control VM resource utilization from a single control point us-

ing secure connections to the physical hosts. The current testbed comprises of a small number of nodes hosting production Grid jobs in the Usher-based environment with plans to expand the system to manage a few hundred nodes [Kur].

2.8 Conclusions

Usher is an extensible, event-driven management system for clusters of virtual machines. The Usher core implements basic virtual machine and cluster management mechanisms, such as creating, destroying, and migrating VMs. Usher clients are applications that serve as user interfaces to the system, such as the interactive command-line shell *Ush*, as well as applications that use Usher as a foundation for creating and manipulating virtual machines for their own purposes. Usher supports customizable plugin modules for flexibly integrating Usher into other administrative services at a site, and for installing policies for the use, placement, and scheduling of virtual machines according to the site-specific requirements. Usher has been in production use both at UCSD and at the Russian Research Center in Kurchatov, Russia, and initial feedback from both users and administrators indicates that Usher is successfully achieving its goals.

2.8.1 Usher Availability

Usher is free software distributed under the new BSD license. Source code, documentation, and tutorials are available at:

`http://usher.ucsd.edu`

Source code, configuration files, and initialization scripts for the UCSD plugins are also available for download at the site above.

2.9 Acknowledgement

This chapter, in part, is a reprint of material as it appears in the USENIX LISA Conference, 2007, McNett, Marvin; Gupta, Diwaker; Vahdat, Amin; Voelker, Geoffrey M.. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Virtual Machine Scheduling in a Virtual Cluster Environment

In this chapter we introduce the virtual machine scheduling problem. We define our problem, then discuss work closely related to VM scheduling. We propose reasonable cluster scheduling goals and present our canonical cluster scheduling problem called Fair Maximum Utilization (FMU). After proving that that FMU is in the class of NP-hard problems, we finish with detailed discussion of several heuristic approaches for FMU scheduling. Our solvers generalize to solving any scheduling problem that fits into the framework we define.

3.1 Introduction

Having solved the problem of managing large numbers of virtual machines across a site (Chapter 2), we now look to the problem of mapping virtual machines onto available computing resources for optimum utility. Figure 3.1 depicts this problem. Given a set of VMs, each with its own resource demands, our problem is to place these VMs onto a (possibly heterogeneous) set of physical machines such that an arbitrarily defined utility is optimized. In this chapter, we refer to an optimal assignment of VMs to physical machines as a "solution". We refer to an arbitrary assignment of VMs to

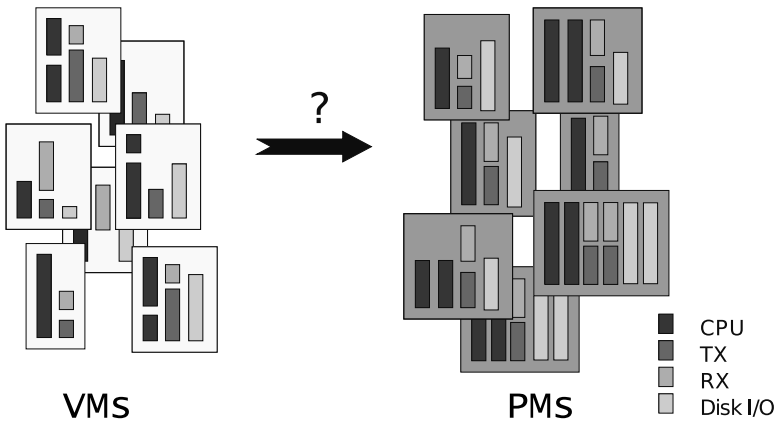

Figure 3.1 Scheduling virtual machines (VMs) onto a set of physical machines (PMs).

physical machines as either an “assignment” or a “schedule”.

VM scheduling can be partitioned into two categories: contractual and best-effort. A contractual scheduler seeks to place VMs onto PMs so as to provide guaranteed quantities of resources. This type of scheduling is required by providers offering service-level agreements (SLA) to their clients (e.g., datacenters). In this setting, a scheduler can either find schedules which satisfy all contracts, or not. In cases where all contracts cannot be satisfied, utility functions are often employed to determine which contracts to honor to maximize profit. Another possibility in contractual scheduling is for service providers to create SLAs after they have determined how to divide resources among VMs so as to maximize profit.[1] In this case, users do not specify exact resource needs, but rather how valuable a unit of resource is to them. In either case, contractual scheduling uses utility functions to determine what quantities, if any, of a particular resource VMs will receive.

On the other hand, best-effort scheduling seeks to place VMs onto PMs so as to maximize arbitrarily defined utility without providing guarantees about how much of a particular resource a VM will receive, or exclude VMs from running. This type of scheduling uses utility functions to determine where VMs should run rather than what quantities of resources they will receive. In best-effort scheduling, weights and admis-

[1] This is essentially a multi-dimensional multiple-knapsack (MDMK) problem.

sion control can be employed to provide pseudo guarantees about resource allocations if desired. This flexibility may make best-effort scheduling more suitable for general purpose cluster computing where multiple operational goals might exist. In this dissertation we focus on best-effort VM scheduling which is ideal for shared research clusters.

Another consideration in the VM scheduling problem is how VM resource demands are made known. Establishing the resource demands of a VM involves either reporting or monitoring. Reporting originates within the VM whereas monitoring is external. Both approaches have advantages. With reporting, VMs know their applications and can send notices to indicate whether they are receiving sufficient resources. This reporting is typically handled by the application(s) itself, or by a separate monitoring application running within the VM. With reporting no predictions need be made about how VMs will behave in the future. When a VM's resource demands change, it will notify the system. This type of "grey box" auditing lends itself to scheduling for enforcement of SLAs.

On the other hand, monitoring is completely black box. Administrators do not need to know anything about the operating systems or applications running on the VMs in their cluster, and users do not need to implement their own reporting mechanisms. Since VMs do not report when their resource demands change, monitoring requires that predictions be made about how a VM will behave in the future.

In this dissertation, virtual machines are treated as black boxes; we know nothing about the applications running on our VMs and make no a priori assumptions about what resources a VM might desire. Hence, we use monitoring to determine our VMs' resource demands as they run. Our schedulers base predictions about a VM's resource desires in the near future upon how it has behaved in the recent past. Further, since VM resource desires can vary with time, a solution at one moment is not necessarily a solution at another, making our problem one of "scheduling" rather than "placement". As a result of varying resource desires, our schedulers must continuously monitor and estimate VM resource demands to find new solutions as the system runs.

Another consideration is whether a distributed or centralized approach to VM

scheduling should be adopted. Though distributed approaches provide scalability (assuming information dissemination overhead can be controlled) and reliability (no single point of failure), we only consider centralized approaches. With a centralized scheduler all VM resource demands are collected by a single scheduler (of course, a failover controller could be used for redundancy) and all migration decisions are made there. This greatly simplifies our problem by eliminating the need for complex dissemination and agreement protocols required by distributed scheduling approaches. In addition, enforcing complex global operational goals can be difficult in a distributed approach. Our experience is that a centralized server can scale to large clusters. Managing over 400 virtual machines induced less than three percent sustained load on our single processor 2.8GHz Xeon controller node.

The schedulers we explore periodically monitor resource usage and evaluate whether assignments exist which improve upon the utility of the current assignment. This is the steady state operation of our schedulers. When an improving assignment is found, the utility improvement is measured against the cost of migrating VMs to the new assignment. If migration cost is acceptable, the scheduler initiates the migrations required to transition the current VM mapping to the new schedule.

As mentioned, moving from one assignment or solution to another has a cost. Migration of VMs is expensive in terms of both network bandwidth and CPU usage. So, any approach to solving the VM scheduling problem should account for assignment transition cost, as well as assignment utility. We discuss methods for reducing migrations below, though this is an area warranting further study.

Given the scheduling problem as formulated above, we implemented an Usher scheduling plugin which utilizes data collected by the monitoring plugin (see Section 2.6.4) to make its scheduling decisions. The scheduling plugin registers for the "monitor_sample" event provided by the monitoring plugin to be alerted when updated resource usage data is available. Scheduling frequency is an integral multiple of the monitoring frequency. Once a new monitor sample is available, the scheduling plugin employs a specified heuristic for finding a utility improving assignment. We present the

scheduling heuristics we evaluated below. First, however, we elaborate on the meaning of utility and discuss some reasonable operational goals for a virtual cluster environment.

For our VM scheduling problem, we represent utility as a mathematical function to be optimized. Utility functions are designed to reflect how well an assignment meets a specified set of operational goals for a computing environment. The domain of a utility function is the set of all *feasible* assignments of VMs to PMs, where feasible solutions are those which satisfy all problem constraints, such as physical memory limits or explicitly stated rules (e.g., VM1 and VM2 cannot be on the same node). Fortunately, feasibility often helps in our approach by pruning the solution space to make searching for good solutions faster. We discuss feasible solutions to our problem in more detail below.

Clearly, utility functions vary from site to site. Unfortunately, deciding upon goals for a computing environment, let alone a defining a utility function designed to achieve those objectives, can be onerous. Operational goals are often more educated intuition than absolute and very often conflict with other ideas, or notions, about how a site should operate. These facts make constructing a utility function based on those goals as much art as science. In Section 3.3 we present a few possible operational goals to give a sense for the myriad of scheduling policies (and hence, different utility functions) under which sites may wish to operate. Section 3.5 discusses construction of a utility function for our FMU problem.

Before moving on, we must point out that the opposite of utility is cost. We refer to our problem as trying to maximize utility or minimize cost. Where utility functions are bounded between zero and one, cost can simply be defined as one minus utility. Another definition when utility functions are not bounded may be to define cost as one divided by utility. In any case, when we talk of utility, our goal is to maximize our utility function. When we talk of cost, our goal is to minimize our cost function.

3.2 Related Work

We now present work closely related to our virtual machine scheduling problem. This problem has many similarities to process and thread migration for load balancing in physical clusters. Process and thread migration is a rich topic with a long history. Here, we only cover highly relevant work in process and thread scheduling. Another area similar to VM scheduling is application scheduling. In application scheduling large, distributed applications consisting of several processes (or tasks) compete for resources in a cluster. Schedulers must determine where processes in an application are able to receive the resources they desire. This is a special class of process scheduling worth discussing separately. Finally, we discuss more recent work on actual VM scheduling and how our approach differs.

Before proceeding, we must point out that, although many approaches to scheduling of applications in the Grid [FK98] and Planetlab [CCR+03] exist [CZBL00, TC00, VD02, VD03, CDK+04, OCP+06], these are not appropriate for our VM scheduling problem. In many ways, schedulers for the Grid and Planetlab are more difficult to design and implement. First, these schedulers must handle resource discovery in an environment where availability is highly dynamic. In our problem available resources are known by a centralized scheduler (although resource availability can change, mechanisms are in place to keep the scheduler updated as to what the global availability of resources is at any one time). In addition, schedulers for the Grid and Planetlab are not centralized and do not make decisions for all entities in the system. Rather, they attempt to find the best places for a single application to run, and have only the interest of a single user in mind. Though there is much we can learn from these schedulers, the key differences between our cluster environment and these geographically distributed computing platforms preclude us from directly applying their scheduling approaches to our problem.

3.2.1 Process Scheduling

Leland et al. [LO86] perform simulations of heuristics for initial placement and migration of processes in a homogeneous cluster environment. They studied the benefit of initial placement onto processors with the fewest resident processes, in addition to migration to processors with the fewest resident processes. Their simulations were driven using data from a five month trace of 9.5 million processes. They found that simple heuristics for initial placement and migration can significantly improve response ratios of processes that demand large amounts of CPU time without negatively impacting other processes. Though initial placement heuristics are future work in this dissertation, their initial placement heuristic places newly arriving processes on the least loaded processor. It is unlikely that such a simple initial placement heuristic would work in virtual clusters since users tend to start VMs in groups, then run similar jobs on each. Because of this this usage model, we expect round robin placement which evenly spreads newly arriving VMs across the cluster would be better. Many of the experiments we have run indicate that round robin placement is indeed better, but further exploration is necessary to definitively make this claim. Their migration heuristics are distributed and receiver initiated. No migrations occur until a processor becomes idle, at which time it sends a broadcast to all other processors indicating that it will accept bids to run processes. Upon receipt of all bids, the idle processor notifies the winner which migrates a process to the idle processor. This strategy is clearly not suitable for VM scheduling as it simply looks to keep all processors busy without sufficient regard to fairness or other possible cluster operational goals.

Harchol-Balter and Downey [HBD95] compare two cluster process migration strategies using trace driven simulation. The first strategy makes migration decisions based upon process age. The second migrates new processes based upon whether their name is on a list of processes known to be long lived. Both approaches resulted in considerable reduction in process slowdown in their simulations. Their first approach migrates only older processes which can likely benefit from migration. The migration decision is a function of memory size and number of processes at the source and desti-

nation hosts. Such a heuristic would not be suitable for VM scheduling which depends upon what resources VMs are consuming at the source and destination rather than the number of VMs at each node. In addition, migrations are only considered when new processes enter the system. This approach is clearly insufficient for VM scheduling since a VMs resource usage often changes considerably during its lifetime. As opposed to a single process with a single purpose, VMs consist of several processes, each with its own purpose resulting in a much more dynamic entity than a CPU bound process. Regarding their second approach, although we could maintain a list of VMs known to be heavy resource users, we have no notion of VM lifetime. In addition, because a VM runs several processes rather than a single process over its lifetime, it is more difficult to maintain a list which classifies VM behavior.

Nuttall and Sloman [NS97] examine workload characteristics to determine when dynamic process migration is beneficial. They separately consider both CPU and I/O bound workloads. They contend that dynamic migration is not useful under realistic workloads and that papers showing benefits use an unrealistic exponential distribution for process lifetimes. Interestingly, their simulations showed their dynamic migration heuristics do improve response time by about ten percent. In addition, they acknowledge in a postscript that their synthetic workloads were unintentionally more bursty than trace data collected in the Harchol-Balter and Downey study. They admit this probably decreased the effectiveness of their dynamic migration heuristics. Nuttall and Sloman use load vectors which indicate the level of resource usage for CPU and local and remote I/O to trigger migrations. When a host's load vector exceeds a threshold value, migration of a process who's load vector exceeds a threshold is considered. A centralized algorithm is used to determine to which host a process should be migrated. Nuttall and Sloman's process schedulers do not appear to consider the resource usage vectors componentwise. Rather, they lump all resources into a single vector which makes selection of process and destination "hit-or-miss". Also, as with most process scheduling heuristics, fairness does not seem to be a concern or goal of their scheduler.

MOSIX [BS99] is a system for cluster computing which preemptively and

transparently migrates processes between networked workstations for load balancing. MOSIX has no central control and each node operates as an autonomous system. Load balancing algorithms continuously attempt to balance load by migrating processes from higher to less loaded nodes. Migration decisions are based on process profiling and resource availability provided by their information dissemination algorithm. Online algorithms determine the best location for processes based on resource availability and process behavior. Migration decisions in MOSIX are based upon multiple criteria. However, their criteria are somewhat different than virtual clusters due to the differences in platform. For example, system-call rates and IPC volume factor into their migration decisions. Ultimately, the MOSIX scheduler is concerned with CPU load and available memory. Their decentralized approach requires information dissemination algorithms to keep nodes informed about resource availability, and nodes must cooperate to make any migration decisions. Though nodes in the MOSIX system work closely together, it is not clear how well their distributed scheduler applies to VM scheduling. Despite this, the goals of the MOSIX scheduler are, perhaps, the most similar in spirit to those of our VM scheduler.

Ultimately, there is no fundamental difference between process and VM scheduling. In fact, many hypervisors simply run VMs as processes in a privileged or host operating system (e.g., [kvm] and [vmwb]). Nonetheless, many process migration schemes focus on CPU, typically with the goal of minimizing makespan of jobs or load balancing CPU across a cluster.

Another important distinction between process and VM scheduling is that local resource sharing models are tunable in a virtualized cluster. Hypervisor schedulers are more sophisticated than operating system process schedulers, allowing for more flexibility in how VMs are scheduled to run locally and what resources they are allowed to access. Because of this, VM schedulers must understand the behavior of a node's underlying hypervisor scheduler to predict how VMs will behave on that node. This understanding is critical in making scheduling decisions.

Many process scheduling heuristics assume that processes will consume as

much CPU as they are given. This assumption is not always true and is certainly not the case with VMs. This assumption greatly simplifies the problem by removing the need to monitor process behavior in order to determine which processes should be migrated from overloaded nodes. Such a simplifying assumption cannot be made in VM scheduling.

Clearly, VMs bring a new usage model to a cluster environment and hypervisors provide additional knobs for local resource provisioning and functionality for monitoring resource consumption. As a result, there is a necessity to provide scheduling mechanisms which consider complex local resource sharing policies and can utilize additional resource usage information to make better scheduling decisions.

3.2.2 Application Scheduling

Application scheduling is a class of process scheduling in which large distributed applications compete for cluster resources. Typically, these applications run as services, so there is no notion of completion time. Application schedulers usually seek to provide resource guarantees (SLAs) to applications as a whole. This is typical in a datacenter.

Kelly [Kel03, Kel04] considered the problem of computing optimal resource allocations to agents given agent utility functions for those resources. Kelly related this problem to that of allocating resources in a utility data center. This problem is related to our problem of deciding where VMs should run to optimize utility based upon VM resource desires. However, Kelly's model lumps all cluster resources into a single pool on which agents bid for discrete sets of resources, called "bundles". We, on the other hand, do not partition our computing resources into discrete sets (e.g., two CPUs, three disks, etc.), but rather share those resources between all VMs competing for them on each node. Kelly's model applies to multi-tiered applications which scale horizontally. This model allows Kelly to formulate the allocation problem as a multi-dimensional multiple choice knapsack (MDMCK) problem for which efficient dynamic programming solutions exist. Because FMU does not exclude VMs from running, it

cannot be framed as a knapsack problem. Also, we do not treat our system as one large pool, so each node in our system would have to be a separate sack. Finally, Kelly's approach requires partitioning resources into bundles, which we do not do in FMU. For these reasons, Kelly's dynamic programming approach cannot be applied to our VM scheduling problem.

Other application placement heuristics have been proposed (e.g., [CDK+04, KKP+06]). The problem with applying these methods to VM scheduling is that most prior approaches attempt to place applications on nodes where their resource demands will be met, based upon prior knowledge of application resource demands (stated explicitly or through SLAs) or assumptions as to what those demands will be. This approach is in line with "contractual scheduling" defined in the chapter introduction and reduces the problem to one of packing, where a satisfying assignment is one which places applications on nodes which can deliver the resources they desire. Often, utility functions are used to distinguish good from bad satisfying assignments. Policies must be established to handle cases where satisfying assignments cannot be found.

On the other hand, we focus on a best-effort approach which does not necessarily provide virtual machines with specified amounts of resources or prevent VMs from running when sufficient resources are not available. Our target is that of a general purpose cluster environment where resources are shared in arbitrary ways in accordance with policies imposed by the site administrator. Though our scheduler can be setup to provide pseudo resource guarantees, we are not restricted to such settings. In this way, we wish to support a virtual cluster environment similar in spirit to a multi-user operating system. There, users' resource allocations are governed by the resource sharing policies of the underlying operating system; processes typically share resources fairly, but can be "niced" and limited in their resource consumption using `ulimit`.

3.2.3 Virtual Machine Scheduling

Recently, researchers have begun to consider VM scheduling. Sandpiper [WSVY07] develops policies for detecting and reacting to hotspots in virtual cluster

systems while satisfying application SLAs. Sandpiper determines when and where to migrate virtual machines under the constraints of meeting the stringent SLA requirements of a data center. Migrations are triggered when resource usage of a node exceeds a threshold value. Since Sandpiper is scheduling for SLAs, their problem reduces to one of packing. Their scheduler then looks to find new mappings which satisfy all SLAs while alleviating all hot spots. In cases of high loads, Sandpiper is unable to find solutions and gives up. This behavior is different from our best-effort approach which looks to find the best place for VMs to run based upon their resource desires.

VirtualIron provides LiveCapacity [liv] for scheduling VMs across pools of physical machines. LiveCapacity migrates VMs only if placing the VM on the destination does not exceed a specified utilization threshold. It is not clear whether only CPU is considered in this decision. LiveCapacity balancing is only re-evaluated every few minutes, so highly dynamic environments may not benefit from using this scheduler.

VMware offers their Distributed Resource Scheduler (DRS) [vmwa] as an add-on to their VM management suite. DRS continuously monitors VM loads and determines if any user-specified allocation rules have been violated. If a violation is found, resources are allocated to the VM by either migrating it to another server with more resources, or migrating other VMs away from the node on which the VM in violation is running. DRS appears to be another example of a packing approach where VM assignments which meet all VM resource requirements are sought. It is not clear what heuristics VMware uses to find satisfying assignments.

The LBVM (load balancing of virtual machines) system [lbv] seeks to migrate VMs to load balance workloads across a cluster. Each candidate for migration is listed in a configuration file and, at regular intervals, algorithms for each VM are executed. These algorithms may choose to migrate their VM based on arbitrary predicates (e.g., node five minute CPU load is greater than 70 percent).

It is not clear what the criteria for selecting a destination node is for LBVM. Also, there is no mention of coordination between algorithms to avoid unintended consequences such as hot spots arising due to all VMs in a round migrating to the same

physical node. The flexibility of separate algorithms for each VM may make such coordination difficult. Further information is needed before additional comparison between LBVM and our method can be made.

DRS and LBVM both provide flexible VM scheduling where per VM allocation rules can be specified. However, because they consider VMs individually, it is unlikely that they could easily support scheduling policies for global characteristics such as fairness to all hungry VMs. Also, both schedulers seem to be designed for environments with excess capacity (this is also true for LiveCapacity). That is, migrations are initiated when a node which can meet all of a VMs resource demands has been found. Again, this situation is not best-effort scheduling where VMs can tolerate varying levels of service.

3.3 Operational Goals

Determining where VMs should be placed in a cluster depends upon the operational goals of the site. Once these goals are defined, utility functions for achieving them can be specified. Though it is easy to imagine any number of operational goals, we give just a few examples of reasonable objectives.

Power Management: One popular area of interest today is power management. High costs of energy necessitate practical solutions to minimizing energy use. This is a reasonable goal in nearly any setting. Approaches to achieving this goal range from very simple to extremely complex. For example, a simple policy for achieving this goal may be migrating VMs off sets of PMs when overall utilization of particular computing resources are below threshold values [KUS+04]. More complicated policies may strive to place VMs on machines on which they have been determined to run more energy efficient [NS07].

Another reasonable power management goal proposed for use in the UCSD SysNet group is to balance power consumption across racks. This goal emerged from a desire to prevent racks from tripping breakers in a machine room lacking proper power

provisioning.

Simple VM Placement: A reasonable policy in VM placement may be to try to evenly spread VMs across physical machines when workloads are known to be similar with the goal of load balancing across physical machines. Another may be to simply put constraints upon how many VMs can run on a physical machine with the goal of better accommodating workload spikes.

Network Traffic: Since VMs on the same node communicate without using the physical network interface, a practical policy may be to place heavily communicating VMs on the same physical machine when possible. Another may be to simply place them in the same rack so that their network traffic is restricted to a single switch.

Service Level Agreements: Where contracts are involved, administrators need the ability to guarantee levels of performance to their clients. These service level agreements (SLAs) define constraints under which a site may operate. Strict SLAs in which guarantees are absolute essentially reduce VM scheduling to a multi-dimensional multiple knapsack problem. While complicating the problem, soft SLAs which guarantee resources with some probability can result in much higher resource utilization than strict SLAs. Our approaches (discussed below) may also be good for providing soft SLAs.

Resource Utilization: Using the UCSD SysNet group as an example, allocating computing resources to users at the granularity of physical machines often results in poor resource utilization. Thus, a reasonable goal for a VM scheduling policy may be to maximize overall resource utilization, where multiple resources are considered important.

Unfortunately, a policy of maximizing overall resource utilization which does not consider fairness can result in starved and frustrated users. So, a goal to maximize overall resource utilization while being fair to those sharing the resources is more tenable. This is the exact goal of the SysNet installation. We refer to this as the Fair Maximum Utilization (FMU) problem and present it in detail in the following section.

As a final note, notice that many operational goals are actually constraints (e.g., hard SLAs) whereas others are rather fuzzy (e.g., FMU) and lend themselves to being expressed as utility functions. Though constraints which do not greatly restrict the feasible solution space can be enforced in our solution methodology, satisfying complex constraints which greatly limit the ratio of feasible to all possible assignments (feasible or infeasible) can be problematic. When assignment feasibility checks cannot be performed quickly, the solution methodology presented here breaks down.

3.4 Fair Maximum Utilization

As introduced above, the goal of the Fair Maximum Utilization (FMU) problem is to maximize overall resource utilization across a site while maximizing fairness to VMs contending for those resources. We now clarify the terms "overall resource utilization" and "fairness".

Overall resource utilization: Overall resource utilization implies that utilizations of a specified set of resources (e.g., CPU, network, disk I/O, memory) all factor into the total utilization calculation. As a simple example, consider a site consisting of two PMs offering ten units of CPU and ten units of network transmit bandwidth each (here, CPU and network transmit are the only resources being considered important). Now, imagine scheduling the four VMs in Table 3.1.

Table 3.1 Sample VM resource demands.

VM Name	CPU Units	Network TX Units
VM1	12	10
VM2	11	15
VM3	5	1
VM4	6	2

Figure 3.2 depicts VM resource allocations for four possible assignments. In Figure 3.2 a, VM1, VM2, and VM3 are assigned to run on PM1 and VM4 is assigned to

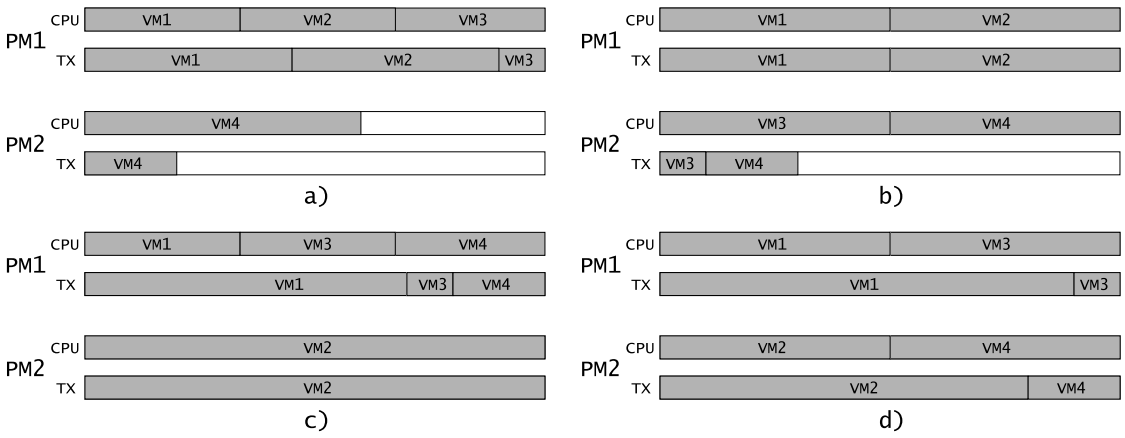

Figure 3.2 Virtual machine resource allocations for three different assignments.

run on PM2. The length of each shaded rectangle represents the amount of the resource received by that VM. Notice that, in this schedule, PM2 is not being fully utilized.

In this section we assume that resources are fairly shared locally in a work conserving manner. For example, network transmit on PM1 in Figure 3.2 a is being shared by VM1, VM2, and VM3. VM3 only desires one unit and is able to get it since there are only two other VMs contending for that resource. VM1 and VM2 are then able to receive their fair share of $10/3$ units, plus an additional $7/6$ units each (total of 4.5 units) since VM3 is not using all of its fair share.

The schedule in Figure 3.2 b fully utilizes available CPU, but not network transmit bandwidth. So, this schedule does not maximize overall resource utilization.

Clearly, any schedule which separates VM1 and VM2 maximizes overall utilization as shown in Figure 3.2 c and d. Notice that, concerning utilization, VM3 and VM4 can be placed anywhere while separating VM1 and VM2 since this already results in 100 percent overall resource utilization. However, placing them both on the same node as in Figure 3.2 c clearly hurts fairness, discussed next.

Fairness: Fairness implies that no VM is able to consume substantially more of a particular resource than another VM desiring more of that resource than it is receiving. We define the following terms:

hungry VM: A VM desiring more of a particular resource than it is currently receiving.[2]

satisfied VM: A VM receiving as much of a particular resource as it desires.

pseudo-hungry VM: A satisfied VM receiving more of a particular resource than the average amount being received by the *hungry* VMs for that resource.

Briefly, pseudo-hungry VMs arise from imbalance in the schedule. For example, imagine a system of two physical machines offering 100 CPU units each and three VMs desiring 75 CPU units each. If we place two VMs on the first PM and the third on the second PM, the two VMs on the first PM are hungry VMs and the VM on the second PM is a pseudo-hungry VM. In this case, the average being received by hungry VMs is 50 units. Since the VM on the second PM is receiving 75 CPU units (and is therefore satisfied), it is classified as pseudo-hungry since it is receiving more than the average amount being received by the hungry VMs for CPU. Note that the set of pseudo-hungry VMs is a subset of the set of satisfied VMs.

Notice that each resource, r, under consideration has a set of *hungry* VMs associated with it:

$$H_r = H_{r,1} \cup H_{r,2} \cup \cdots \cup H_{r,M}$$

where M is the number of physical machines.

Letting V be the set of all VMs, the set of *satisfied* VMs for resource r, is simply $S_r = V \setminus H_r$.

Using the above definitions, fairness implies that *hungry* VMs of a particular resource should all be receiving approximately the same amount of that resource, with no *satisfied* VMs receiving more, across all physical machines. Letting ρ_r be the set of *pseudo-hungry* VMs for resource r, define the set of *augmented hungry* VMs as:

$$\hat{H}_r = H_r \cup \rho_r$$

Then, the goal of fairness is to ensure that all VMs in $\hat{H}_r$ are receiving approximately the same amount of resource r. We refer to this as being "fair to hungry VMs" rather

[2]These are referred to as "hogs" in the scheduler code.

than "fair to augmented hungry VMs" since fairness to pseudo VMs is meaningless. The mere existence of *pseudo-hungry* VMs implies that the *hungry* VMs are being treated unfairly.

Revisiting the above example, Figure 3.2 a is not fair since a satisfied VM, VM4, is receiving more CPU (6 units) than the average allocation of those desiring more ($10/3$ units). In other words, not all VMs in $\hat{H}_{CPU} = \{VM1, VM2, VM3, VM4\}$ ($VM4 \in \rho_{CPU}$) are receiving the same amount of CPU.

Figure 3.2 b is a fair schedule since all VMs desiring more CPU and transmit bandwidth are all receiving the same amount of those resources. That is, all VMs in $\hat{H}_{CPU} = \{VM1, VM2, VM4\}$ are all receiving the same amount of CPU (five units) while all VMs in $\hat{H}_{TX} = \{VM1, VM2\}$ are all receiving the same amount of network transmit bandwidth (five units). This schedule, however, does not maximize overall resource utilization as do those of Figure 3.2 c and d.

As mentioned above, the schedule in Figure 3.2 c is unfair since VM2 is receiving more CPU than VM1, VM2, and VM3 (ten units vs. $10/3$ units) which all desire more CPU. Also, this schedule allocates ten units of network transmit to VM2 while only seven units are given to VM1, which desires more.

Though not completely fair, the schedule of Figure 3.2 d best meets our criteria of maximizing overall utilization while being fair to *hungry* VMs. Notice that VM3 and VM4 both receive as much network transmit bandwidth as desired since they do not exceed their local fair share of that resource. The remaining network transmit is consumed by VM1 and VM2 on their respective PM since the local schedulers are work-conserving[3] and their demand exceeds the amount of that resource available locally.

The schedule in Figure 3.2 d is only slightly unfair to VM2, which only receives eight units of network transmit bandwidth versus nine units for VM1. This, however, is the most fair schedule possible. One remedy for this slight imbalance might be to periodically swap VM3 and VM4. However, VM migration is not free, and an administrator should consider how much unfairness she is willing to tolerate before taking

[3]Work-conserving scheduling ensures that resources do not go idle while there is demand.

such actions.

3.4.1 NP-hard Proof for Fair Maximum Utilization

As evidence that the Fair Maximum Utilization (FMU) problem is not trivial, I prove that it is in the class of NP-hard problems.

Formally, the FMU problem[4] is:

Given:

- an ordered set of R resources,
- a multiset of M physical machines with some capacity of each resource to fairly share,[5]

  $$P = \{\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_M | \vec{p}_i \in \mathbb{R}^R, i = 1, \ldots, M\}$$

  where each component of $\vec{p}_i$ is the amount of that resource shared at machine i,
- a multiset of N virtual machines with some desire for each resource,

  $$V = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_N | \vec{v}_i \in \mathbb{R}^R, i = 1, \ldots, N\}$$

  where each component of $\vec{v}_i$ is the amount of that resource desired by VM i

find an assignment of VMs to PMs which maximizes overall resource utilization

$$U = \frac{1}{R} \sum_{i=1}^{R} (\frac{w_i}{c_i}) \sum_{j=1}^{M} u_{ij} \tag{3.1}$$

where,

w_i = weight of resource i, $\sum_{i=1}^{R} w_i = 1$,
c_i = total capacity of resource i
u_{ij} = amount of resource i consumed at node j

while maximizing fairness to hungry VMs

$$F = \frac{1}{R} \sum_{i=1}^{R} w_i \frac{(\sum_{j=1}^{H_i} a_{ij})^2}{(H_i \sum_{j=1}^{H_i} a_{ij}^2)} \tag{3.2}$$

where,

[4] Actually, the FMU problem is more general, where a physical machine can offer multiple instances of a particular resource and a VM can desire multiple instances of a resource. Nonetheless, for proving FMU is NP-hard, the definition here is sufficient and far more tidy notationally.

[5] That is, the underlying scheduler will fairly share the resource in a work-conserving manner.

w_i = weight of resource i, $\sum_{i=1}^{R} w_i = 1$,
H_i = number of augmented hungry VMs for resource i
a_{ij} = amount of resource i allocated to augmented hungry VM j

Equation 3.2 is known as Jain's Fairness Index [JCH84].[6] Notice that $U \in [0, 1]$ and $F \in [0, 1]$.

Theorem 3.4.1. *FMU is NP-hard.*

Proof. The $3-PARTITION$ problem has been shown to be NP-complete in [GJ79]. So, to prove Theorem 3.4.1, we show that $3-PARTITION \leq_p$ FMU.

Let S be a multiset of $N = 3M$ integers such that the sum of the integers in S equals MB and $B/4 < i < B/2 \ \forall i \in S$. The $3-PARTITION$ problem asks whether S can be partitioned into M subsets such that the sum of the integers in each subset equals B.

The reduction to an instance of FMU is as follows. For each $i \in S$, add a VM to V with scalar desire equal to i. Next, add M VMs to V with scalar desire $MB + 2$ (i.e., we add M hungry VMs). Let P be the multiset of M scalars with value $MB + 1$. Note that, in this case, we have an instance of FMU with $R = 1$ (it is not important what the resource actually is, just that there is a single one).

Claim: $\exists$ an assignment of VMs to PMs where $U = 1$ and $F = 1 \iff S$ can be partitioned into M subsets with sum B.

($\Rightarrow$) Assume $\exists$ an assignment of VMs to PMs with $U = 1$ and $F = 1$. Since the capacity of each physical machine is $MB+1$, the only possible way to get $U = 1$ is to place a hungry VM on each physical machine. Since we only have M hungry VMs, we clearly must have exactly one hungry VM per physical machine.

Now, since we have exactly one hungry VM per physical machine and $F = 1$, it must be the case that the sum of resource consumed by the non-hungry VMs placed on each physical machine must be equal. In particular the sum of resource consumed by the

[6] We originally found that the coefficient of variation worked well as a fairness measure. However, the coefficient of variation is not guaranteed to be in [0,1]. The astute reader may have noticed that $F = \frac{1}{1+COV^2}$.

non-hungry VMs placed on each of the M physical machines must equal B. Therefore, S can be partitioned into M subsets with sum B.

($\Leftarrow$) Assume S can be partitioned into M subsets with sum B. Then, placing the VMs in each of the M subsets on a different physical machine along with exactly one hungry VM will clearly give us an assignment with $U = 1$ and $F = 1$.

The above reduction is obviously polynomial time, so $3{-}PARTITION \leq_p$ FMU. $\therefore$ FMU is NP-hard. □

Notice that the FMU problem as defined above is for the case where each physical and virtual machine has a single availability and desire for each resource, respectively. The more general case is each physical machine offering multiple instances of a particular resource (e.g., multiple CPUs) and each VM having multiple desires for a particular resource (e.g., multiple virtual CPUs). It is easy to see that the above proof applies to the more general problem since the above reduction also polynomial time reduces $3{-}PARTITION$ to an instance of this more general form.

3.5 Problem Classification

In its purest form, FMU is an instance of a multiple objective combinatorial optimization (MOCO) problem. *Combinatorial optimization* is the search for an "optimal arrangement" of a set of discrete objects [Law01]. In FMU, the objects are VMs and the arrangements are placements of VMs onto physical machines. So, we seek to find optimal assignments of VMs to PMs. Optimal refers to minimizing (or maximizing) a function, typically referred to as the *cost function*, whose domain is the set of all feasible assignments of VMs to PMs. In standard *combinatorial optimization*, this is a scalar valued function. Recall, however, that FMU possesses two objectives: maximize Equation 3.1 while simultaneously maximizing Equation 3.2. Hence, we have a MOCO problem. Formally, a MOCO problem is:

Given a discrete vector valued cost function:

$$C(s) : s \in S \rightarrow \mathbb{R}^N \tag{3.3}$$

where

S is a finite discrete set of feasible solutions

$N \in \mathbb{Z}^+$

find an arrangement, $s* \in S$, which minimizes cost:

$$C(s^*) \leq C(s) \forall s \in S$$

Possible strategies

In MOCO, the objectives are often in conflict. Therefore, one must give sufficient consideration to defining the total order "$\leq$". For example, defining the FMU cost function as

$$C(s) = \begin{bmatrix} 1 - U(s) \\ 1 - F(s) \end{bmatrix} = \begin{bmatrix} C_U(s) \\ C_F(s) \end{bmatrix} \tag{3.4}$$

one possibility for the total order is lexicographical ordering where

$$C(s_1) \leq C(s_2) \text{ if } C_U(s_1) < C_U(s_2) \vee (C_U(s_1) = C_U(s_2) \wedge C_F(s_1) \leq C_F(s_2))$$

Defining the total this way treats utilization as the primary consideration, with fairness only being considered once utilization has been maximized. This is often far from the best strategy. It is easy to imagine scenarios where a very slight decrease in utilization between one assignment and another dramatically increases fairness.

Fuzzy logic [ZKY96] is another option for treating multiple objectives. In this approach, objects have a degree of membership in a set, and may be a member of multiple sets. This degree of membership is defined by membership functions with a range in $[0, 1]$. In FMU, the object are values of utilization and fairness of an assignment.

In fuzzy logic, linguistic values are used to define the sets wherein assignments may fall for each of the objective functions. For example, *high* could be used to

characterize the values of utilization and fairness for an assignment. Then, satisfaction of a schedule is determined by its degree of membership in the sets of high utilization and high fairness. Rules can then be defined to characterize the "goodness" of solutions. Since we do not employ fuzzy logic in our solution approach, the interested reader can refer to [ZKY96] for additional information.

A simplification

Rather than defining a total order or resorting to fuzzy sets, FMU can be simplified into a scalar valued function. Experience has shown that simplification works well in practice.

Certainly, there are several possibilities for converting Equation 3.3 into a scalar valued function. One logical choice may be to take the L^p-norm of the solution vector of Equation 3.3:

$$C(s) = |\vec{x}|_p,\ \vec{x} \in \mathbb{R}^N \tag{3.5}$$

where,

$$|\vec{x}|_p \equiv \left(\sum_i |x_i|^p \right)^{1/p}$$

The most common norm , often simply denoted $|\vec{x}|$, is the L^2-norm:

$$|\vec{x}| = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_N|^2}$$

What is interesting about the L^p-norm for FMU is that, at its extremes,

$$|\vec{x}|_1 = \sum_i |x_i|$$

and

$$|\vec{x}|_\infty \equiv \max_i |x_i|$$

larger components of the vector are treated with increasing emphasis. In other words, as $p \to \infty$, we move from focusing on each objective with equal weight, to focusing exclusively on the objective with highest cost. So, when utilization is low (i.e., its cost is high) in FMU, its effect on the cost function will be more dramatic than if utilization

and fairness were treated equally. What is a reasonable value for p in FMU? We discuss experiments for various values in Chapter 4.

Another possibility for transforming Equation 3.3 into a scalar valued function is to simply compute a weighted sum of each component. For FMU, this yields,

$$C(s) = C_U(s) + wC_F(s) \tag{3.6}$$

where $w \in [0, 1]$. This works well, but in practice, a better scalar valued cost function is:

$$C(s) = C_U(s) + w(U(s))C_F(s) \tag{3.7}$$

That is, make the fairness weight a function of utilization. The intuition here is that, when utilization is low, fairness is not very important since hungry VMs should already be placed where they are getting as much as the system can possibly give them. If this were not the case, utilization could be increased by moving hungry VMs to PMs where they will get more of their coveted resource. For FMU, we found the following cost function:

$$C(s) = C_U(s) + (1 - C_U(s))^2 C_F(s) \tag{3.8}$$

works well in practice. We use this cost function in the experiments in Chapter 4.

As a final note, none of the above modifications to the cost function change the fact that FMU is still NP-hard since $C(s) = 0 \iff U(s) = F(s) = 1$ in all cases.

3.6 Hungry Detection

The distinction between *hungry* and *satisfied* VMs is important to the FMU problem. Certainly, Equation 3.2 cannot be evaluated without knowing which VMs are in $\hat{H}_r$ for each resource, r. If we had a mechanism by which a VM could notify us when it is not getting the resources it desires, we could easily classify those as hungry and all others satisfied. Unfortunately, the black box nature of our scheduling system precludes the use of such mechanisms. So, the job of distinguishing between hungry and satisfied

VMs fall to our system. Making this distinction in a running system requires careful consideration.

The first step in hungry detection is understanding how resources are shared on a physical machine. Thus, the sharing policy of the underlying VMM must be properly understood. The Usher scheduling plugin allows for specification of arbitrary resource sharing models for the resources of interest. One must simply subclass the `SharingModel` class of the scheduler code.

The CPU sharing model we use is based on the Xen [BDF+03] Credit scheduler. This scheduler fairly shares physical processors between virtual machines in a work-conserving manner. The scheduler provides transparent, cross CPU migration and can be tuned per-VM with weight and cap specifications. In our work, we assume that all VMs are weighted equally, though this restriction is not necessary for our solution methodology.

Keep in mind that, a virtual CPU[7] (VCPU) can only consume up to what is provided by a single processor in a multiprocessor PM. So, if a single threaded (i.e., single VCPU VM) VM is the only VM running on a two processor PM, it would only consume cycles offered by a single CPU rather than both. On the other hand, a VM with two VCPUs could potentially consume all available cycles from both physical CPUs. As you can imagine, figuring out how many cycles each VCPU will receive for arbitrary numbers of VMs with arbitrary numbers of VCPUs on machines with several physical CPUs can be difficult. Figuring out which of these VCPUs is hungry adds another challenge.

Though an example using the Xen Credit scheduler model would be overly complicated, a very simplified example will help to clarify how understanding of the underlying scheduler helps us to determine which VMs are hungry and which are satisfied. Imagine a fair CPU scheduler on a single CPU physical machine hosting only single VCPU VMs. Let N be the number of VMs and Q be the number of CPU cycles per second being offered by the physical machine. In this case, the following simple

[7] A virtual CPU is the CPU abstraction presented to the VM by the hypervisor. VMs can have any number of virtual CPUs up to the limit of the VMM.

heuristic works to determine which VMs are CPU hungry:

If the CPU load on the physical machine is Q, then

1. find the maximum CPU allocated to a VM, V_{MAX}
2. add all VMs with CPU allocation equal V_{MAX} to $\hat{H}$

$\hat{H}$ is the set of hungry VMs.

This, of course, is a very simplified example. Also, since the Xen scheduler is not exact, some error must be tolerated. For example, we may need to change the condition on machine load to $0.95Q$ and put all VMs with CPU allocation equal to $0.95VM_{MAX}$ in $\hat{H}$. Due to these inaccuracies, any black box strategy for categorizing VMs as hungry or satisfied will not be perfect.

3.7 Heuristic Approaches to FMU

Loosely speaking, if you could count the number of computing clusters in existence today, you would likely have a good guess at the number of different cluster operational goals as well. Many of those goals lend themselves to the use of virtual machines. For this reason, any approach to solving the virtual cluster scheduling problem must be sufficiently general to be applicable to a wide range of virtual cluster usage scenarios.

As with FMU, many combinatorial optimization problems are NP-hard. Those which are not often lend themselves to much simpler solution techniques. Occasionally, even NP-hard scheduling problems can be tackled when known greedy algorithms or polynomial time approximation schemes (PTAS) exist. Of course, one must be willing to sacrifice optimal solutions for near optimal solutions when using these, but this is often acceptable in scheduling. Indeed, near optimal solutions are acceptable for FMU since finding good schedules is only part of our problem.[8] Unfortunately, no good PTAS has been discovered for the general FMU problem. However, simplified versions of the problem do present themselves to efficient solvers.

[8] As discussed below, we still have the problem of getting to the new configuration.

3.7.1 A Simple Single Resource Approach

Restricting FMU to a single resource greatly simplifies our problem and known greedy heuristics and PTAS for similar problems can be employed to assist in finding good schedules. As an example, the reader may be wondering about the FMU's relationship to bin packing or the knapsack problem. Though there is some similarity between FMU and these problems, they are not the same. The Related Work section (Section 2.2) elaborates on the key differences between these problems. Nonetheless, reducing FMU to a single resource allows us to use either greedy methods or PTAS for bin packing to help find good assignments. The heuristic for this is as follows:

1. bin pack all satisfied VMs (e.g., using first fit decreasing [D07] or a known PTAS if tighter bounds are desired)
2. round-robin place all hungry VMs where they will receive the highest CPU allocation

We call this the Divide and Bin Pack (DBP) algorithm. Note that this heuristic still requires hungry detection.

Using a scheduling simulator written to facilitate quick evaluation of different scheduling heuristics, we tested DBP with 25 physical nodes, 300 VMs, and 45 CPU hungry VMs. Load for the remaining VMs was taken from a Pareto distribution with shape parameter of one. We found that DBP routinely found schedules with 99 percent CPU utilization with the coefficient of variation of CPU allocation to hungry VMs less than 0.1. In other words, DBP met the goals of FMU quite well. Of course, this approach is only valid for settings involving a single instance of a single resource at each PM and VMs with only a single virtual instance of that resource. This is clearly not in line with hardware and computing environments of today.

For this reason, we look to more powerful solution methodologies for FMU. A popular approach for many combinatorial optimization problems is to use what are known as *metaheuristics*. The term "metaheuristic" refers to a heuristic which guides a lower-level heuristic in a search for an optimal solution to a given utility function. We look at one such heuristic in Section 3.8 called Simulated Annealing. We will also

explore simpler greedy approaches to FMU which arrive at new schedules much quicker than SA, but are less likely to find very good schedules. First, however, we introduce a very simple scheduling heuristic which will give us a baseline with which we can compare our more sophisticated schedulers. We call this heuristic "Balanced CPU" (BCPU).

3.7.2 Balanced CPU Scheduler

As a baseline, the Balanced CPU (BCPU) scheduler is a very simple scheduler with which we compare results of our more complex schedulers discussed below. The purpose of this comparison is to determine how much better (or worse) our schedulers perform than an extremely simple solver. The BCPU solver simply tries to balance CPU load across all PMs by minimizing the maximum load of all physical nodes in the system. It only considers the CPU resource and knows nothing about hungry and satisfied VMs (unlike the more sophisticated DBP scheduler mentioned above). All decisions are based on the current VM CPU usage, not what the VM might actually want. Note, this is identical to the minimum makespan problem (where time is now load), which itself is NP-hard [GJ79]. We, however, apply a known $4/3$ approximation algorithm [Gra69] for minimum makespan to our problem. The heuristic for BCPU is as follows:

1. sort VMs in decreasing order of their CPU allocations
2. assign VM to PMs in sorted order, scheduling the VM on the PM that has the lowest CPU consumed so far

Note that one issue with the above BCPU scheduler is that migration count can needlessly increase when two VMs with the same desire swap positions in the ordered list due to scheduling imperfections. For example, two VMs with 100MHz desire may get 99MHz and 101MHz respectively in one sample, then 100MHz and 98MHz the next. This will cause the BCPU scheduler to assign them to a PM in a different order, likely resulting in needless migrations of these VMs. To alleviate this, we propose a second BCPU scheduler we call BCPU2. This uses the simpler two approximation for

minimum makespan presented in [Gra69]. Applying this to FMU yields the following algorithm:

1. sort VMs in order of their fully qualified domain name
2. assign VM to PMs in sorted order, scheduling the VM on the PM that has the lowest CPU consumed so far

Note that [Gra69] orders jobs arbitrarily which, in essence, is what sorting by VM fully qualified domain name does. Since this will always sort VMs in the same order (excluding VM arrival and departure), the needless swap issue above is eliminated. We present results for both schedulers alongside those for our more elaborate schedulers in the following chapter.

We make one small modification to the above algorithms by introducing a migration resistance in step 2 to avoid excessive migrations. If the CPU consumption after placement on the PM with the lowest utilization in step 2 is within five percent of the CPU consumption of the VM's current PM (were the VM placed there), we leave the VM assigned to its current PM. This greatly reduces migrations which do not strongly affect the characteristics of the final schedule.

As a final comment, we could certainly conceive of other possible tweaks to our simple schedulers. However, tweaking simple schedulers quickly complicates them. Since our intent is to show that our schedulers are better than extremely simple schedulers, we do not consider additional modifications to our BCPU schedulers. Note, we do not claim that a simple scheduler which performs as well as or better than our more sophisticated schedulers does not exist.

3.8 Simulated Annealing

Simulated annealing (SA) [KGV83] is a well known metaheuristic for solving combinatorial optimization problems. SA can be viewed as an improvement over the simple hill climbing/descending algorithm to allow for non-convex solution landscapes to be searched for globally optimal solutions. The inspiration for this method comes

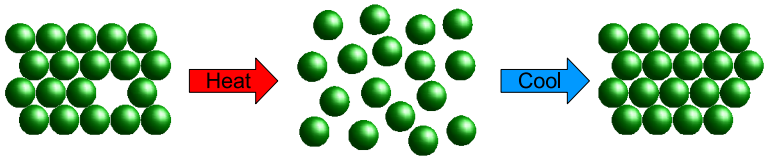

Figure 3.3 Annealing of a solid to reduce its internal energy.

from annealing in solids whereby a solid is heated to a temperature at which the chemical bonds between its atoms begin to break, allowing atoms to move freely. The solid is then slowly cooled to allow atomic bonds to reform. Figure 3.3 depicts the process of heating and slowly cooling of a solid. The new state of the solid presumably has a much lower internal energy than its previous state. The slow cooling is key in allowing atoms to find these lower energy configurations.

Similarly, SA seeks optimal solutions by allowing candidate solutions with higher energy to be accepted with probability based upon the difference in energy between the current and candidate solution and current simulation temperature. At high temperatures, nearly all candidate solutions are accepted. As the simulation runs, temperature is gradually decreased, causing solutions with higher energy (i.e., less desirable) to be accepted with less probability. So, as temperature decreases, SA behaves more and more like simple hill climbing/descending (hill descending in this case).

In the remainder of this section, we use the terms “energy” and “cost” interchangeably since utility functions in optimization problems are often referred to as “cost functions”.

The accepting of higher energy solutions is what enables SA to climb away from local minima and more thoroughly explore the entire solution space. Hill climbing/descending alone does not posses this property. Figure 3.4 contains an example solution space searched by SA. Here, cost is plotted against the states in the feasible solution space.

Starting at the “`current`” state, a simple hill descending algorithm would terminate at state S_1 as its solution. SA’s ability to climb out of “bad” local minima allows it to move out of the S_1 trough and over into the S_2 trough, resulting in a better

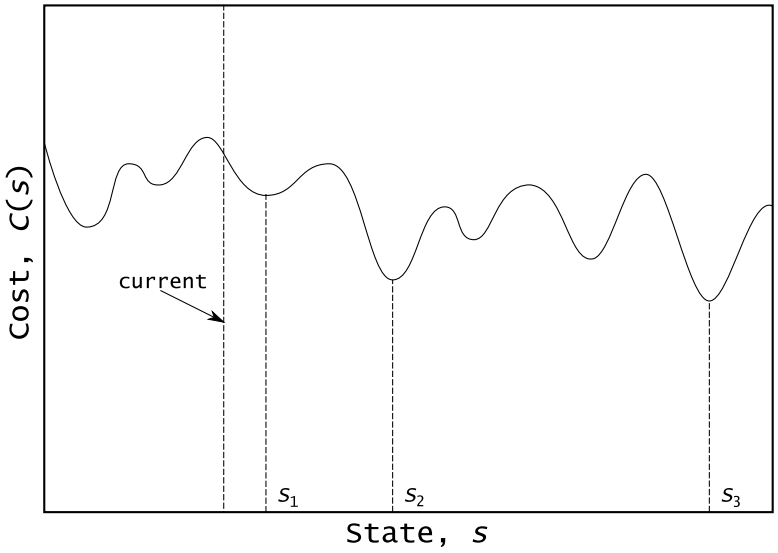

Figure 3.4 Sample solution landscape.

solution at state S_2. Given sufficient running time, a properly constructed SA solver would eventually find the global minima at S_3. The ability to escape troughs in the solution landscape is a key feature of metaheuristics.

3.8.1 The SA Algorithm

We apply the SA algorithm as originally proposed in [KGV83] to FMU. Pseudocode for the [KGV83] variant of SA is as follows:

```
SA(s_cur, alpha, mod, temp, time_max, cost_thr)
    time = 0
    cost_cur = cost(s)
    cost_best = cost_cur
    s_best = s_cur
    while time < time_max
        s_new = neighbor(s)
        cost_new = cost(s_new)
        if cost_new ≤ cost_cur
            s_cur = s_new
            cost_cur = cost_new
            if cost_new ≤ cost_best
                cost_best = cost_new
                s_best = s_new
                if cost_new ≤ cost_thr
                    return s_new
        else if random() < exp((cost_cur − cost_new)/temp)
            s_cur = s_new
            cost_cur = cost_new
        if not time%mod
            temp = alpha ∗ temp
        time = time + 1
    return s_best
```

where,

$alpha$ - temperature reduction factor

$cost()$ - function returning cost of a given state

$cost\_best$ - lowest cost (energy) encountered so far

$cost\_cur$ - current state cost

$cost\_new$ - new state cost

$cost\_thr$ - threshold value for early termination

$exp()$ - exponential function

mod - number of annealing steps at each temperature

$neighbor()$ - function returning a neighbor of a given state

$random()$ - function returning a random number in $[0, 1]$

s_best - best state visited so far
s_cur - current state
s_new - new state to examine
$temp$ - simulation temperature
$time\_max$ - maximum time to run simulation

This variant uses the *Metropolis procedure* [MRR+53] to simulate annealing at a given temperature. Here, at each temperature, a series of mod states are considered. The $neighbor$ function is used to probabilistically choose a new state in the neighborhood of the current state. Briefly, this function typically performs a small perturbation to the current state (e.g., randomly migrate a VM) to yield a new "nearby" state for consideration. For each new state, if the cost is less than the current state, the new state becomes the current state. If the cost is greater than the current state, it can still become the current state if the condition $random() < exp((cost\_cur - cost\_new)/temp)$, known as the *Metropolis criteria*, is met. This is what allows SA to climb out of troughs to explore larger regions of the solution space. The heuristic repeatedly runs the Metropolis procedure at each temperature for mod steps, decreasing temperature by a factor of $alpha$ after each mod steps, until the total time reaches $time\_max$.

3.8.2 Setting SA Parameters

Notice that there are several parameters which must be specified for the SA algorithm: $s\_cur, alpha, mod, temp, time\_max, cost\_thr$. As with most metaheuristics, SA must be properly tuned to the problem at hand. Here, the input parameters must be tuned for the given $cost$ and $neighbor$ functions, and, to some extent, the acceptance criteria (Metropolis criteria in this case).

Worth mentioning is the $cost\_thr$ parameter, sometimes referred to as the "stopping criteria". Setting this is at the discretion of the experimenter. If an acceptable value of the cost function in known, setting $cost\_thr$ can often save substantial simulation time when an acceptable assignment is close to the current assignment.

Indeed, there is a wealth of information on tuning SA, as well as, optimizations to the original algorithm. However, it is not the intent of this chapter to study properties

and convergence aspects of SA (or any other metaheuristic) applied to FMU, but rather show that metaheuristics are a reasonable approach to efficiently finding good mappings of VMs to PMs for FMU scheduling. Improving the efficiency of SA, in addition to, the behavior of other metaheuristics applied to FMU is beyond the scope of this dissertation.

Below, we present only a high-level discussion on setting parameters and defining the neighbor function of SA. The interested reader is referred to [SY99] for in depth coverage of the theory for properly setting these values and defining this method.

Neighbor function: The $neighbor$ function for SA must be given careful consideration. An important property of this function is that it have the ability to reach any state in the feasible solution space from any other state in a finite number of steps. For example, a $neighbor$ function for FMU which swaps VMs between two PMs does not posses this property since all PMs would always host the same number of VMs. In addition, efficient searching requires that the number of steps between any two states in the valid solution space be sufficiently small.

Another important property of the $neighbor$ function is that it not be biased toward very good moves. The reason for this is that very good moves in a states neighborhood can move the current solution into a deep local minima. This may prevent SA from escaping to find other, potentially better solutions. We slightly relax this requirement for FMU, as will be explained in Section 3.8.4 where we define our neighbor function.

Cooling schedule: The $alpha$, mod, $temp$, and $time\_max$ parameters define the "cooling schedule" for SA. It is common to determine values for these parameters through trial and error, although methods have been proposed for setting and dynamically tuning these as a simulation runs [SY99].

As with annealing of solids, the cooling schedule is important to finding a global minimum. If cooling is too fast, crystal lattices do not form in the solid and internal energy is not minimized. Similarly, a fast SA cooling schedule (small mod and/or small $alpha$) will quickly fall into a local minima and be unable to escape since the prob-

ability of accepting higher cost moves approaches zero as temperature approaches zero. Very slow cooling schedules require very long running times to find good local minima, or result in $time\_max$ being reached before the simulation ends, stopping before a good local minima is found. We use parameter sweeps in our scheduling simulator to discover reasonable constant values for these.

Efficiency: Two major contributors to the computational cost of SA are the calculation of the cost and neighbor functions. These should be fairly inexpensive since both will be computed at each step. The cost function should be such that only local changes need to be recomputed. Though our cost function is somewhat hefty, this property does hold.

3.8.3 SA Parameters for FMU

As shown in Section 3.8, there are few tuning knobs for SA which affect new schedule quality. We used our VM scheduling simulator to perform parameter sweeps of $alpha$, $temp$ (initial temperature), $time\_max$, and mod to determine good values for these parameters for FMU. Though we did not find excellent exact values for these in our simulations, we did discover acceptable ranges for each parameter. We summarize these ranges in Table 3.2. There, M and N are the number of VMs and PMs, respectively. As should be expected, good ranges for $time\_max$, and mod were dependent upon simulation size.

Table 3.2 Acceptable simulated annealing parameters for FMU.

Parameter	Range
$alpha$	$[0.7, 0.9]$
$temp$	$[7.5, 12.5]$
$time\_max$	$[\frac{MN}{2}, MN]$
mod	$[\frac{time\_max}{2N}, \frac{2time\_max}{N}]$

Unless otherwise specified, we chose $alpha = 0.8$, $temp = 10$, $time\_max = \frac{MN}{2}$, and $mod = \frac{2time\_max}{N}$ for the experiments in this dissertation.

One pitfall to avoid with SA is the desire to reach a solution quickly by spec-

ifying low values for $alpha$, $temp$, or mod. This results in quick descent into a local minima which may be of poor quality (E.g. S_1 in Figure 3.4). In addition, a low value for $time\_max$ does not allow the algorithm sufficient time to "feel around" the solution space in search of good assignments. The temptation to set these values low stems from the desire to reduce migration costs. We discuss this tradeoff and methods for controlling high migration counts in more detail below.

3.8.4 Reducing SA Migrations

Finding good mappings of VMs to PMs in virtual machine scheduling is only part of the problem. Once a better assignment of VMs to PMs is found, the problem of migrating the VMs to their new locations remains. VM migration comes at a cost of both network bandwidth and CPU. It is, therefore, prudent to minimize the number of migrations necessary to move to a better schedule. However, limiting the number of moves (migrations) actually works against the spirit of SA, in which many configurations are examined and often accepted in an effort to find a global optimum or good local minima.

Fortunately for us, finding the globally optimal assignment is not always necessary or even desirable. Looking back at Figure 3.4, if we imagine that distance along the x-axis is proportional to the number of migrations necessary to move between states, the migration overhead of moving from "current" to the globally optimal solution at S_3 may be far too expensive. There, the S_2 mapping may be more appealing due to its proximity to "current" and the fact that it is only slightly more costly than the assignment at S_3.

So, what we are really striving for in FMU is a solver which finds good schedules near our current schedule. For SA, we simply want to ensure that it is given sufficient opportunity to climb out of nearby "shallow" local minima to look for nearby "deep" local minima. If SA falls into a deep nearby local minima, we have probably found an acceptable new schedule.

The $neighbor$ function can play a pivotal role in quickly finding nearby deep

local minima. As mentioned above, the $neighbor$ function can be biased toward making good moves. Although this is not good for SA in general, it coincides with our goal of finding good nearby solutions, as opposed to searching the entire solution space for a global minima. This bias can be tuned as the simulation runs to allow for more aggressive searches (i.e., less bias towards good moves) when stuck in a local minima with unacceptably high cost.

Our neighbor function takes a parameter which specifies how often a good move should be made. If a good move should be made, our neighbor function finds the set of most hungry VMs consisting of the hungry VM for each resource receiving the least of that resource and randomly migrates them. Otherwise, our neighbor function simply migrates a randomly chosen VM to a randomly chosen PM. A feasibility check is performed for each suggested migration.

Notice that randomly migrating the most hungry VMs does not necessarily result in good moves. Nonetheless, the possibility of good moves is certainly higher than that of a purely random move.

Another possibility for reducing migration count is to roll the cost of migration back into our cost function to penalize solutions requiring large numbers of migrations. Unfortunately, this approach has two problems. First, adding migration cost back into the cost function changes our solution landscape as we search for a good assignment. It is not clear how this change affects convergence aspects of SA. Such a change would certainly make it much harder for SA to move away from poor local minima, limiting the advantage of this approach.

A second problem with charging our cost function with migration cost is that it requires maintaining how many migrations are required to get from one schedule to another. For example, imagine that SA moved a VM multiple times in its effort to find a better schedule. It would be likely that we could simply move that VM from its initial position to its final position, skipping all the intermediate migrations performed by SA during its solution search. In addition, it is not always as trivial as removing intermediate migrations from the path. As discussed next, problem constraints may

preclude migrating VMs directly to their destination. Having to keep track of how many migrations are necessary between two assignments adds additional computation cost to our cost function evaluation (which is already slightly expensive).

What is more, its not clear how the cost of migration should be charged to our cost function. Should it be a constant charge or depend upon link speeds between nodes, CPU speeds, or something else? For these reasons, we do not use this approach to limiting SA migration counts.

3.8.5 Migration Paths

Another issue we must consider is moving from an old to a new schedule. Moving between schedules cannot always be done in arbitrary order. Migrations must not violate constraints such as physical memory limits or resource allocation guarantees. As a simple example, imagine that all PMs in Figure 3.2 b have ten units of memory and each VM there consumes five units of memory. Then, it would not be possible to swap VM2 and VM3 to arrive at the more fair schedule in Figure 3.2 d since this would involve first migrating VM2 to PM2, which would temporarily require 15 units of memory. In this case, hibernating VM2 and restoring it on PM2 after VM3 has been migrated to PM1 is one option for getting around this limitation. If there were a third PM in the system, another option would be for VM2 to be migrated there, then to PM2 after VM3 has been migrated to PM1.

Fortunately, our $neighbor$ function does not move to assignments which violate problem constraints since those assignments do not belong to the set of feasible solutions. However, the shortest path (i.e., the one requiring the fewest migrations) to a new assignment is not always the same as the order of migrations taken by the SA heuristic. As an optimization, we first check to see if a simple ordering which migrates VMs directly to their new destination does not violate any problem constraints. If a violation is found, the path as taken by SA is used. For simplicity, optimizing this to check for additional shortened paths was not done. In practice, we found constraint violations in our optimized path were rare.

3.8.6 Allocation Prediction

Notice that SA requires cost, and hence Equations 3.1 and 3.2, to be calculated for each new schedule under consideration. This, in turn, requires that we know how resources will be shared on a PM given a list of local VMs and their resource desires. Thus, the sharing policy of the underlying VMM must be properly modeled.

In our experiments, resources are assumed to be fairly shared in a work-conserving fashion. The Xen Credit scheduler is assumed with default parameters for CPU, and networking resources are assumed to be fairly shared between all VMs. Neither of these assumptions are required. If properly modeled, it is possible to support more sophisticated sharing policies such as relative weights and maximum threshold values for CPU, or network traffic shaping.

For a fair-share, work-conserving scheduler, a simple algorithm determines the resource "hungry level", hl, for a set of VMs running on a given PM. Any VMs desiring more of the resource than hl are given hl units and placed in the set of hungry VMs for that resource for that PM. All others VMs are given their desire of the resource and placed in the set of satisfied VMs for that resource. The hungry level determination algorithm is as follows:

```
get_hungry_level(capacity, desires, max_alloc)
    if max_alloc < 0
        max_alloc = capacity
    sort(desires)
    hl = minimum(capacity/length(desires), max_alloc)
    # get smallest desire
    desire = desires.pop()
    while desire and desire < hl
        capacity-= desire
        hl = minimum(capacity/length(desires), max_alloc)
        desire = desires.pop()
    return hl
```

where,

$capacity$ - total amount of resource available on PM

$desires$ - list of all VM desires for the resource

$length()$ - function returning the length of the desires list

max_alloc - maximum number of units of resource which can be consumed by a virtual resource

$minimum()$ - function returning the minimum of two values

$pop()$ - member function returning the list head

$sort()$ - sorting function which sorts desires in ascending order

The max_alloc variable merits some explanation. This is the maximum amount of a resource which can be consumed by a VM virtual resource. For example, with multiple CPUs, a virtual CPU cannot consume more than the capacity of a single CPU. Imagine a four processor PM with capacity of 10 units each. A single virtual CPU would not be able to consume more than 10 CPU units, even though the CPU capacity of the PM is 40 units.

Each time a VM is moved from one PM to another, the hungry level, hl, $hungry$ and $satisfied$ sets, and allocations for each resource for the new schedule must be determined using the above algorithm. Once these are established, the cost for the new schedule can be computed and tested to see if the new schedule should become the current schedule. This somewhat expensive cost function calculation is acceptable for our problem. We discuss this in more detail below.

3.8.7 Resource Dependencies

One difficulty in determining resource allocation is dependencies between resources. For example, in Xen, network traffic to and from a guest VM induces a significant CPU load in both that domain,[9], as well as, Domain 0.[10] So, when a VM is migrated to a location where it can receive more network bandwidth, its CPU desire will increase accordingly. Likewise, if a VM with heavy network traffic is migrated to a location where it will receive less CPU than it needs to maintain its bandwidth, its network load will also decrease.

[9] In Xen, a VM instance is referred to as a "domain".

[10] Domain 0 is a privileged VM in Xen through which all other VMs and their virtual devices are managed. We assume that handling of network traffic has not been delegated to another domain.

Understanding this dependency and representing it mathematically has proven challenging.[11] This is an area warranting further study. Note, however, that our solution strategy applies equally well once this relationship is fully understood since the cost function of our SA solver can still be evaluated.

3.9 A Greedy Approach

Recall that one problem encountered in our application of SA to FMU is the large number of moves (i.e., migrations) incurred by the annealing process. We mentioned that methods for reducing high migration counts often work against the spirit of SA by precluding the solver from fully exploring its solution space. Nonetheless, experiments using SA with a reasonable threshold, $cost\_thr$, and a $neighbor$ function biased toward good moves often yielded very good schedules for FMU with far fewer migrations.

The Greedy Best Move (GBM) solver evolved from our experiments using the adapted SA solver above. It was discovered that turning up the periodicity of biased moves often resulted in good solutions with very few migrations. Taking this to extreme, we conceived of the GBM solver which always tries to make "smart" moves. The result is a solver which often finds good nearby schedules, but cannot climb out of bad local minima.

3.9.1 The GBM Heuristic

Under GBM, the best moves for each VM are determined and kept in a list sorted by cost improvement. Cost for GBM uses the same cost function as that for SA. For each VM, this list contains the best destination PM for that VM at its head. Each list is constructed by evaluating the value of the cost function with the corresponding VM at every other PM, while keeping all other assignments static. To illustrate, a system with ten VMs and five PMs would create ten lists (one for each VM) of four tuples each.

[11] This is at least for Xen, where strange interactions between CPU usage and networking performance has been observed.

Each tuple would consist of two components: i) the destination PM and ii) the change in cost, ΔC, if the VM were migrated to that PM. These lists are sorted by the ΔC component of each tuple. Using these move lists, the GBM heuristic is as follows:

1. for each VM, create a smaller, candidate move list from its move list consisting of moves within a prescribed fraction of its best move.
2. for each VM, randomly choose a move from its candidate move list
3. insertion sort VMs in ascending order of the ΔC value of their selected move into a global moves list
4. for each VM in the global moves list, check to see if its selected move still reduces cost. If so, migrate VM to its destination.

Programatically, creating the moves list and steps 1-3 are all done in a single loop to avoid storage overhead of keeping these lists for each VM.

The point of steps 1 and 2 in the GBM heuristic is to avoid having all VMs choose a single PM to which to migrate. It is easy to imagine alternative methods for avoiding an overly attractive PM. Passing the best N candidates for each VM to step 4 and checking them in turn may be prudent, but comes at the cost of additional checks for each VM. Another option might be to remove PMs from all VM move lists after they have been selected as a destination.

In step 4, we simply walk down the global moves list, migrating VMs if their move still reduces cost. A previous migration can easily invalidate a VM's selected move. For example, a global moves list may have two hungry VMs migrating to the same PM when it should only receive a single hungry VM. This can happen since we are only looking at single moves at a time, as opposed to move combinations, when constructing the VM move lists. This approach, however, is far more efficient than finding the globally optimal move, assuming it will be made, finding the second best globally optimal move, assuming it will be made, and so on.

We present experimental results for the GBM heuristic in Chapter 4.

3.10 A Hybrid Approach?

The main drawback of using the SA heuristic is that it incurs high migration counts. The benefit of this approach is its ability to escape bad local minima to more thoroughly search the solution space for good assignments. On the other hand, a shortcoming of our GBM heuristic is that it lacks the ability to escape poor local minima, whereas its advantage is that it often finds acceptable assignments with a much smaller migration overhead.

As a result, another approach is to apply a hybrid heuristic to FMU which uses GBM until its solutions are no longer suitable, then switches to our more aggressive SA solver to move to a better location in the solution space. After finding a better assignment using SA, the hybrid solver returns to using the GBM solver.

Though this hybrid approach seems to be a reasonable balance between finding good schedules and high migration counts, time constraints prohibited us from exploring its behavior in this dissertation.

3.11 Conclusions

In this chapter we defined our virtual machine scheduling problem and distinguished it from other types of process, application, and VM scheduling problems. We seek scheduling solutions to best-effort VM scheduling where VMs are not guaranteed quantities of resources and VMs are not excluded from running based upon system loads and resource availability. We introduced our canonical VM scheduling problem in this framework called Fair Maximum Utilization (FMU). FMU seeks to maximize overall cluster resource usage in a fair manner. We put forward several algorithms for our virtual machine scheduling problem of various levels of sophistication. As opposed to our more elaborate schedulers, our basic approaches make no attempt to interpret monitoring data to classify virtual machines as hungry or satisfied. In addition, these basic approaches do not try to determine how VMs will behave on a physical node after a VM migration to or from that node.

In the following chapter, we evaluate the scheduling heuristics presented here. One question we seek to answer is whether basic algorithms are (or can be) as effective as more sophisticated heuristics. An important goal in our evaluation is to show which of our heuristics, in general, are practical approaches to enforcing operational goals of a virtual cluster installation.

Chapter 4

Scheduling Evaluation

We now focus on evaluating our proposed scheduling approaches. To determine how well our schedulers perform, we focus on: i) quality of the scheduler's solutions (i.e., utilization and fairness), ii) scheduler agility (i.e., how quickly does it arrive at a better schedule), and iii) the overhead of finding and moving to better schedules.

In evaluating our schedulers, it is critical to understand the experimental environment and how our schedulers behave in them. Having several items to specify in an experiment (e.g., VM and PM counts, initial placement, initial load, load characteristics, etc.), it easy to inadvertently favor one scheduler over another. Though we cannot always provide "a level playing field" for each scheduler in our experiments, we try to remove as many advantages as possible, or clearly state when one scheduler has an advantage over another in a given environment. Our ability to game an experiment in favor of one scheduler or another makes it difficult to provide solid quantitative results of one scheduler's performance versus another. For example, stating that scheduler X is 10 percent better than scheduler Y in regard to fairness carries little weight since any tweak to the experimental environment almost surely changes this number (sometimes substantially).

In addition, some experimental setups simply do not lend themselves to substantial improvement. For example, an experiment which begins in an optimal state without significant changes in loads would not cast our schedulers in a favorable light.

On the other hand, one must question the validity of an experiment which begins in the most unfavorable configuration, since such would be uncommon in practice. For these reasons and those above, we often discuss the quality of our scheduler's assignments and their suitability to a particular environment (e.g., "good in a dynamic environment"). Though we also give quantitative results, the reader is encouraged to consider them in context.

4.1 Methodology

We study the behavior of our BCPU, BCPU2, SA, and GBM schedulers under the following virtual cluster scenarios:

Hot spot incident: Here, several VMs on a small set of nodes suddenly increase their CPU demands.

Pseudo general purpose cluster workload: Here, virtual cluster workloads are generated from statistical properties of observed general purpose cluster workloads.

Before presenting experimental results, we first discuss a few details.

Testbed: Our testbed consists of 20 Dell PowerEdge R200 servers equipped with dual Intel Xeon X3210 CPUs running at 2.13GHz. These are dual core processors, so each PM has a total of four CPU cores. Each machine has a total of 4GB of main memory and dual onboard 1Gbps network interfaces. Only one of the two network interfaces is used in our experiments.

We run Xen 3.1.0 with Linux 2.6.18 kernels for both Domain 0 and guest domains. Each guest domain runs with 128MB of main memory. All VMs run NFS-root (including Domain 0 VMs), so no local hard drive is ever accessed.

CPU load reporting: In our experiments, we report CPU as the number of cycles per second averaged over the previous minute rather than a load percentage. Using cycles rather than percentage is necessary since we may be operating in a heterogeneous

environment where some machines have faster processors than others. Though this is not a perfect measure since cycles are not exactly equivalent across different processor models, it is the best measure we have for comparing a VMs allocation on different machines. Certainly, a relationship should be established to compare cycles between very different architectures (e.g., between Xeons and Opterons). We do not consider migration between machines with vastly different architectures here.

CPU loader application: To induce a desired CPU load in our experiments, we have written a CPU loader application. Writing an application to consume a specified number of cycles per second is not a trivial task. Our CPU loader application currently loops over generating 1000 random numbers, checking to see how much system and user time have been consumed over the previous second using the `getrusage` system call, and sleeping every 0.05 seconds for an adjusted number of milliseconds to reach the target usage. Sleep time must be adjusted to mitigate interactions between this application, the imperfect Xen scheduler, and VM migration — all of which cause our application to occasionally miss its target load. Our loader calibrates itself every two seconds to ensure it is within five percent of its target load. If not, our application adjusts its load, causing the sleeping time to be increased or decreased as necessary. As a result of this design, our loader may be off by as much as five percent.

Scheduling granularity: As mentioned above, our schedulers rely upon loads averaged over one minute intervals. So, reported resource usage for a VM is only valid one minute after a migration. The same holds for the source and destination PMs involved in the migration. Therefore, scheduling can only be done at a minimum granularity of one minute plus our data collection interval. We collect monitoring data at 20 second intervals so our minimum scheduling interval is 80 seconds. Note that the duration over which load is averaged as well as data collection interval are tunable. We have selected these intervals as a balance between fidelity and monitoring/scheduling overhead.

4.2 Hot Spot Alleviation

The purpose of the hot spot alleviation experiments is to see how well our schedulers react to an imbalance in resource demands across a set of PMs. This also applies to the case where a subset of the running VMs suddenly increase their resource demands. Such is a common scenario in a multi-tenant environment where jobs, applications, or services are started on all of a tenants VMs simultaneously. This causes a subset of the VMs to suddenly become active and possibly require rescheduling.

To observe our scheduler's reaction under the worst case scenario, we limit our testbed to use only a single core on each PM and increase the load on a set of VMs residing on the same PM or PMs.

In these experiments there is no load on the VMs and no scheduler running at time zero. At time 20, each VM begins consuming 213MHz (ten percent) of the physical machines CPU. At 140 seconds, a number of VMs increase their CPU demands. These VMs are all packed onto the fewest number of nodes possible. VM desires do not change after 140 seconds for the remainder of the 600 second experiment. At 260 seconds, a scheduler is turned on and runs for the remainder of the experiment (shaded area in the figures). During this time, VMs are periodically migrated to better assignments found by the scheduler. Depending upon the scheduler, there may or may not be multiple migration rounds. For example, the SA scheduler is often very good at determining the optimal placement in a single iteration.

The above experiments are performed in two environments. The first consists of a small cluster of 18 VMs running on three PMs. The second is a larger cluster of 120 VMs running on 20 PMs.

In all hot spot alleviation experiments we study VM CPU allocations, overall system CPU utilization, and system fairness to hungry VMs. Our graphs show individual VM CPU allocations and overall system CPU utilization (right y-axis) over time. VM allocations are plotted with solid lines, Domain-0 loads with dashed lines, and overall CPU utilization with a dash-dot line. Shaded areas of the graphs indicate regions

where the schedulers were running.

4.2.1 Small Cluster (HSA-18:3)

In our first set of experiments, 18 VMs run on three physical machines. The small number of VMs and PMs involved allows for us to reason about our schedulers' behaviors and resulting assignments. We refer to these as the "HSA-18:3" experiments. Initial placement evenly spreads the VMs so that six VMs are running on each PM. Results are presented for the following scenarios:

- five VMs increase their desire to 427MHz[1]
- five VMs increase their desire to 747MHz[2]
- six VMs increase their desire to 747MHz

Although many additional scenarios were studied, these three provide good insight into the behavior of our heuristics. Results of those additional studies were consistent with results presented here.

Five VMs to twenty percent demand (HSA-18:3_5-20)

The first subset of the HSA-18:3 experiments have five VMs on a single PM increase their CPU demand to 427MHz (20 percent of a single PM's CPU capacity). Figures 4.1 through 4.4 depict results of this experiment for BCPU, BCPU2, SA, and GBM.

What is noteworthy is that this schedule does not impose more total CPU demand on the system than it is capable of supporting. Total CPU demand in this case is 4907MHz whereas the system can provide a total of 6400MHz (i.e., total demand is only 77 percent of total capacity). So, if properly scheduled, all five VMs which increase load should receive their desired 427MHz of CPU.

[1] Here, this is 20 percent of a PM's CPU capacity.
[2] Here, this is 35 percent of a PM's CPU capacity.

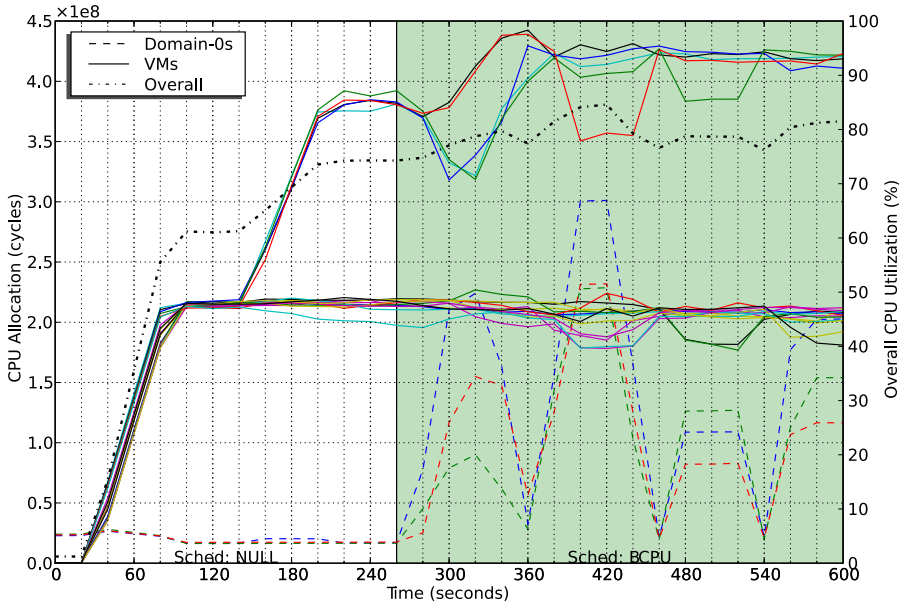

Figure 4.1 VM CPU allocations over time for BCPU applied to HSA-18:3_5-20. Here, five VMs increase their desire from 213MHz to 427MHz at time 140. The scheduler is activated at time 260. This scenario does benefit from the very simple BCPU scheduler.

From Figure 4.1, we can see that our system does benefit from the very simple BCPU scheduler. Within one minute of our scheduler activation, all hungry VMs begin receiving near their desired CPU. Recall that CPU allocation is averaged over a one-minute interval, so we can infer that VMs are receiving their desired CPU from the fact that the VM CPU allocations steadily climb for one minute following the minute after activating the scheduler. We can also infer that all VMs have reached their newly assigned PM in less than one minute from the Domain 0 loads, which fall steadily after one minute of migrating VMs (database logs confirm this). Notice, however, that our simple BCPU scheduler never really settles on a new assignment — even after finding a good assignment — and begins a new round of migration after one minute of monitoring VMs at their new locations. This behavior is due to BCPUs lack of categorizing VMs as hungry. Instead, BCPU assumes that VMs will only desire what they are currently receiving and tries to spread them such that the minimum available CPU of all PMs is maximized. As a result, slight variations in allocations to VMs desiring the same amount

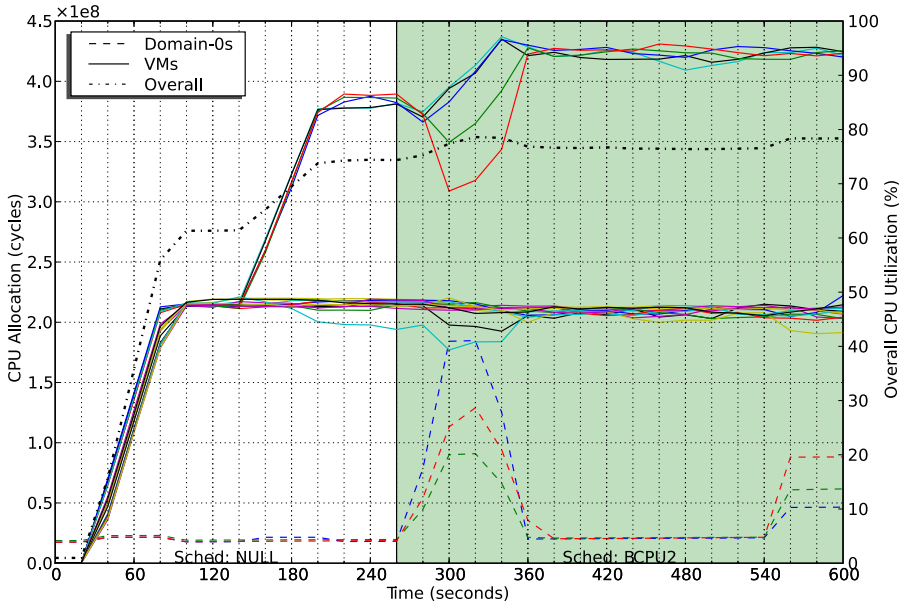

Figure 4.2 VM CPU allocations over time for BCPU2 applied to HSA-18:3_5-20. Here, five VMs increase their desire from 213MHz to 427MHz at time 140. The scheduler is activated at time 260. BCPU2 performs better than BCPU since it maintains VM assignment order.

of CPU cause the VM assignment order to be changed at each iteration. This ordering change results in needless migrations. As discussed above, we mitigate this issue in our heuristic with a migration resistance. However, this resistance does not completely eliminate such migrations.

We must also point out that this scenario (and all other hot spot alleviation experiments) provides an ideal initial placement for the BCPU scheduler, since, by getting more CPU than all other VMs, the increased desire VMs end up being placed first. So, it should be no surprise that BCPU finds a near optimal solution in its first iteration. Finding optimal solutions in its first iteration is not the case in general, since hungry VMs often receive less of a resource than satisfied VMs in an unbalanced system. This general case causes BCPU to incorrectly guess actual VM CPU desires, resulting in repeated adjustment as it discovers that the system is imbalanced. This effect is not seen in these experiments.

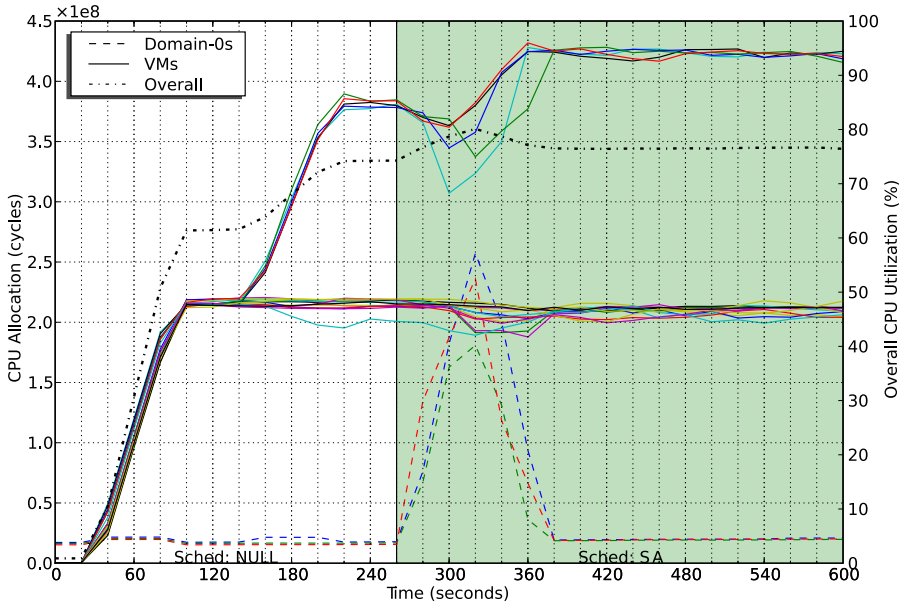

Figure 4.3 VM CPU allocations over time for SA applied to HSA-18:3_5-20. Here, five VMs increase their desire from 213MHz to 427MHz at time 140. The scheduler is activated at time 260. SA finds an optimal schedule in a single iteration, albeit with high migration count.

Migration cost is evident from the decrease in VM CPU allocations during the migration process as seen in Figure 4.1. Not evident from this figure is that individual PM loads are often 100 percent while sending and receiving VMs. As a result, we see cycles being stolen from both migrating and stationary VMs during migration periods. This loss is compounded for migrating VMs which must pause for a time during their final round of memory copying in addition to possible performance degradation due to enabling of shadow paging during migration in Xen [CFH+05]. To control migration impact, the number of simultaneous migrations to and from a PM, as well as network bandwidth available for migration, are tunable. We, however, did not turn that knob here. Migration cost will be discussed in more detail below.

Figure 4.2 shows results for BCPU2. Here, we see that BCPU2 performs better than BCPU since it maintains VM assignment order from iteration to iteration. Notice that BCPU2 actually finds an optimal assignment after a single iteration. In-

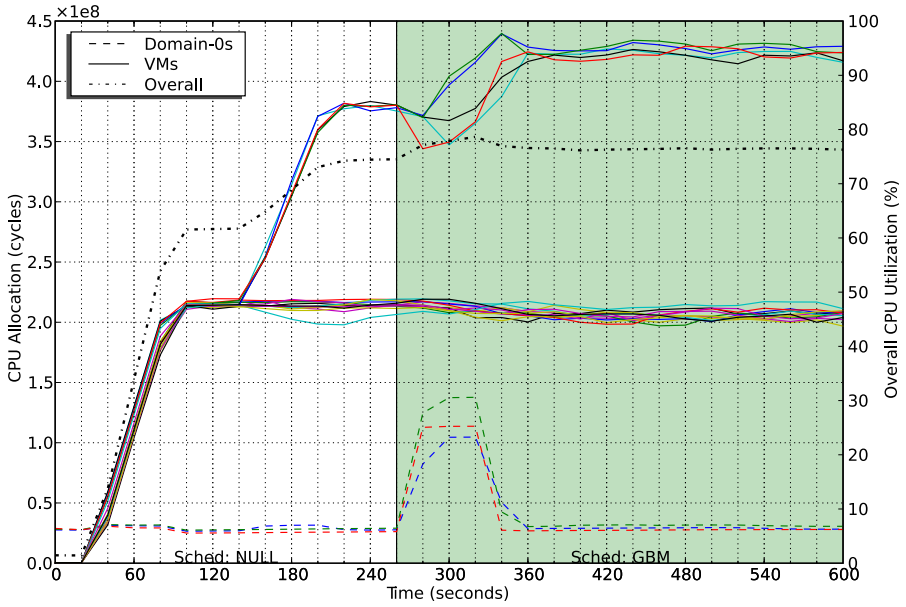

Figure 4.4 VM CPU allocations over time for GBM applied to HSA-18:3_5-20. Here, five VMs increase their desire from 213MHz to 427MHz at time 140. The scheduler is activated at time 260. GBM finds an optimal schedule in a single iteration with low migration count.

terestingly, BCPU2 does do an unnecessary (although not harmful) migration at 540 seconds. Overall, BCPU2 performed seven total migrations compared to BCPU's 33.

Figure 4.3 shows that simulated annealing performs quite well under this scenario. The SA scheduler settles on an optimal assignment after a single schedule iteration. In this case, the scheduler assigns two of the PMs three 213MHz and two 427MHz VMs and the third PM seven 213MHz VMs and one 427MHz VM. Notice also that the system is now utilized to our target of 77 percent. In addition, we see that VM migration produces loads in the Domain 0s slightly greater than that of BCPU2, although it does not migrate VMs beyond the first iteration. Our SA solver performed nine migrations in this particular experiment.

Figure 4.4 depicts results for GBM. Notice that GBM also found an optimal solution in a single scheduling iteration. Interestingly, this schedule places one 213MHz and two 427MHz VMs on the first PM, seven 213MHz and one 427MHz VMs on the

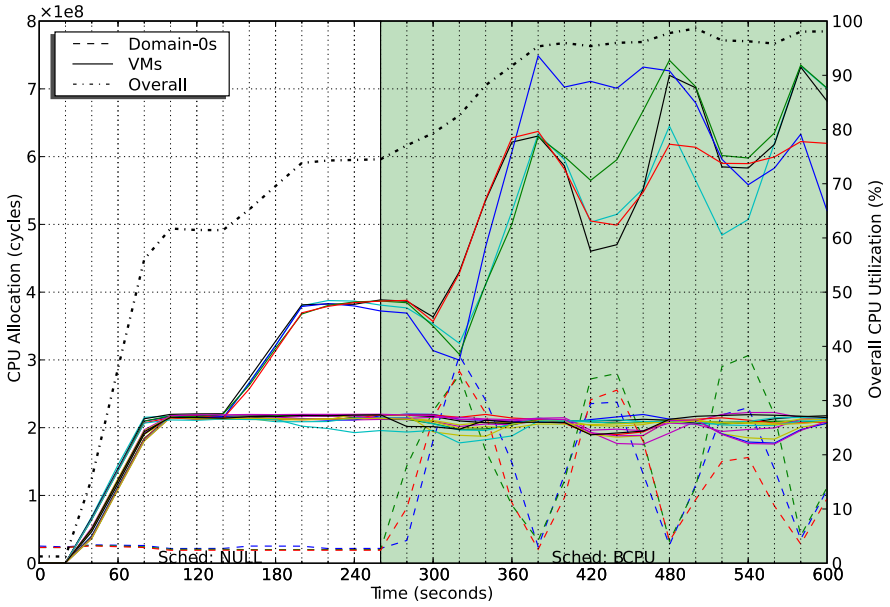

Figure 4.5 VM CPU allocations over time for BCPU applied to HSA-18:3_5-35. Here, five VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. Here, BCPU's high migration overhead robs CPU cycles from VMs.

second, and five 213MHz and two 427MHz VMs on the third. Clearly optimal schedules are not unique in this case. Finally, notice that the migration time and overhead are less than for SA, BCPU, or BCPU2. GBM performed only four migrations to reach an optimal schedule.

Five VMs to thirty-five percent demand (HSA-18:3_5-35)

We now have our five VMs increase their desire to 747MHz (35 percent of a single PM's CPU capacity). Under this scenario demand (6507MHz total) slightly exceeds our system's capacity (6400MHz total), yet there is no schedule which maximizes overall utilization that is perfectly fair to our VMs with increased CPU demand. Results of this experiment are plotted in Figures 4.5 through 4.8.

Figure 4.5 contains results of the BCPU scheduler applied to this scenario. Again, BCPU never settles on a particular assignment. Notice that, although overall

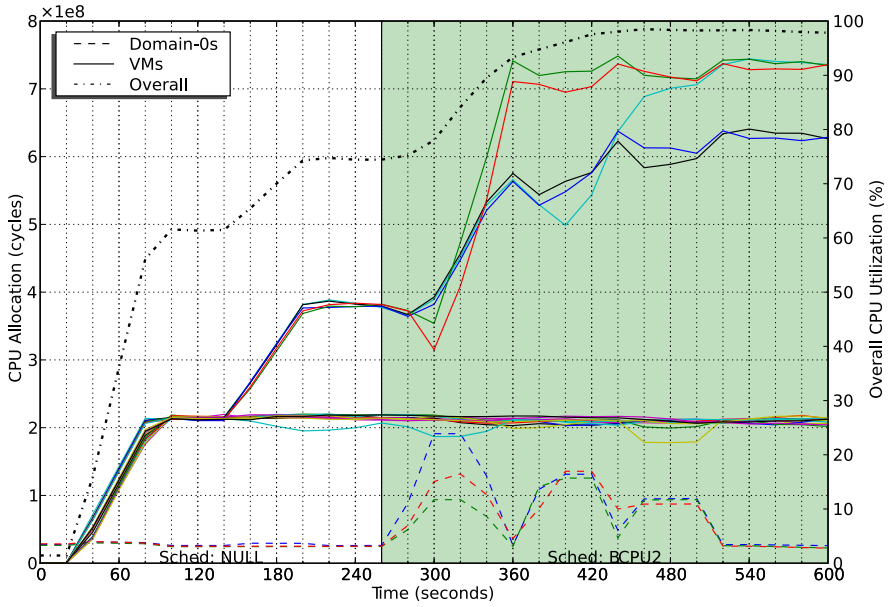

Figure 4.6 VM CPU allocations over time for BCPU2 applied to HSA-18:3_5-35. Here, five VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. BCPU2 eventually settles on a very good assignment.

CPU consumption is nearly 100 percent, our 747MHz-desire VMs receive approximately 600MHz on average due to excessive migration overhead. We can clearly see that the inefficiencies of BCPU are exacerbated at higher loads.

Results for BCPU2 are shown in Figure 4.6. These results are quite impressive for such a simple scheduler. Compared with BCPU, BCPU2 incurs much lower migration cost and actually looks to settle on a very good assignment after three rounds of scheduling. Overall, BCPU performed 49 migrations over the experiment duration whereas BCPU2 performed only 15. Clearly, the better packing of the BCPU scheduler does not compensate for the increased migration overhead in our two experiments thus far.

Applying our SA solver reveals the strength of this approach. Figure 4.7 shows results for SA applied to HSA-18:3_5-35. After a single scheduling round, overall utilization is maximized (100 percent) and 4 out of 5 747MHz VMs are receiving approximately 733MHz and the fifth approximately 665MHz. This assignment places two

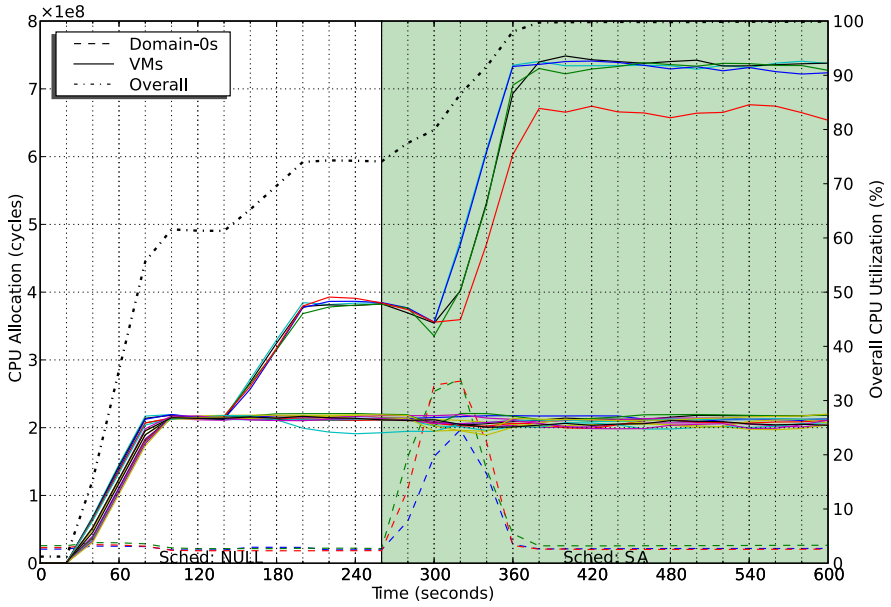

Figure 4.7 VM CPU allocations over time for SA applied to HSA-18:3_5-35. Here, five VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. SA finds an optimal schedule in one iteration in this scenario.

747MHz VMs and three 213MHz VMs on each of two PMs and one 747MHz VM and seven 213MHz VMs on the third PM. The observant reader may notice that this schedule should result in four 747MHz VMs receiving 747MHz, and one 747MHz VM receiving 640MHz. Recall, however, that our loader runs at up to five percent error. The actual numbers are within this error bound.

Another very good schedule places three 747MHz VMs on a single PM, two 747MHz VMs and three 213MHz VMs on another, and 10 213MHz VMs on the third PM. It is very unlikely, however, that our scheduler would have suggested this assignment. The reason is that 10 VMs, each receiving 10 percent of the available CPU on a single PM, would have been categorized as *hungry* VMs by the scheduler's hungry determination heuristics. As a result, there would be a substantial fairness cost incurred upon this schedule since 10 hungry VMs would receive near 213MHz while two receive near 747MHz and three receive near 640MHz.

Missing out on good schedules which perfectly pack (i.e., to 100 percent ca-

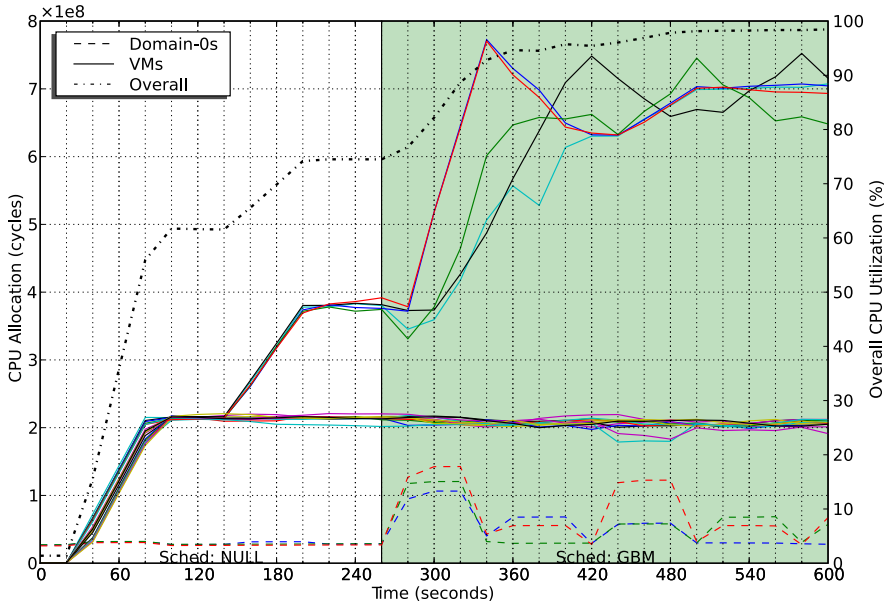

Figure 4.8 VM CPU allocations over time for GBM applied to HSA-18:3_5-35. Here, five VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. GBM found good schedules which it continued to refine, but did not find an optimal schedule within the 600 second experiment.

pacity) *satisfied* VMs onto a single node is a limitation of our SA and GBM approaches. Missing some good schedules is a result of the black-box approach we take to hungry VM detection which will always categorize a VM or set of VMs as hungry on a PM where a resource is fully utilized. For this, there is no good solution. Fortunately, schedules which perfectly pack *satisfied* VMs are quite rare.

Finally, Figure 4.8 shows results of the GBM scheduler applied to the HSA-18:3_5-35 scenario. In this case, GBM does not find a great schedule after its first scheduling iteration. It continues to make small corrections and refinements to its schedule round after round, increasing overall CPU utilization and fairness. Within the 600 second experiment, the system does not converge upon an assignment.[3] Nonetheless, GBM clearly outperforms the BCPU scheduler and is far better than doing no scheduling at all. Notice that the small corrections made each round do not have a large impact

[3]It eventually will, it just did not within 600 seconds in this experiment.

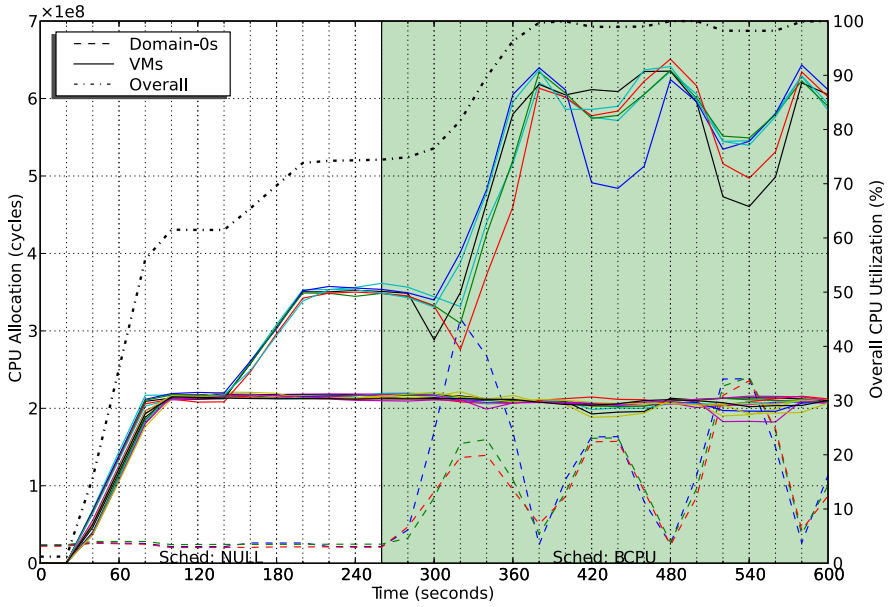

Figure 4.9 VM CPU allocations over time for BCPU applied to HSA-18:3_6-35. Here, six VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. BCPU continues its habit of needlessly migrating VMs after initially finding a good schedule.

upon VM allocation. The ability to make small corrections without large impact upon VM performance is a key feature that makes GBM desirable for the hybrid approach.

Note that GBM is not deterministic since there is randomization involved in the choice of which locally optimal PM to move a VM. Therefore, in our experiments, it was common for GBM to actually find an optimal solution within one round under this scenario. For this reason, we believe that a bit more work in making this scheduler deterministically choose the best location based upon previous VM destination choices would help GBM to converge more quickly.

Six VMs to thirty-five percent demand (HSA-18:3_6-35)

We now look at increasing the number of VMs with increased CPU desire to six. We only show the case of six VMs increasing their desire to 747MHz since BCPU2,

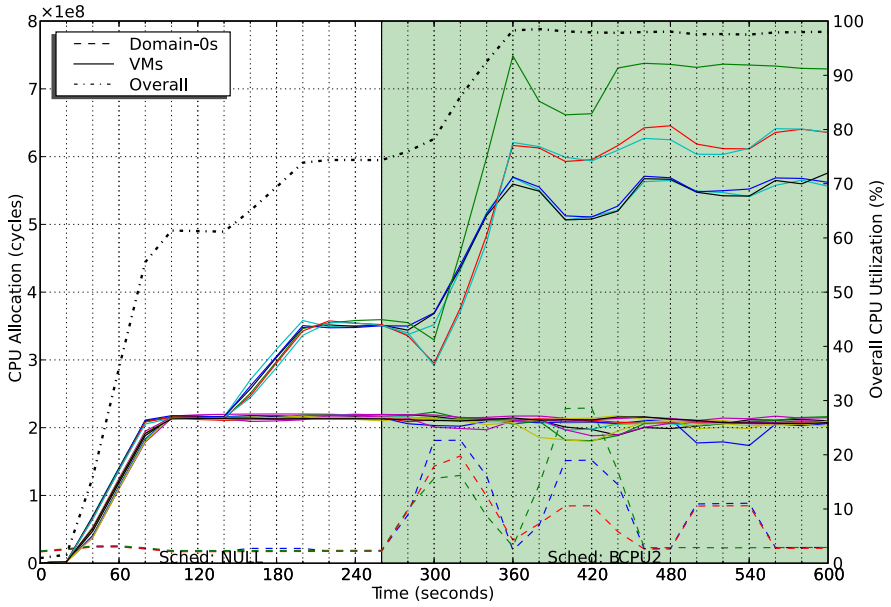

Figure 4.10 VM CPU allocations over time for BCPU2 applied to HSA-18:3_6-35. Here, six VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. This figure shows that BCPU2 begins having trouble with higher demand scenarios, eventually settling on an unfair schedule.

SA, and GBM all easily handle scenarios where the system is not fully utilized.[4] What is different about this setting is that there are multiple optimal solutions in which our 747MHz desire VMs each receive 640MHz (i.e., 30 percent). Figures 4.9 through 4.12 contain results for this scenario.

We see no surprises concerning the behavior of the BCPU scheduler in Figure 4.9. BCPU continues its habit of needlessly migrating VMs after a good schedule has been found. This behavior was observed in each of our experiments with BCPU. On the other hand, we see that our BCPU2 scheduler begins to have difficulty with the higher CPU demand scenario. This difficulty is a result of its poorer quality packing of VMs onto PMs. Recall that BCPU2 uses a two-approximation job scheduling heuristic. Though BCPU2 does seem to settle into an assignment by experiment end, it settles into a rather unfair one.

[4]Note that BCPU2 only handles a single resource, however.

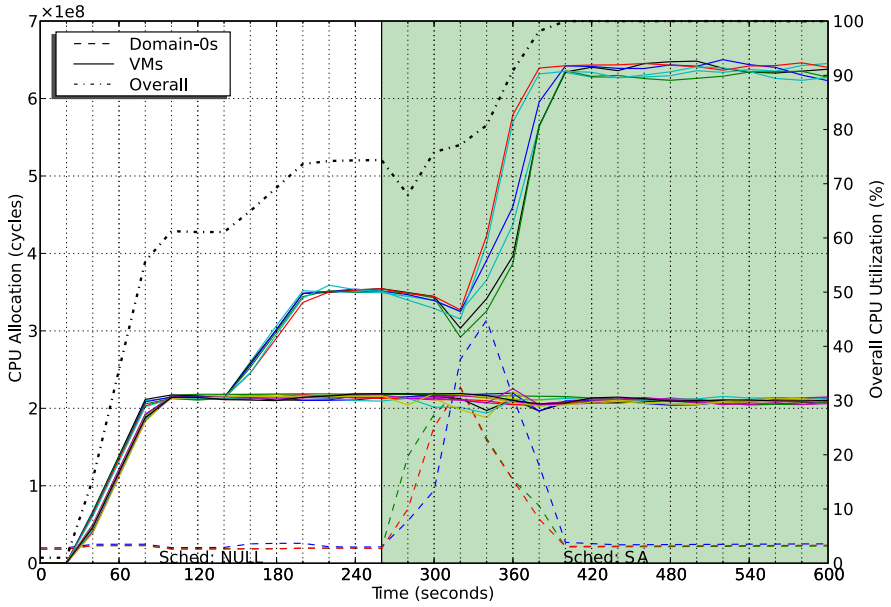

Figure 4.11 VM CPU allocations over time for SA applied to HSA-18:3_6-35. Here, six VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. Again, SA shows its strength of quickly finding an optimal schedule, and its weakness of requiring many migrations to reach the new schedule.

On the other hand, both SA and GBM quickly find optimal solutions (all hungry VMs receiving 640MHz of CPU). Our SA scheduler is able to find an optimal schedule in one iteration, whereas GBM takes two. Interestingly, they settle on two very different solutions. SA finds an optimal schedule in which two 747MHz VMs and four 213MHz VMs are assigned to each PM. GBM, however, manages to find an interesting solution in which three 747MHz and one 213MHz VMs are placed on one PM, two 747MHz and four 213MHz VMs on another, and one 747MHz and seven 213 MHz VMs on the third. Again, the VM receiving approximately 650MHz in the GBM solution is within the five percent error of our CPU loader application.

Small cluster hot spot alleviation conclusions

To be thorough, we ran many other combinations of HSA-18:3 experiments. The experiments presented above, however, provide good coverage of the behavior

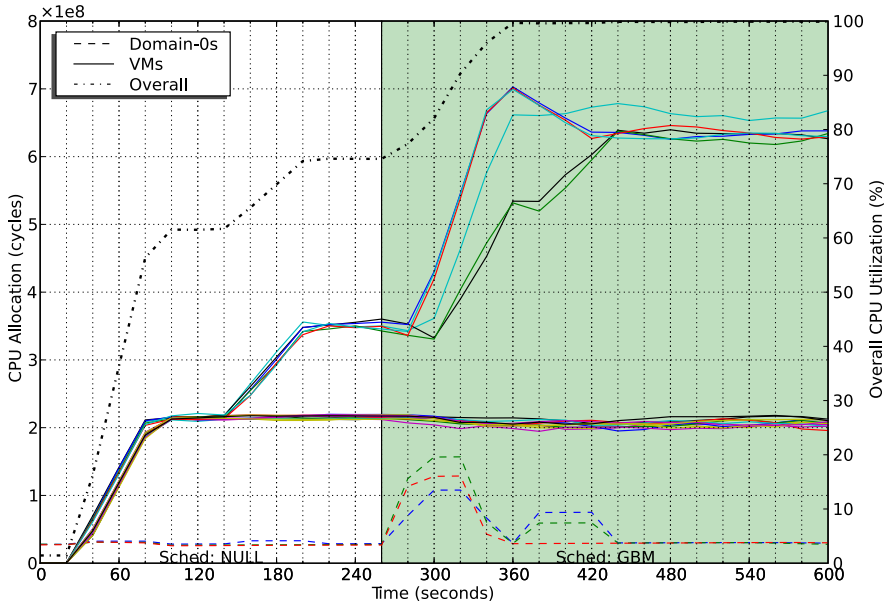

Figure 4.12 VM CPU allocations over time for GBM applied to HSA-18:3_6-35. Here, six VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. GBM begins to separate itself as a good, general purpose scheduler by quickly finding an optimal solution requiring few migrations.

of our schedulers, so presenting additional experimental results would be redundant. Nonetheless, observations from the additional experiments did highlight a few high level characteristics not necessarily evident from the previous results.

First, our BCPU scheduler continuously migrates VMs, even after finding good schedules. However, BCPU does often treat hungry VMs fairly at the cost of stealing CPU cycles due to migration overhead.

The BCPU2 scheduler tends to split the hungry VMs into distinct sets. Though its schedules often maximize CPU utilization, they often suffer in fairness. In addition, as total desire increases further and further beyond system capacity, BCPU2 takes longer to converge on an assignment. These behaviors are also adversely affected by increased standard deviation in VM resource desires.

Our SA and GBM schedulers often quickly converge to optimal solutions. Even a very restricted SA solver often manages to find very good or optimal solutions

in a single iteration. Occasionally GBM takes a few iterations to converge, and in rare instances, slowly converges over several iterations.

Table 4.1 presents total migration counts of our schedulers for each scenario. Notice both our BCPU schedulers tend to suffer from increased migration counts at higher loads. Also, GBM is typically more efficient in terms of migration costs than SA. In cases where GBM results in higher migration counts than SA, those costs are amortized over longer time intervals. In other words, unlike SA, GBM does not typically suggest large numbers of migrations at each scheduling interval.

Table 4.1 Total migration counts for small cluster hot spot alleviation experiments.

Experiment	**Scheduler**	**Migration Count**
5 VMs to 427MHz (HSA-18:3_5-20)	BCPU	33
	BCPU2	7
	SA	9
	GBM	4
5 VMs to 747MHz (HSA-18:3_5-35)	BCPU	49
	BCPU2	15
	SA	9
	GBM	10
6 VMs to 747MHz (HSA-18:3_6-35)	BCPU	33
	BCPU2	16
	SA	10
	GBM	5

Recall our VMs each have 128MB of memory. We have seen that the cost of migration is CPU load in Domain 0 which acts to steal CPU away from hungry VMs. In addition, VM performance is impacted during migration due to shadow paging overhead in Xen. Since increased VM memory size will increase migration duration, it will act to increase total migration cost, an important consideration when choosing a scheduler.

Finally, keep in mind that our BCPU and BCPU2 schedulers only consider a single resource. Since that is all we are considering in these experiments, these two simple schedulers perform quite well relative to our more complicated SA and GBM schedulers. Due to other resources dependencies upon CPU, it remains to be seen whether

considering other resources greatly affects schedule quality.

4.2.2 Large Cluster (HSA-120:20)

We now verify that scheduler behaviors observed in our small scale experiments hold at a larger scale. These large cluster hot spot alleviation experiments, referred to as HSA-120:20, ran 120 VMs on 20 physical machines. The HSA-120:20 experiments were valuable because they emphasized characteristics observed in the small scale experiments.

Again, initial placement evenly spreads the VMs so that six were running on each PM at time zero. We limit the results presented here to a single scenario in which 40 VMs increase their desires to 747MHz (35 percent) at 260 seconds. Other experiments support the behaviors reported here.

Forty VMs to thirty-five percent demand (HSA-120:20_40-35)

Figures 4.13 and 4.14 contain results for BCPU and BCPU2. Notice that BCPU clearly outperforms BCPU2 in this scenario in terms of utilization and fairness. Recall that BCPU2's advantage was a reduction in migrations by fixing assignment order. However, this advantage shrinks when more resource demand is placed upon our system than it is able to provide. This experiment clearly highlights this weakness of BCPU2.

Once again, from Figure 4.15, SA is able to find an optimal solution in a single scheduling step. However, the larger system did delay finding the optimal schedule by 20 seconds, as VMs do not begin migrating until time 280 seconds. This step involved migrating 110 of our 120 VMs. These migrations took a total of two minutes and 20 seconds. This heuristic clearly does not appear to be a good scheduler for a very dynamic system.

Finally, GBM's behavior of making small corrections at each iteration holds here. This behavior can be seen in Figure 4.16 where GBM continues adjusting its schedule for the remainder of the experiment. Notice that GBM does not manage to find

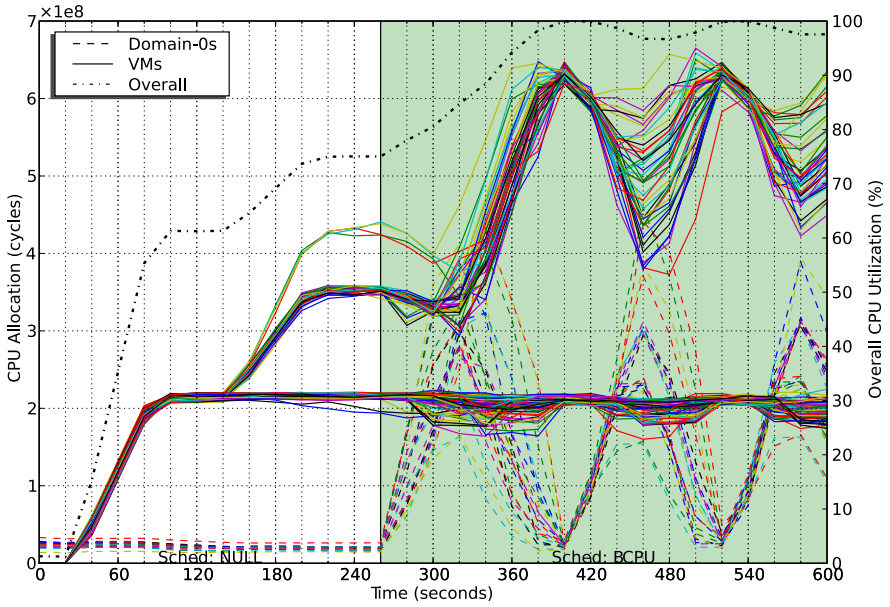

Figure 4.13 VM CPU allocations over time for BCPU applied to HSA-120:20_40-35. Here, 40 VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. BCPU continues its habit of migrating needlessly in this large cluster setting.

either fair or maximum utilization schedules. It does continue refining and improving its schedule however.

Table 4.2 provides migration counts for our schedulers under the HSA-120:20_40-35 experiment. Clearly, BCPU and BCPU2 are extremely expensive schedulers for maintaining fairness and high utilization in larger systems with high demand. It seems unlikely such simple schedulers would be of great value in a general setting. Of course, our failure to find a simple scheduler which outperforms our SA and GBM schedulers does not mean such a scheduler does not exist.

The tendency of SA to make large improvements and GBM to make small improvements to a schedule makes them ideal partners for a hybrid approach. Such an approach would use SA to make large schedule corrections to high cost schedules, then switch to GBM to keep a system running near optimal through small schedule corrections. Once GBM is no longer able to find schedules below a threshold cost, SA

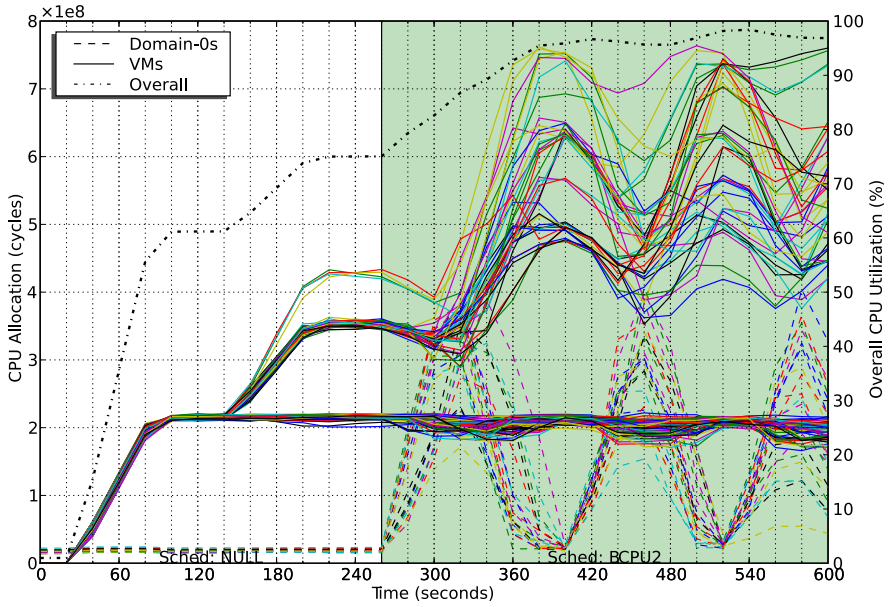

Figure 4.14 VM CPU allocations over time for BCPU2 applied to HSA-120:20_40-35. Here, 40 VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. Again, BCPU2 suffers when overall resource demand is high due to non-optimal packing.

could be used again.

4.3 General Purpose Cluster Workloads

We now experiment with our scheduling heuristics under a few synthetic, general purpose cluster workloads. Our workload generator is based upon statistical properties of cluster workload traces collected in [LGW04]. In particular, we use their results to generate Weibull random variates for our job durations and exponential random variates for our job inter-arrival times. These traces were collected over a year from five research clusters dedicated to parallel and distributed computing research. These clusters ranged in size from 32 to 72 nodes. Since such trace data for virtual cluster usage does not exist, we believe this a reasonable approximation of user behavior in some virtual cluster environments. To be thorough, we run our experiments for various values of

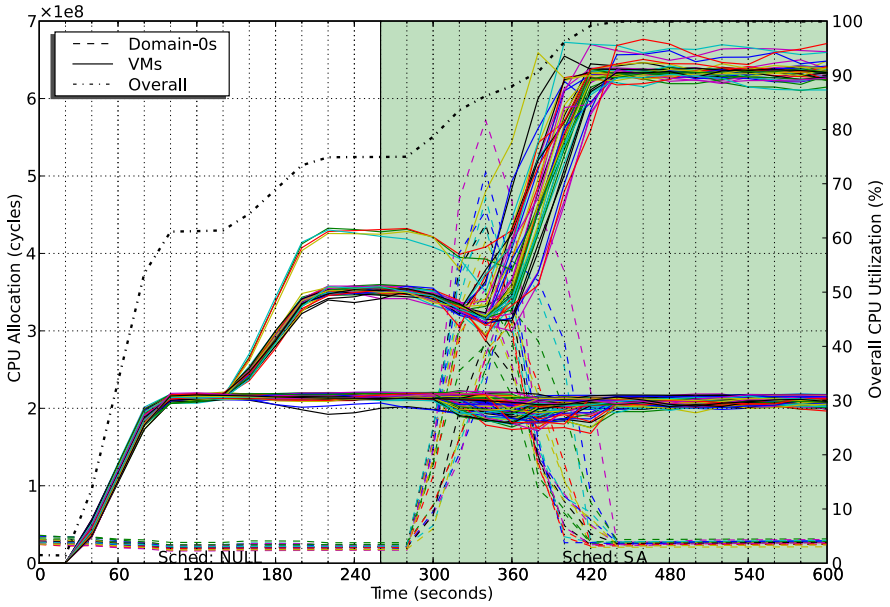

Figure 4.15 VM CPU allocations over time for SA applied to HSA-120:20_40-35. Here, 40 VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. SA again finds an optimal schedule in one iteration in this large cluster. Migration count, however, is quite excessive.

job duration and inter-arrival rate.

Unlike process scheduling where it is often assumed that a process will consume as much CPU as it is given, we randomly select a desired CPU cycle rate for each of our jobs. Certainly, it would be difficult to collect trace data on a process' desired CPU rate (we know of no such attempt). So, in the absence of real world data, we have opted to use a Weibull distribution from which we draw a job's desired CPU rates. For all experiments, we set the shape parameter of our CPU demand Weibull distribution at 0.53 and scale parameter at 21. The values drawn from this distribution are interpreted as percent of single CPU capacity of our physical nodes. We have set a minimum of eight percent[5] and a maximum of 100. These parameter values result in 45 percent of all jobs desiring eight percent (171MHz in our setting), 10 percent desiring 100 percent (2133MHz), and an average desire of 38 percent (811MHz).

[5]This minimum ensures we do not have too many idle clusters in our experiments. Perhaps we could massage distribution parameters a bit to avoid this, but we are already approximating an unknown distribution.

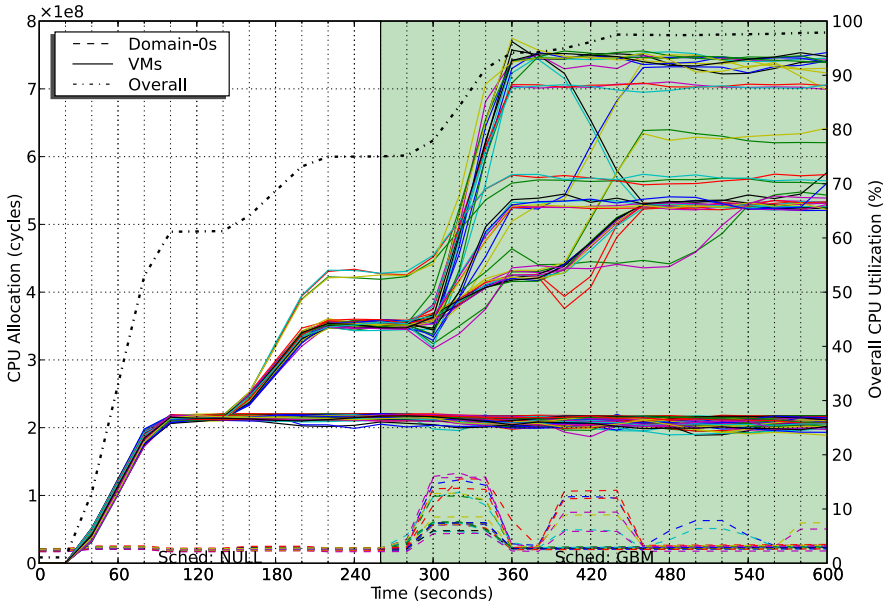

Figure 4.16 VM CPU allocations over time for GBM applied to HSA-120:20_40-35. Here, 40 VMs increase their desire from 213MHz to 747MHz at time 140. The scheduler is activated at time 260. GBM continues to refine its schedule, but doesn't find an optimal schedule withing 600 seconds. The GBM heuristic suffers from not maintaining VM moves as it generates its next schedule.

In each experiment, we use our 20 node cluster running 120 virtual machines for one hour. Again, VMs are initially evenly spread with six VMs per node. Job inter-arrival times are exponential variates (i.e., we consider job arrival to be a Poisson process), job durations drawn from a Weibull distribution, and job desired CPU cycle rates (discussed below) are drawn from a Weibull distribution. For all experiments we study VM CPU allocations, total VM allocation, overall system CPU utilization, and system fairness to hungry VMs.

To model multiple virtual clusters running on our physical cluster, we randomly divide our 120 virtual machines into clusters ranging in size from one to 18 VMs. For our experiments, this resulted in 12 different virtual clusters of size 9, 18, 17, 15, 12, 1, 8, 8, 3, 8, 8, and 13 VMs. Round robin initial placement of VMs is used so that no two VMs in a cluster are on the same node. When a job arrives, it is assigned to

Table 4.2 Total migration counts for large cluster hot spot alleviation experiment.

Scheduler	Migration Count
BCPU	307
BCPU2	282
SA	110
GBM	27

all VMs in a randomly selected virtual cluster. So, if our first virtual cluster above were randomly chosen to run a job, that job would run on all nine VMs in that virtual cluster. This behavior is in accordance with that observed in our Usher installation at UCSD. Our users typically setup a virtual cluster for a particular purpose and tend to start jobs simultaneously on all nodes in that cluster.

In these experiments, we refer to the case where no scheduler is in use as running a "NULL" scheduler. The NULL scheduler simply returns the current schedule as the next assignment. We introduce the NULL scheduler here since this experimental setup is extremely favorable to the case where no scheduler is used — especially for more dynamic workloads. Since all VMs in a cluster are initially placed round robin onto PMs, and jobs startup simultaneously on all nodes in a cluster, new jobs are already evenly spread across physical machines. In addition, jobs are randomly assigned to clusters, so we are nearly evenly spreading jobs across different clusters. Finally, we are experimenting in a homogeneous environment where all physical machines are identical. Since this scenario is so favorable to no scheduling, any scheduler doing better than the NULL scheduler is performing quite well.

A key motivation for running these experiments is to evaluate how well our schedulers perform at different levels of cluster activity. In other words, how dynamic can the job load be before our schedulers are unable to provide benefit? To this end, we run experiments for three levels of workload dynamism: low, moderate, and high. Each dynamism level uses a different value for average job inter-arrival times (by changing the rate parameter of our exponential distribution) and average job duration (by modifying

the job duration Weibull scale parameter). Our parameters for both inter-arrival rate and job duration are in accordance with [LGW04]. Values for all distribution parameters are specified in the respective results section.

In each section, we present results for each scheduler's time to find a new schedule in addition to time to migrate VMs to their new assignments. Recall that our schedulers base their migration decisions on one minute average utilizations. So, after VMs are migrated, one minute must pass before a new scheduling can occur. All heuristics were run on a Dell PowerEdge 1750 server with 2.8GHz Intel Xeon processor and 2GB of memory.

Note that our BCPU scheduler's behavior of excessive migrations resulted in instability in our testbed in the high dynamism general purpose workload scenario. Kernel and Xend errors were prevalent under this scheduler in the highly dynamic experiments. For this reason, we exclude results for our BCPU scheduler in this section. Needless to say, the results we were able to get did not bode well for the BCPU scheduler in this setting, so it would not be a good choice, regardless.

4.3.1 Low Dynamism

For our low dynamism general purpose cluster workload experiments, we draw job inter-arrival times from an exponential distribution with expected value of 120. So, jobs arrive every 120 seconds on average. Job durations were taken from a Weibull distribution with scale parameter of 240 and shape parameter of 0.46. This resulted in jobs with average duration of 567 seconds and median duration of 108 seconds. This may represent a cluster with a small number of users which regularly submit long running jobs or run services. We refer to these as the "GPCW-LD" experiments.

CPU allocations

Here we study individual VM CPU allocations, total VM CPU utilization, and overall system CPU utilization (includes Domain-0 CPU allocations) in an environment with low workload dynamism for each scheduler. Plotting both total VM allocation and

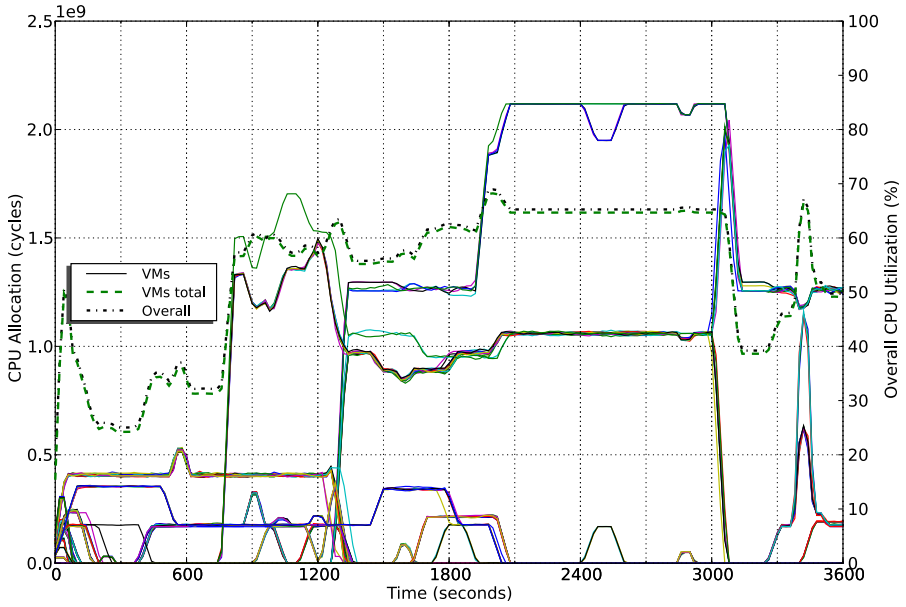

Figure 4.17 VM CPU allocations over time for NULL applied to GPCW-LD. The workload is not highly active during the one hour run.

overall CPU allocation enables us to see the CPU cost of excessive migrations.

Figure 4.17 contains plots of the above values for this scenario without a scheduler. We plot VM CPU allocations using solid lines (values on left y-axis), total VM allocation — as a percent of available CPU — with a dashed line (values on right y-axis), and overall CPU utilization with a dash-dot line (values on right y-axis).

Notice that there are relatively few events in this workload with jobs starting and stopping every few minutes. Only a few jobs last under a minute and appear as small bumps in the corresponding VM allocation plot. Since we are allocating jobs to clusters, notice that sets of allocation lines tend to overlap. Though difficult to see, this trend breaks occasionally where clusters overlap due to the round robin scheduling. This overlap is one source of unfairness for the NULL scheduler. We discuss fairness in more detail below.

Figure 4.18 contains VM CPU allocations for the BCPU2 scheduler applied to GPCW-LD. BCPU2 does improve overall utilization over the NULL scheduler. How-

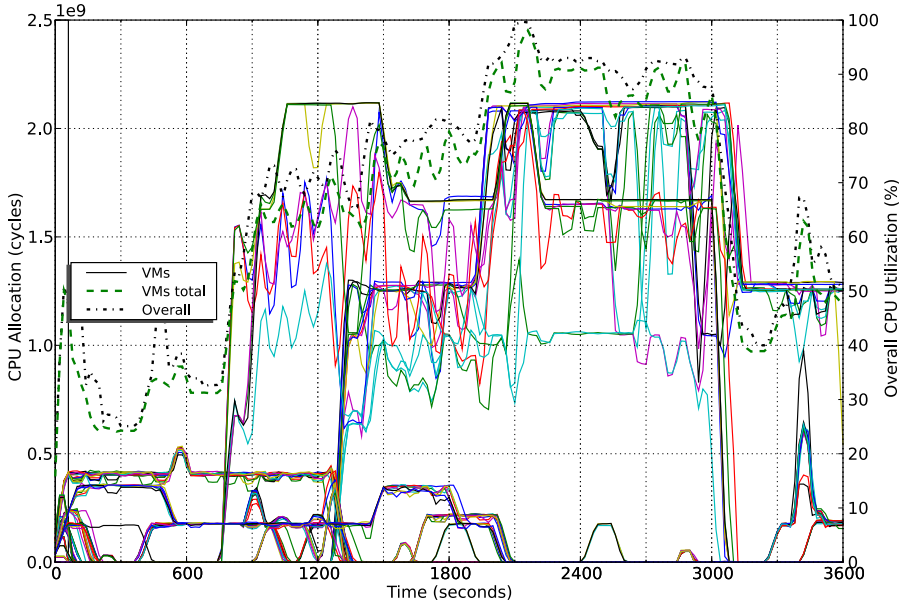

Figure 4.18 VM CPU allocations over time for BCPU2 applied to GPCW-LD. Though better than no scheduler, this scheduler does exhibit excessive migrations which could rob VMs of CPU cycles at higher loads.

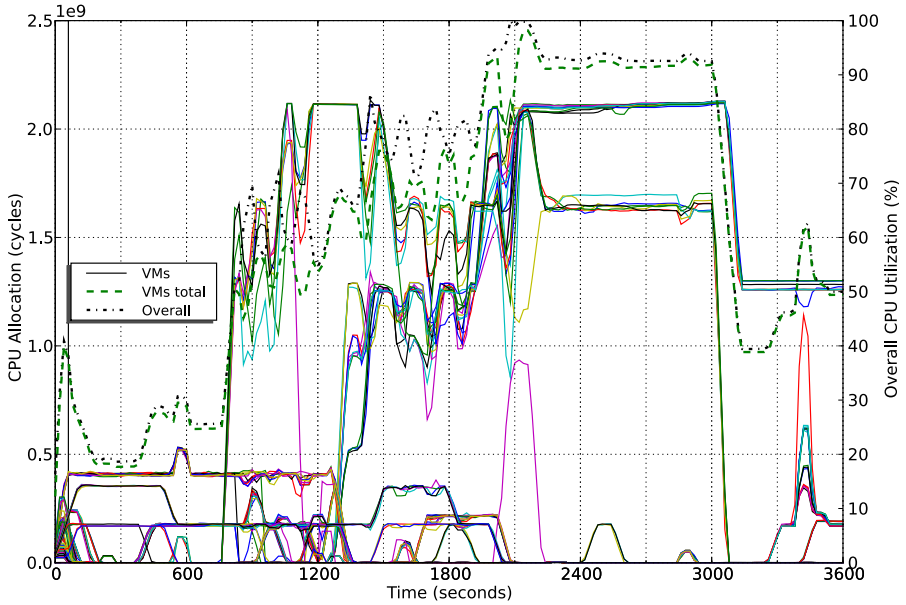

Figure 4.19 VM CPU allocations over time for SA applied to GPCW-LD. Though migration overhead is high, SA does an excellent job finding utilization improving assignments. SA is a viable scheduler in this low dynamism environment.

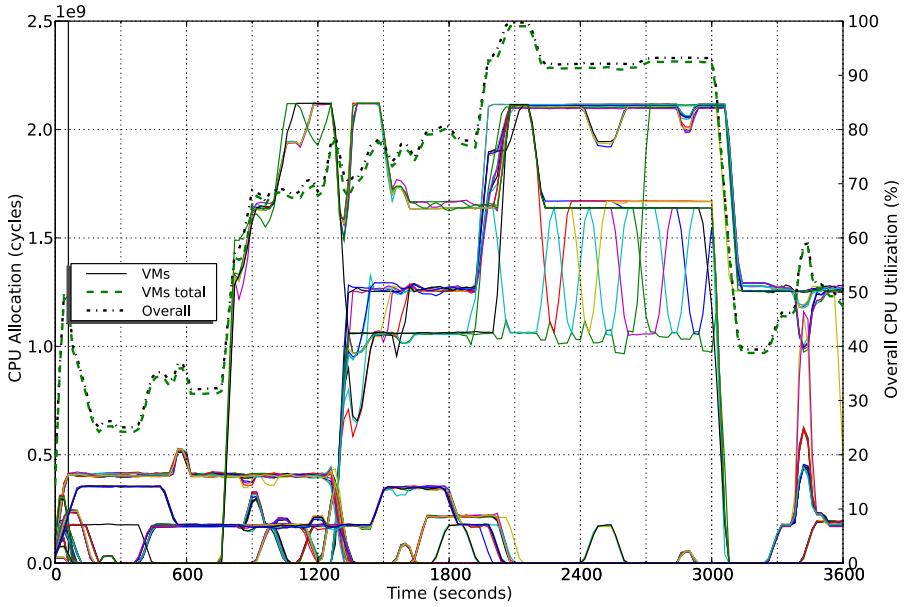

Figure 4.20 VM CPU allocations over time for GBM applied to GPCW-LD. GBM perfoms admirably in this environment, clearly outperforming the NULL scheduler and often outperforming SA. GBM isn't able to settle on a schedule during inactive periods as does SA.

ever, BCPU2's excessive migrations make VM allocations vary wildly and could rob VMs of CPU cycles at higher loads.

Table 4.3 contains costs of scheduling for all low dynamism general purpose cluster workload experiments. On average, BCPU2 took 35 ($7 + 28 = 35$) seconds to find and move to a new schedule. BCPU2 generated 36 migrations on average at each scheduling and a total of 1231 migrations in 34 scheduling rounds over the one hour simulation. In other words, BCPU2 continued rescheduling at every opportunity.

Figure 4.19 reveals that our SA solver performs quite well in this low dynamism scenario. Though migration counts are high, SA does settle on very good assignments during inactive periods.

On average, SA was able to find new schedules in 23 seconds in this scenario. On average, SA performed 109 migrations per schedule. This average migration count resulted in an average of 44 seconds to migrate VMs to their new locations with a max-

Table 4.3 Scheduling costs for low dynamism general purpose cluster workload experiments.

	BCPU2	**SA**	**GBM**
Avg schedule search time (sec)	7	23	19
Max schedule search time (sec)	18	34	27
Number of reschedules	34	8	20
Avg migration time (sec)	28	44	4
Max migration time (sec)	66	59	10
Avg migration count	36	109	4
Max migration count	78	118	20
Total migration count	1231	869	73

imum migration time of 59 seconds. In total, SA took 67 seconds ($23 + 44 = 67$) to find new schedules and migrate VMs to their new destinations. In this low dynamism environment, 67 seconds is an acceptable duration for arriving at a new schedule.

Our GBM scheduler also performs very well in this environment. Figure 4.20 contains CPU allocation plots for GBM applied to the GPCW-LD scenario. GBM clearly outperforms the NULL scheduler and often out performs SA. However, GBM isn't able to settle on a schedule during inactive periods as does SA. Interestingly, this is a result of VMs oscillating between hungry and satisfied as they migrate between nodes.

From Table 4.3 GBM finds new schedules in an average of 19 seconds and migrates VMs in an average of four seconds. This results in a 23 second average to find new schedules and migrate VMs. Surprisingly, GBM takes nearly the same amount of time to find schedules as SA. However, its schedules result in an average of just four migrations per update. This behavior gives GBM much more agility than SA (which helps in the highly dynamic experiments).

Fairness

We have yet to discuss fairness in detail. Previous experiments have been rather straightforward, and fairness could be deduced from CPU utilization plots. Here, however, it is not clear how fair our schedulers are since we have no way of knowing which VMs are hungry as the experiment runs. Each of Figures 4.21 through 4.23

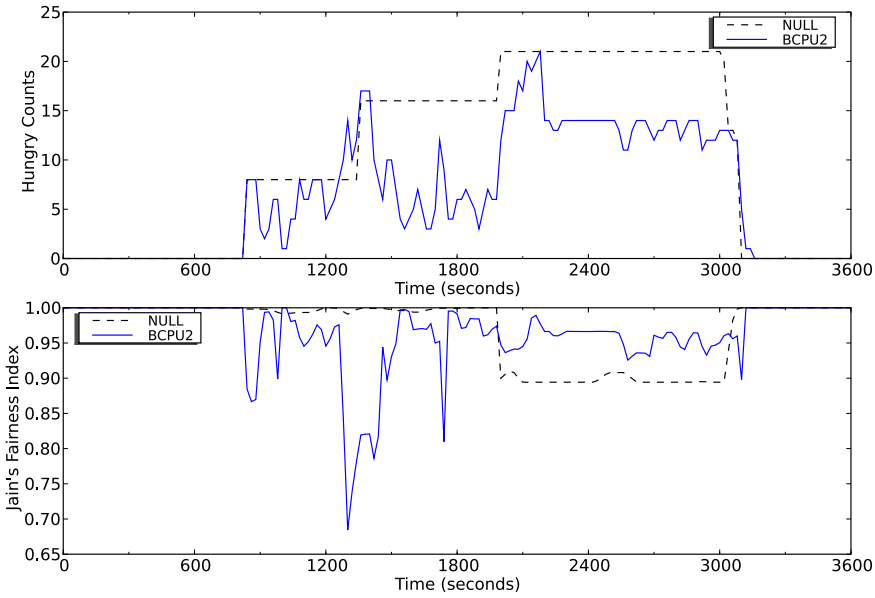

Figure 4.21 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for BCPU2 applied to GPCW-LD. BCPU2 is able to reduce hungry counts, but has difficulty with fairness.

contain low dynamism plots of CPU hungry VMs counts (top) and Jain's Fairness Index (bottom) as the simulation runs for BCPU2, SA, and GBM. These are plotted against the same values for the NULL scheduler for comparison.

We plot CPU hungry counts to show that periods where fairness is perfect are often periods where there are no hungry VMs in the system. Also, a scheduler able to reduce the number of hungry VMs is actually performing well by satisfying more VM demands.

Figure 4.21 reveals BCPU2's occasionally has trouble maintaining fairness as compared with the NULL scheduler. This should not be a surprise since the BCPU2 heuristic does not consider fairness, but tends to do well when there is low activity in the workload. Also notice that BCPU2 does succeed in decreasing the number of hungry VMs throughout the experiment. These results, combined with the utilization results, indicate that BCPU2 is a viable scheduler in this environment.

Figure 4.22 shows that SA does very well in this scenario. Though there is

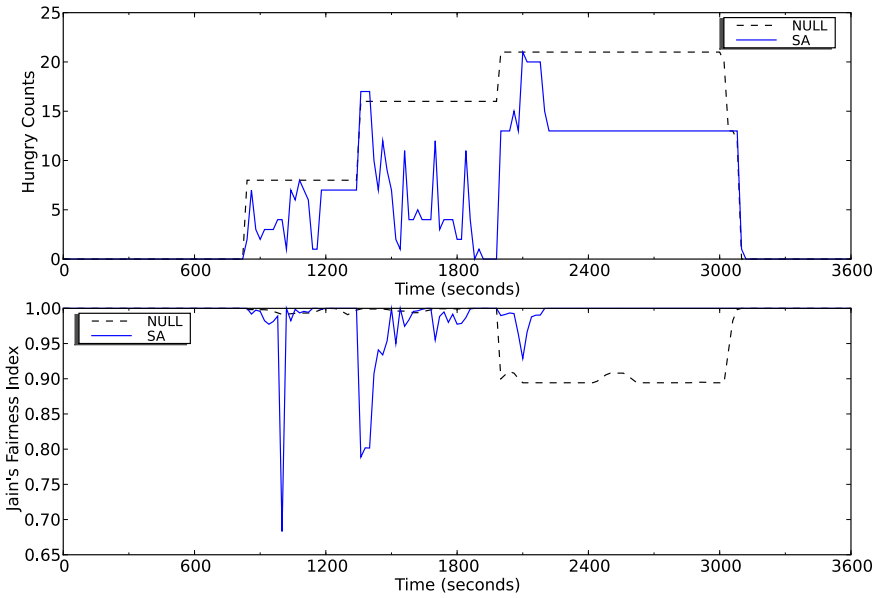

Figure 4.22 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for SA applied to GPCW-LD. SA does a better job of maintaining fairness than BCPU2 in this environment, often finding perfectly fair schedules.

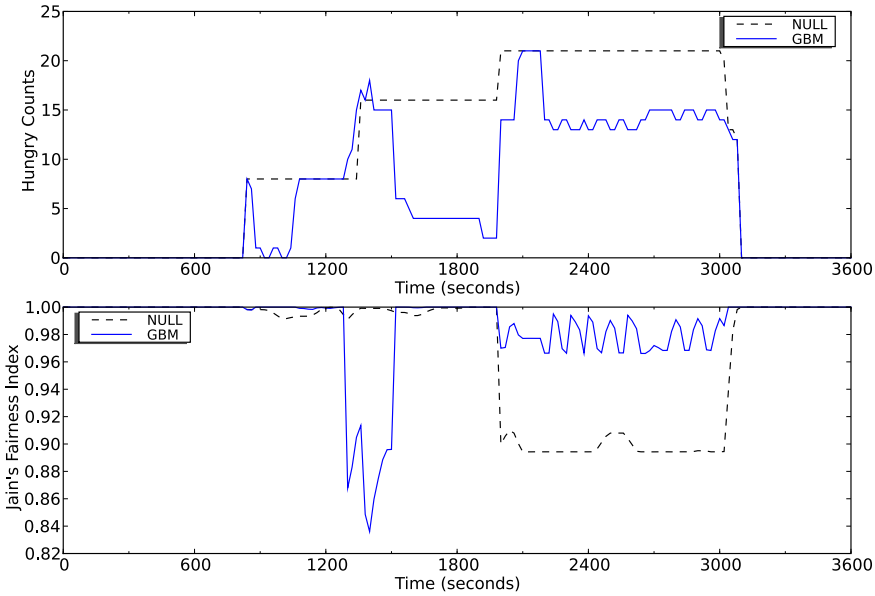

Figure 4.23 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for GBM applied to GPCW-LD. GBM does a good job maintaining fairness in this environment.

one brief spike, SA quickly corrects the unfairness and eventually finds a perfectly fair schedule. SA is certainly a viable scheduler in this low workload activity environment, outperforming BCPU2 in all areas.

Finally, Figure 4.23 shows GBM's effectiveness in maintaining fairness in this environment. Note that the scale is different on this fairness plot, and Jain's Fairness Index for GBM only briefly drops to 0.84.

4.3.2 Moderate Dynamism

We now increase our workload dynamism. For our moderate dynamism general purpose cluster workload experiments (GPCW-MD), we set our job average inter-arrival rate at 60 seconds. For job durations, we set our Weibull scale parameter to 121.7 (taken from [LGW04]) and leave our shape parameter set at 0.46. This results in jobs with average duration of 287 seconds and median duration of 55 seconds. We expect to see our slower schedulers begin to have difficulty with this scenario.

CPU allocations

Here we study our schedulers in an environment with moderate workload dynamism for each scheduler. Figure 4.24 contains plots of individual VM CPU allocations, total VM CPU utilization, and overall system CPU utilization (includes Domain-0 CPU allocations) for this scenario without a scheduler. We plot VM CPU allocations using solid lines (values on left y-axis), total VM allocation — as a percent of available CPU — with a dashed line (values on right y-axis), and overall CPU utilization with a dash-dot line (values on right y-axis). Notice that our workload is now more dynamic than the previous experiments with many more jobs arriving and ending during the one hour experiment.

Figure 4.25 contains VM CPU allocations for the BCPU2 scheduler applied to GPCW-MD. Clearly, BCPU2 is not ideal for this more dynamic setting. Notice that most schedules are less desirable than our round robin initial placement. Also, BCPU2 seems to begin having trouble finding decent schedules at approximately time 1500.

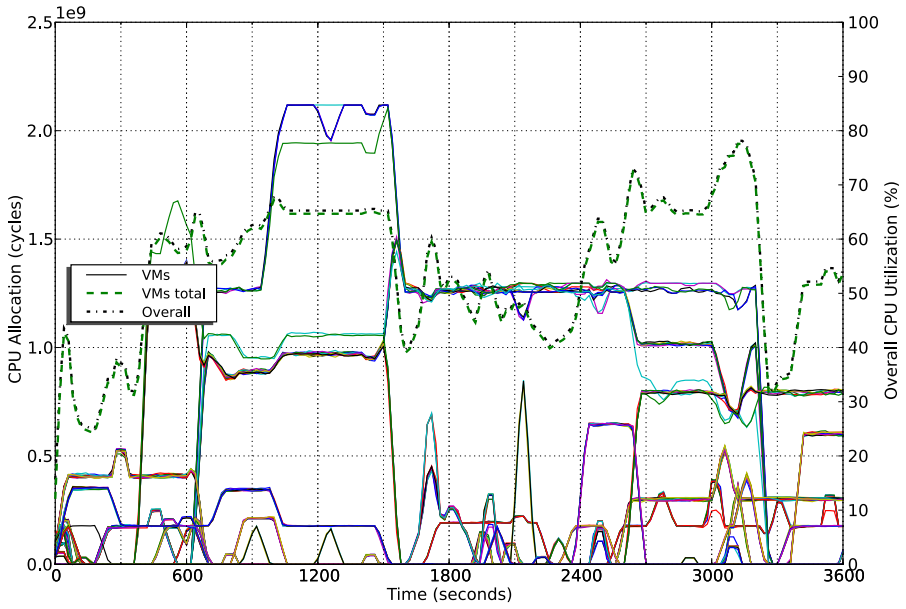

Figure 4.24 VM CPU allocations over time for NULL applied to GPCW-MD. The workload varies moderately during the one hour run.

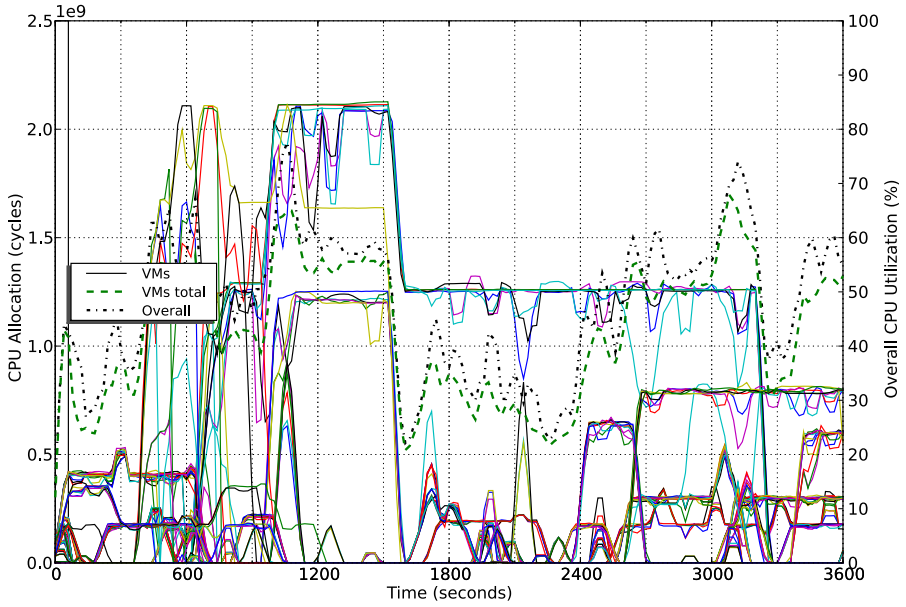

Figure 4.25 VM CPU allocations over time for BCPU2 applied to GPCW-MD. The excessive migrations are ineffective in this scenario with schedules often less desirable than the NULL scheduler.

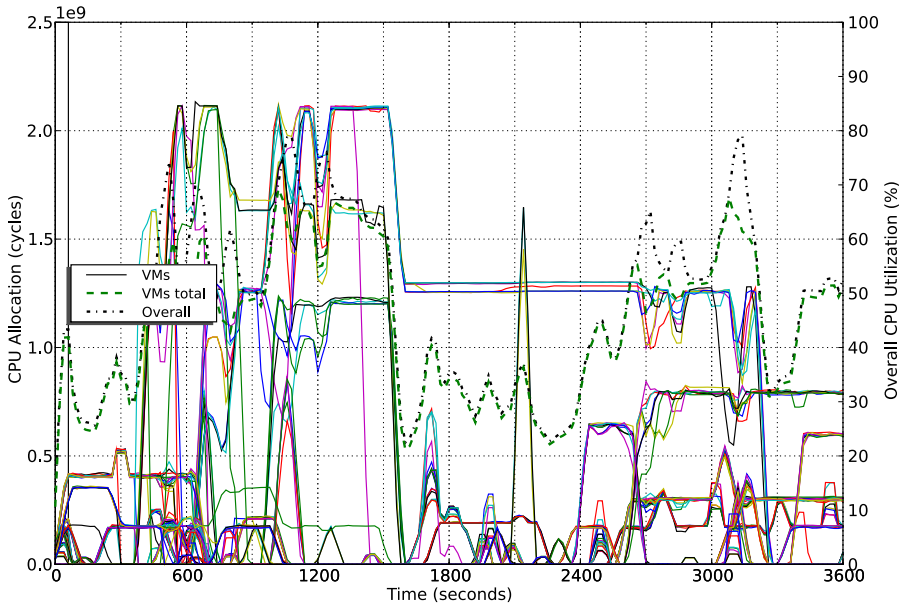

Figure 4.26 VM CPU allocations over time for SA applied to GPCW-MD. Though somewhat effective, SA suffers during periods of high activity.

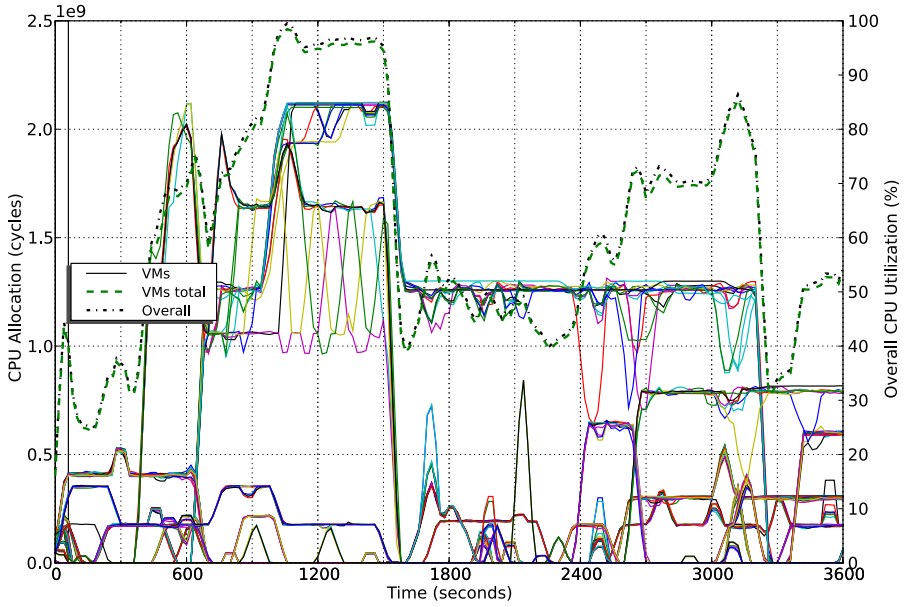

Figure 4.27 VM CPU allocations over time for GBM applied to GPCW-MD. GBMs low migration count adjustments easily handle this scenario. GBM clearly outperforms the NULL scheduler.

This could be due to poor packing of the workload combination. BCPU2 does recover from this toward experiment end, but this is obviously a poor choice under this scenario.

Table 4.4 contains costs of scheduling for all moderate dynamism general purpose cluster workload experiments. On average, BCPU2 took 35 ($3 + 32 = 35$) seconds to find and move to a new schedule. BCPU2 generated 43 migrations on average at each scheduling and a total of 1469. These are a bit higher than those of the low dynamism experiments. These higher migration counts are not surprising in this more dynamic setting with more VM CPU demand changes. Again, BCPU2 migrates VMs on 34 occasions over the one hour simulation, continuing its habit of rescheduling at every opportunity.

Table 4.4 Scheduling costs for moderate dynamism general purpose cluster workload experiments.

	BCPU2	**SA**	**GBM**
Avg schedule search time (sec)	3	19	14
Max schedule search time (sec)	9	26	21
Number of reschedules	34	8	24
Avg migration time (sec)	32	50	4
Max migration time (sec)	173	86	11
Avg migration count	43	106	4
Max migration count	85	117	20
Total migration count	1469	851	88

From Figure 4.26 SA begins to reveal its limitations. High migration counts preclude SA from scheduling often enough to improve over the NULL scheduler. SA was only able to reschedule eight times during the one hour experiment. On many occasions, SA performs worse than NULL.

On average, SA was able to find new schedules in 19 seconds with average migration time of 50 seconds. This total of 69 seconds to find and move to new schedules is nearly the same as the low dynamism experiments. However, this is not fast enough for this more dynamic setting.

Our GBM scheduler now begins to show its superiority for general purpose workload scheduling as it performs very well in this environment as well. Figure 4.27

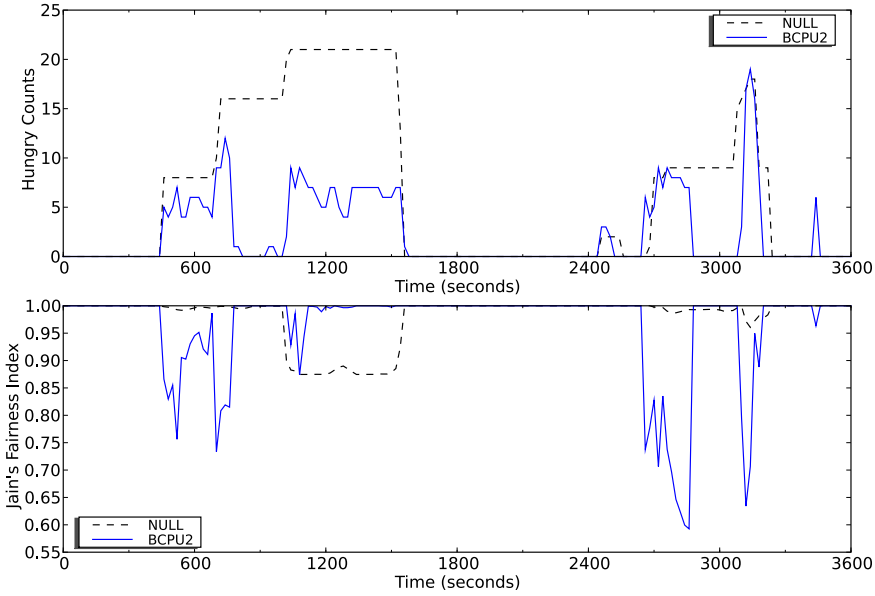

Figure 4.28 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for BCPU2 applied to GPCW-MD. Though it reduces hungry counts, BCPU2 results in poor fairness – occasionally under 65 percent.

contains CPU allocation plots for GBM applied to the GPCW-MD scenario. Again, GBM outperforms the NULL scheduler. From Table 4.4 GBM finds new schedules in an average of 14 seconds and migrates VMs in an average of four seconds. This results in an 18 second average to find new schedules and migrate VMs. Again, GBM migrates only four VMs on average. These small schedule adjustments give GBM much more agility than the other schedulers, making it superior in this environment.

Fairness

Each of Figures 4.28 through 4.30 contains plots of counts of CPU hungry VMs (top) and Jain's Fairness Index (bottom) as the simulation runs for BCPU2, SA, and GBM. These are plotted against the same values for the NULL scheduler for comparison.

Figures 4.28 and 4.29 show that BCPU2 and SA are able to reduce the number of hungry VMs. However, BCPU2 often has difficulty maintaining fairness with Jain's

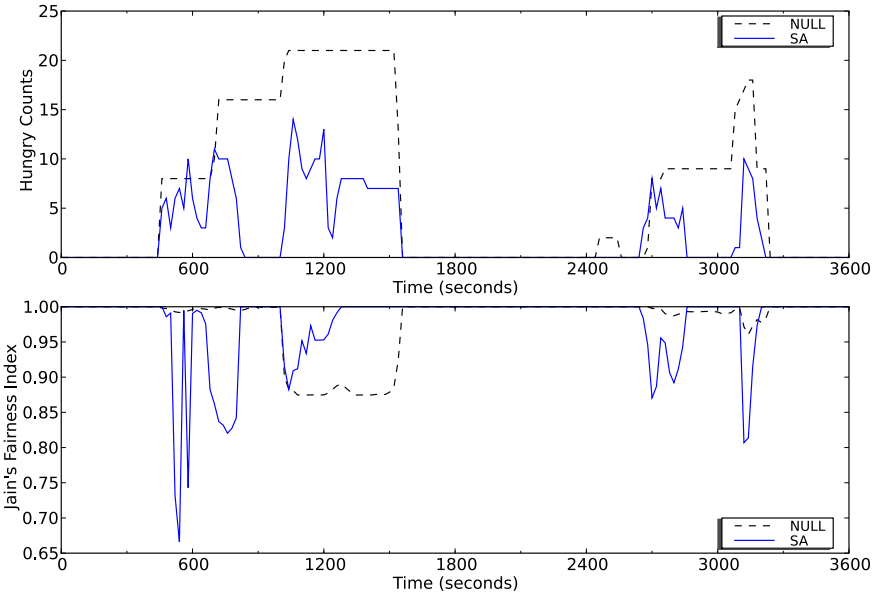

Figure 4.29 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for SA applied to GPCW-MD. SA greatly reduces hungry counts and does a fair job maintaining fairness in this environment.

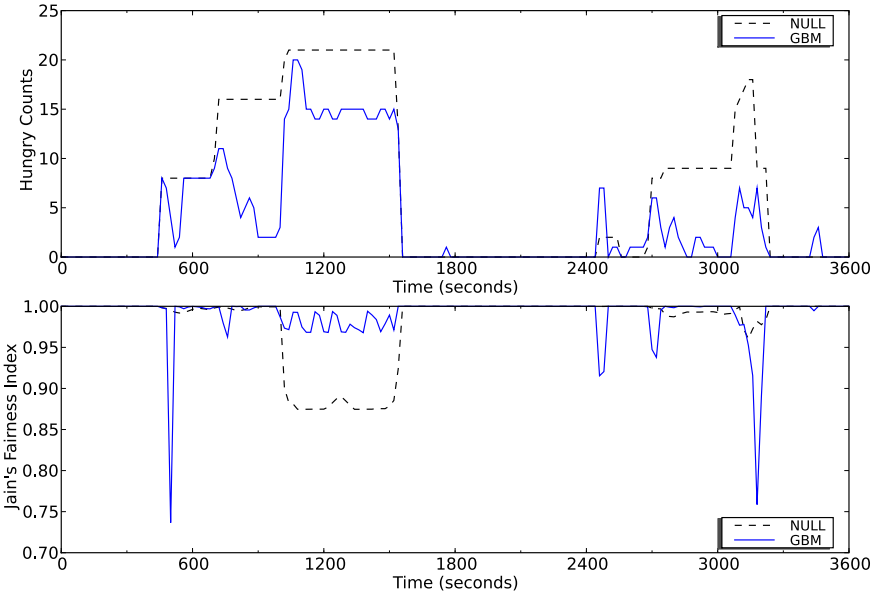

Figure 4.30 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for GBM applied to GPCW-MD. GBM does a decent job maintaining fairness in this dynamic environment.

Fairness Index dropping below 0.65 for a period. SA does a decent job of maintaining fairness. However, both of these schedulers performed poorly in overall CPU utilization, making them undesirable for this scenario.

Fairness for GBM can be seen in Figure 4.30. Compared to BCPU2 and SA, GBM does very well in terms of fairness. However, GBM does not reduce the hungry counts as much as the other two schedulers. Nonetheless, GBMs superior utilization and fairness measures make it a better choice here.

4.3.3 High Dynamism

For our high dynamism experiments, we set our average inter-arrival rate at 25 seconds. We refer to these as the "GPCW-HD" experiments. At this arrival rate, we ran experiments with job durations (in seconds) taken from a Weibull distribution with scale parameter of 60 and shape parameter of 0.46. These parameters correspond to a Weibull distribution with mean of 142 and median 27.

CPU allocations

Figure 4.31 shows individual VM CPU allocations (solid lines) for this scenario without a scheduler. Notice that the workload is highly dynamic and varies considerably over the one hour run. Many jobs last under a minute and appear a small bumps in the corresponding VM allocation plot.

Figure 4.32 contains VM CPU allocations for the BCPU2 scheduler applied to GPCW-HD. Clearly, BCPU2 is not ideal for this dynamic setting. Excessive migrations impose high CPU loads in the node Domain-0s, robbing cycles from VMs. This is evident from the often large difference between the VM total and overall utilization lines in Figure 4.32. Notice that most schedules are less desirable than our round robin initial placement.

Table 4.5 contains costs of scheduling for all high dynamism general purpose cluster workload experiments. Unfortunately, time to find a new schedule was not recorded in these experiments. However, we expect these times to be similar to

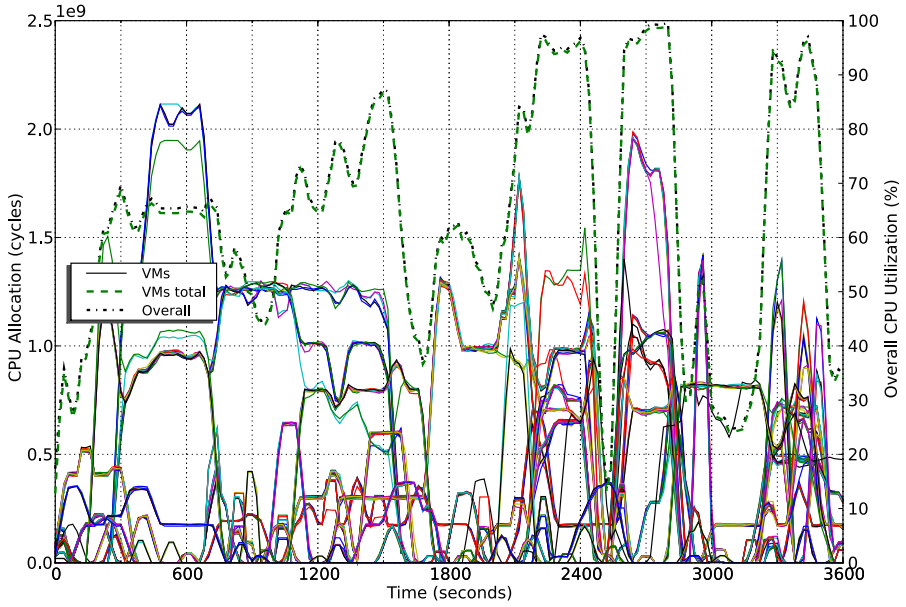

Figure 4.31 VM CPU allocations over time for NULL applied to GPCW-HD. The workload varies considerably during the one hour run.

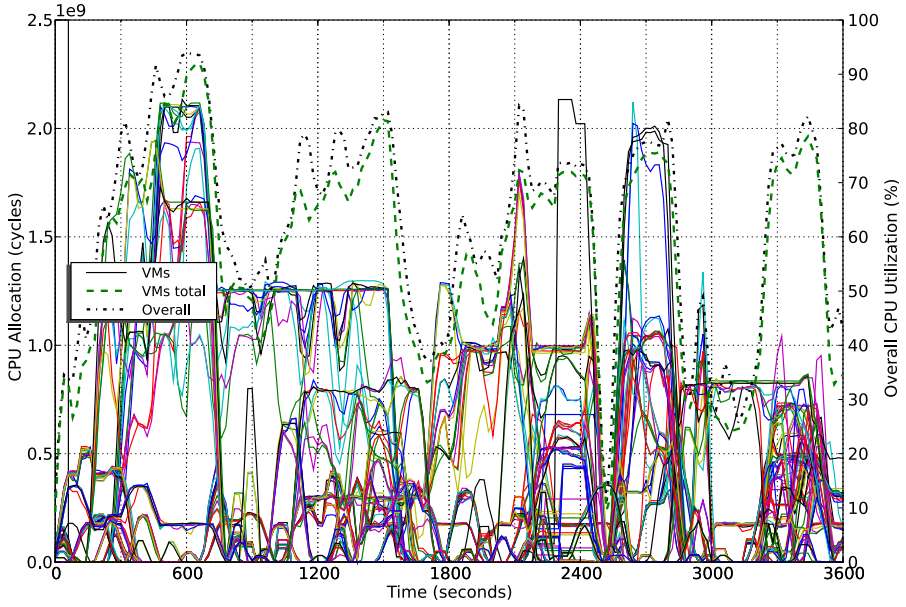

Figure 4.32 VM CPU allocations over time for BCPU2 applied to GPCW-HD. Excessive migrations rob VMs of CPU cycles. Schedules are often less desirable than initial placement.

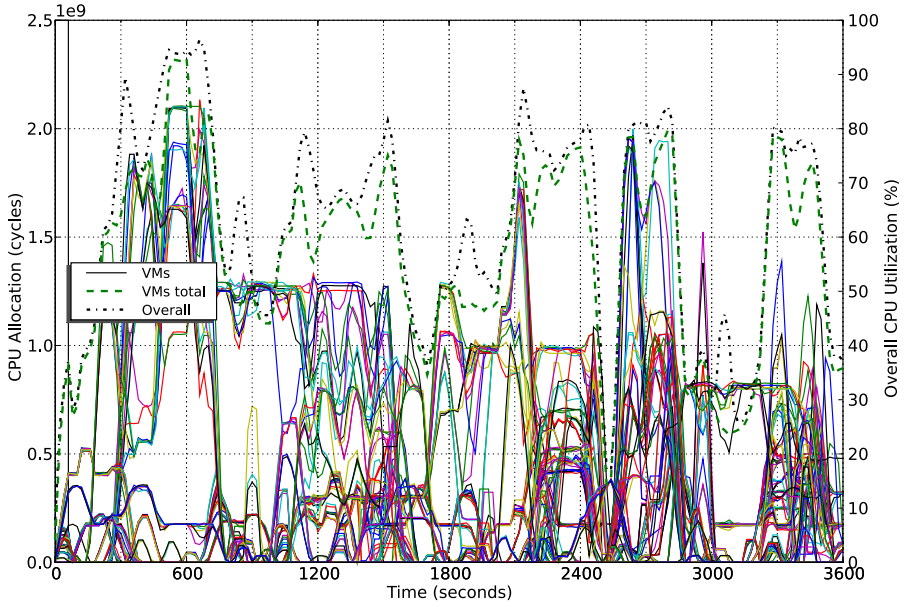

Figure 4.33 VM CPU allocations over time for SA applied to GPCW-HD. SA's high migration count scheduling cannot handle this highly dynamic scenario.

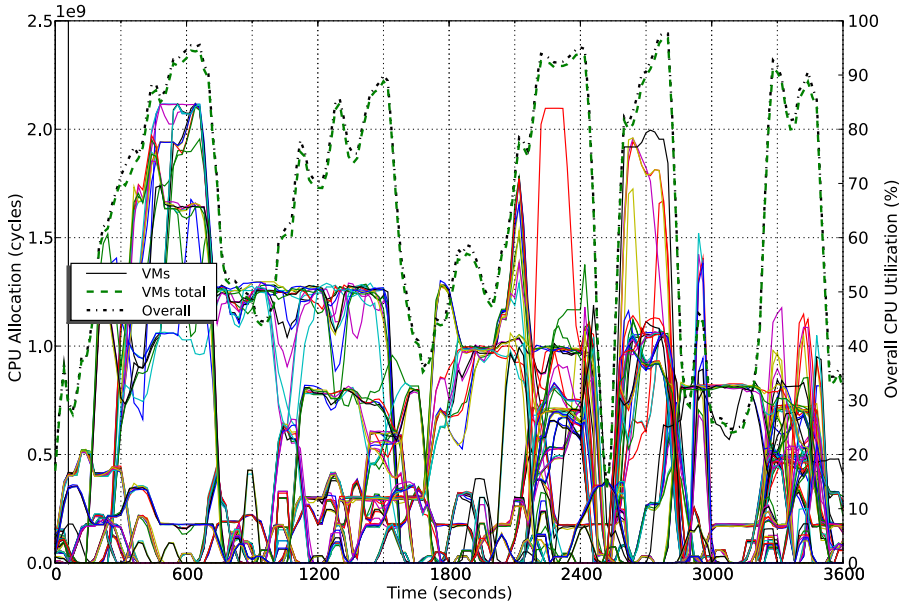

Figure 4.34 VM CPU allocations over time for GBM applied to GPCW-HD. GBM's low migration count adjustments make it feasible even in this highly dynamic environment.

the previous two scenarios. Migration time averaged 86 seconds with a maximum time of 258 seconds for this scenario. So, on average, BCPU2 took at least 86 seconds to find and move to a new schedule. On average, BCPU2 generated 67 migrations at each scheduling and a total of 1410 migrations in 21 schedules over the one hour simulation.

These numbers shine light on BCPU's poor performance in this scenario. Finding optimal schedules is of questionable value if the time it takes to move from one schedule to another is longer than the average time between workload change events.[6]

Table 4.5 Scheduling costs for high dynamism general purpose cluster workload experiments.

	BCPU2	**SA**	**GBM**
Avg schedule search time (sec)	NA	NA	NA
Max schedule search time (sec)	NA	NA	NA
Number of reschedules	21	12	28
Avg migration time (sec)	86	134	9
Max migration time (sec)	258	366	59
Avg migration count	67	113	6
Max migration count	91	119	20
Total migration count	1410	1358	162

Figure 4.33 reveals that our SA solver does not fare much better. Again, excessive migrations take away significant cycles from the VMs themselves. This environment is clearly too dynamic for a scheduler with such high migration counts. On average, SA performed 113 migrations per schedule in this scenario. This migration count resulted in an average of 134 seconds to migrate VMs to their new locations with a maximum migration time of 366 seconds! As with BCPU2, finding and moving to optimal schedules takes longer than the average time between workload change events. Also notice that SA was only able to find 12 new schedules due to its long migration times. This lack of agility makes SA impractical for this scenario.

As we might expect, our GBM scheduler performs much better in this dynamic environment. Figure 4.34 contains CPU allocation plots for GBM applied to the GPCW-HD scenario. GBM's tendency to make slight adjustments to the current schedule results

[6] It is also unadvised for this time to be greater than the history with which the migration decisions were made.

in better schedules with little overhead. As seen in the first 10 minutes, GBM is able to find much better schedules when they exist. Also, it never lags more than a few percent from the NULL scheduler's overall utilization levels.

Table 4.5 reveals the key reason GBM is superior to BCPU2 and SA in this scenario. GBM's average migration count of 9 and average migration time of 6 seconds allowed it to make 28 slight adjustments totaling 162 migrations over the hour long duration. Such agility is critical to successful scheduling in this highly dynamic environment.

Fairness

Each of Figures 4.35 through 4.37 contains plots of counts of CPU hungry VMs (top) and Jain's Fairness Index (bottom) as the simulation runs for BCPU2, SA, and GBM. These are plotted against the same values for the NULL scheduler for comparison.

Figure 4.35 reveals BCPU2's inability to effectively maintain fairness as compared with the NULL scheduler. This is not a surprise since the BCPU2 heuristic does not consider fairness.

Figure 4.36 shows that SA also suffers in its ability to effectively maintain fairness. Although somewhat better since it does consider fairness, SA is simply too slow to correct bad or unfair schedules. Also notice that SA often increases the number of hungry VMs. This increase is due to the high migration overhead which steals CPU cycles from VMs. Clearly, SA is inappropriate for this environment.

Finally, Figure 4.37 shows GBM's effectiveness in maintaining fairness in this dynamic environment. The fact that GBM performs at least as well as NULL in this highly dynamic environment is quite impressive. GBM's low migration count adjustments make it agile enough to effectively schedule in environments with this level of dynamism.

In reality, as clusters are started and stopped, we expect NULL to be inadequate for any level of dynamism. Furthermore, our overly simplistic assumption that

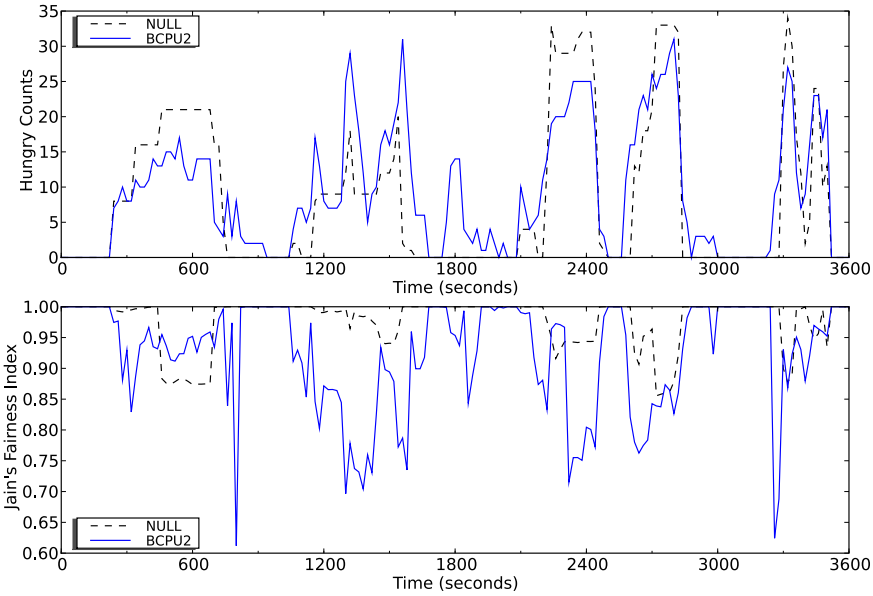

Figure 4.35 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for BCPU2 applied to GPCW-HD. BCPU2 results in poor fairness – occasionally under 65 percent.

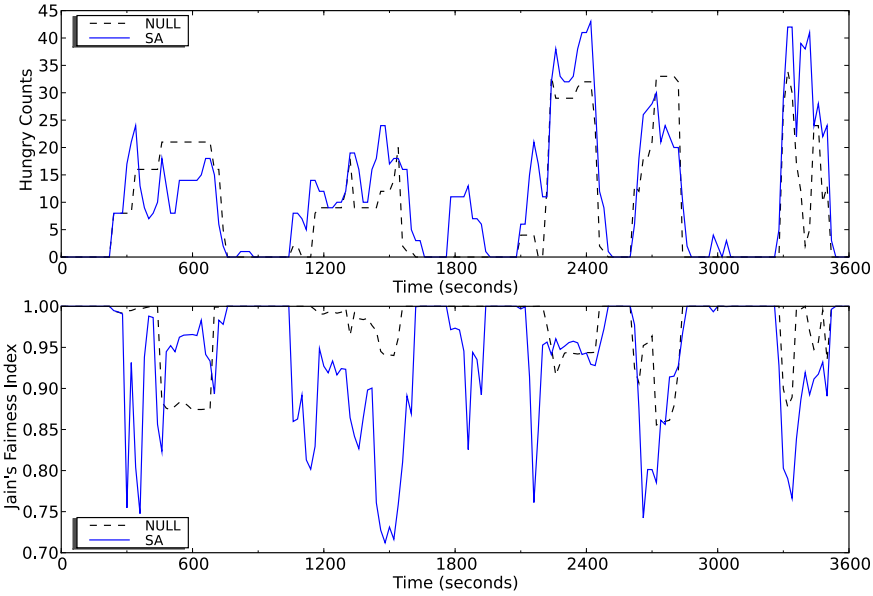

Figure 4.36 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for SA applied to GPCW-HD. SA does a poor job maintaining fairness in this dynamic environment. High migration overhead also results in high hungry VM counts.

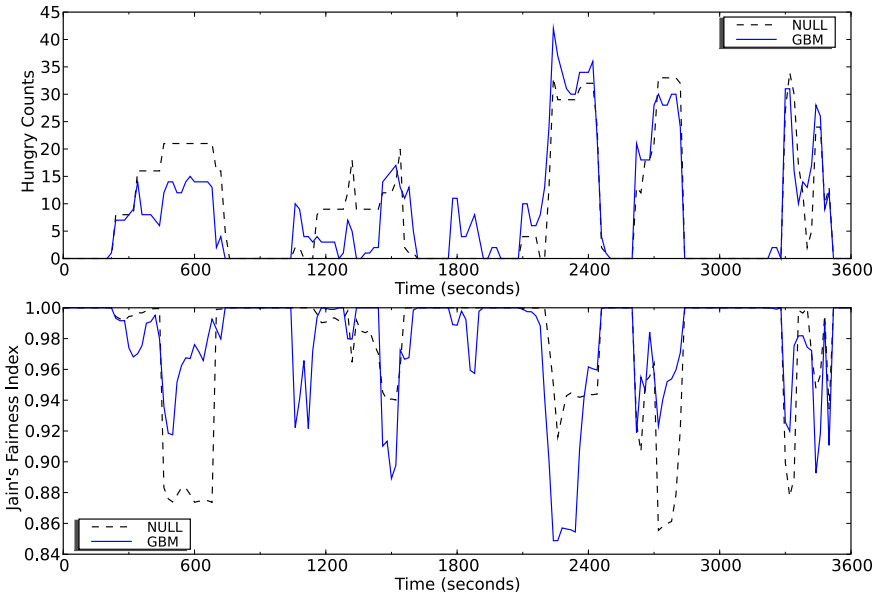

Figure 4.37 Counts (top) and Jain's Fairness Index (bottom) of CPU hungry VMs over time for GBM applied to GPCW-HD. GBM does a decent job maintaining fairness in this dynamic environment.

all VMs in a cluster have the exact same resource demands highly favors the NULL scheduler. In reality, we don't expect this to hold in every case. Therefore, a GBM like scheduler is recommended over doing no scheduling, even in more dynamic environments like this.

4.4 Conclusions

In this chapter we presented experimental results for our scheduling heuristics under two scenarios: hot spot alleviation and general purpose cluster workloads. Small scale hot spot alleviation experiments shed light on each scheduler's behavior when confronted with an imbalance in resource demands across the cluster. We found that our less sophisticated Balanced CPU (BCPU and BCPU2) schedulers were able to find reasonable schedules, but often continued needlessly rescheduling. Resistance thresholds and the constant ordering modification of BCPU2 did not alleviate this. Larger scale

experiments confirmed that these behaviors hold for higher numbers of VMs and PMs.

Our Simulated Annealing (SA) and Greedy Best Move (GBM) schedulers displayed their ability to quickly find optimal solutions in small scale hot spot alleviation experiments. However, SA suffered from high migration counts to move from the current assignment to its improved assignment. GBM, on the other hand, found good assignments in just a few migrations although it occasionally required multiple scheduling rounds to arrive at optimal solutions. These characteristics held for both schedulers in the larger scale experiments.

General purpose cluster workload experiments confirmed that many of the behaviors witnessed in the hot spot alleviation experiments held in a dynamic environment. These experiments also revealed weaknesses in our schedulers under certain workloads. First, we did not even present results for BCPU since its excessive migration counts created kernel and Xen instabilities in our testbed. Similarly, BCPU2 often resulted in high migration counts, albeit not as severe as BCPU. BCPU2 results confirmed that BCPU2's inability to recognize good assignments and not schedule make it inappropriate for even low dynamism environments. Also, BCPU2 often resulted in poor fairness between hungry VMs since it does not consider fairness in its scheduling heuristic.

Although SA worked well in our low dynamism general purpose cluster workload experiments, its high migration counts made it inappropriate in more dynamic settings. Further work on reducing SA migration counts would be required to make SA a viable general purpose scheduling heuristic.

GBM's low migration count adjustments made it agile enough to effectively schedule over the full range of dynamism levels tested. Its resource allocation prediction based move selection made its moves more likely to be utility improving moves — resulting in vastly improved utility in just a few migrations. Our experiments reveal that GBM is clearly suitable for scheduling in a range of cluster environments.

Though GBM found and moved to new schedules much faster than our other heuristics, rules of thumb for how fast a scheduler should be to be effective are, unfortunately, difficult to specify. This really depends upon what characteristics of the workload

one wishes to target. For example, even in highly dynamic environments, there exist underlying stabilities which an administrator may wish to target over short-lived workload events. In a general purpose setting, it is easy to imagine some users will run long jobs (e.g., compute jobs) or services (e.g., apache), while others run many very short lived jobs (e.g., grep). It may be reasonable to schedule infrequently for these longer lived jobs and disregard the short jobs as background noise which does not significantly affect the long term balance of the system.

In our environment, we chose one minute as our minimum scheduling frequency and based our decisions on a VMs one minute load average. In this setting, the longer a process runs, up to one minute, the more impact it will have on new schedules. Though not absolutely necessary, we recommend that a scheduler be able to find and arrive at new schedules in less time than the load average interval used to determine its new schedules.

Chapter 5

Conclusions

Managing virtual machines in a virtual cluster environment poses difficult challenges for site administrators. In this dissertation, I presented Usher, an extensible, event-driven management system for managing clusters of virtual machines which vastly improves an administrators ability to confront the challenges presented by this complex computing environment. Usher's extensibility via its plugin system adds the flexibility to integrate Usher into a site's existing infrastructure and enforce a site's administrative policies and operational goals.

Usher improves administrator efficiency in both virtual cluster setup and maintenance. Usher enables clusters of arbitrary size to be created with a single command, reducing cluster setup time from hours to seconds. Usher's plugin system relieves administrators from the need to manually enforce their site's administrative policies or approve all VM operation requests. This delegation of authority allows administrators to push many VM administration tasks out to the users themselves. Plugins also enable administrators to charge Usher with the responsibility of maintaining their site's operational goals. Scheduling plugins continuously monitor a site's operational status and adjust VM assignments to PMs (via transparent live migration) when goals are not being met.

I implemented one such scheduling plugin and designed heuristics of varying levels of sophistication for optimizing the utility of VM assignments to PMs based upon

utility functions designed for a site's operational objectives. I defined the Fair Maximum Utilization (FMU) problem as a possible operational goal for the SysNet group and proved its inclusion in the set of NP-hard problems. Experimental results support that, using the Greedy Best Move (GBM) heuristic which makes good VM resource allocation predictions and only slight schedule adjustments, Usher is able to effectively maintain the goal of fairly maximizing resource consumption for virtual cluster workloads of varying levels of dynamism.

Work remains for future study. There are a number of additional experiments which could be run to show my scheduler's adaptability to more complex computing environments. Time constraints did not allow for experiments involving scheduling across a cluster of multi-core computers. Likewise, experiments in a heterogeneous environment would most likely have revealed the inadequacy of round robin placement without scheduling in such a cluster. My schedulers consider factors such as resource capacity (e.g., CPU speed) and multiple resource instances per node (e.g., multiple CPUs), yet experimental results for these environments were not generated. Such results would further illustrate the power of the schedulers presented here.

Better scheduling heuristics should also be sought. Alternate heuristics with the properties of good resource allocation predictions and small schedule adjustments should be explored, in addition to improvements to GBM itself. GBM's lack of tracking the moves it will suggest causes it to make mistakes such as suggesting migrating two hungry VMs to the same node. It, therefore, misses many good moves after the first.

It may also be worthwhile to attempt to reduce the number of moves suggested by the simulated annealing (SA) heuristic by categorizing and eliminating unnecessary moves. For example, SA may suggest swapping two VMs with nearly identical resource demands in its search for better assignments. An efficient method for detecting these unnecessary moves may make SA a viable option for VM scheduling.

The GBM-SA hybrid approach was not explored. In addition, more efficient metaheuristics such as Tabu search could be evaluated. These heuristics are likely to reduce migration counts due to their faster convergence and more efficient searching of

the solution space.

Another area for further study is the relationship between non-CPU resource consumption and induced CPU load. Due to the strong dependence of non-CPU resource availability on CPU availability, the importance of considering non-CPU resources in scheduling is not known. Once established, scheduling heuristics can use this relationship to better predict node-local resource allocations when VMs are migrated to and from physical nodes. Comparing results from a multi-resource scheduler with those from a CPU-only scheduler will reveal whether scheduling for CPU alone is sufficient.

Appendix A

Code Overview

Code for the Usher system is written in Python and makes heavy use of Twisted [twi], an event-driven networking framework also written in Python. In particular, Usher uses Twisted's Perspective Broker module to provide asynchronous RPC between clients and the controller and LNMs and the controller. See the documentation on the Twisted site for the fine details of using this module. As shown in Figure 2.2 of Section 2.3.2, the Usher system consists of three main components: a centralized controller, local node managers (LNMs), and clients.

Let us go straight to the code to see where these are implemented. Below is a tree listing of the code directory (`usher/`) in the top-level project directory.

```
usher/
|-- __init__.py
|-- cli
|   |-- __init__.py
|   |-- api.py
|   |-- callbacks.py
|   |-- config.py
|   |-- config_skel.py
|   `-- vm.py
|-- ctrl
|   |-- __init__.py
|   |-- app.tac
|   |-- client.py
|   |-- cluster.py
|   |-- config.py
```

```
|   |-- lnm.py
|   |-- request.py
|   |-- server.py
|   `-- vm.py
|-- lnm
|   |-- __init__.py
|   |-- app.tac
|   |-- config.py
|   |-- server.py
|   |-- stats_upldr.py
|   |-- vm.py
|   `-- xen_vmm.py
|-- plugins
|   |-- __init__.py
|   |-- sample.py
|   |-- sample2
|   |   |-- __init__.py
|   |   `-- plugin.py
|   `-- test.sh
`-- utils
    |-- __init__.py
    |-- config.py
    |-- credcheck.py
    |-- events.py
    |-- misc.py
    |-- notify.py
    |-- plugin.py
    |-- result.py
    |-- upb.py
    `-- usherr.py

6 directories, 37 files
```

Also in the top-level directory are `configs/` and `initscripts/`. The `configs/` directory contains default configuration files for each of the controller, LNMs, and client API. Each file contains a line indicating where Usher will check to find the file or an environment variable to specify where to look. The `initscripts/` directory contains simple startup scripts for both the local node managers and controller. After editing these files for your installation, they should be placed where your system will find and run them at boot time or run manually when you want to start the services.

A.1 Usher Components

From the listing above, notice that each of the main components has a directory in the `usher/` directory wherein its source code resides. We now discuss these in turn.

A.1.1 Controller (`usher/ctrl/`)

Beginning with the controller, we see eight files (ignoring the `__init__.py` files). Below, we give a brief explanation of the purpose of each and any important details. For further information, please refer to the code itself. Also in this section, we introduce a few Twisted objects used by Usher without much explanation. For now, please bear with us and we will give a more detailed explanation of these objects in the following section.

`app.tac`: This is a Twisted application file. In a nutshell, Twisted provides facilities for creating a forking, daemonized server using their "twistd" application. This saves us from having to write that part of the code ourselves. Regular Python code is used in the `app.tac` file which essentially contains code for starting various Twisted servers under a single controlling process. If you need an additional listening server (or client connection) at process start, it should be added there.

`client.py`: This file provides the implementation of the Twisted `IRealm` interface for use by clients as well as the Client API (the `pb.Avatar`). This is the API exposed to clients connecting to the controller.

`cluster.py`: This defines the Cluster object which is a `pb.Cacheable`. Not much is really done with this at this point. It is updated and made available to plugins for their use.

`config.py`: This file subclasses the `Config` object (found in `usher/utils/config.py`). It contains a `ConfigObj` specification file which specifies what configuration parameters will be accepted and default

values for each. Each new configuration parameter should be added to this specification. Configuration files are read at service startup and values set at that time. The `config.py` file contains a function named `get_cfg` for retrieving a reference to the configuration object. An optional section parameter can be passed to `get_cfg` to only get values from a particular section of the configuration file.

`lnm.py`: This is the LNM counterpart to `client.py`. It provides the implementation of the Twisted `IRealm` interface for use by LNMs as well as the LNM API (the `pb.Avatar`). This is the API exposed to LNMs connecting to the controller.

`request.py`: This file defines the `Request` object and all `Request` subclasses (`StartRequest`. `MigrateRequest`, etc). Every request to the controller from a client generates a `Request` object which is first passed to plugins registered for that request type. After traversing its plugin chains, this object is used to initiate the request via its `execute` method.

`server.py`: The core server code can be found here. Looking at the code for the controller, we first see the `UsherCtrl` class. The `UsherCtrl` class is defined here which is basically a container class for maintaining global state variables and provides accessor methods to those needed elsewhere. In addition, the constructor sets up the services we start in the `app.tac` file and retains references to those.

`vm.py`: This file provides the implementation of the `VM` object. This is a `pb.Cacheable` which gets passed to clients and LNMs. Clients which own a VM always receive a cached copy of the `VM` object on the controller. Likewise, LNMs get a cached copy for every VM running locally. This object provides most methods for manipulating VMs (e.g., `start`, `shutdown`, `migrate`, etc). In addition, all operations on a particular VM are serialized here.

A.1.2 Unraveling Twisted

Before moving on, a few notes on what all the Twisted lingo used above really means, and how Twisted's Perspective Broker works are in order.

First, in Twisted's Perspective Broker module, a `Realm` generates capabilities which are returned to authenticated clients. In Twisted speak, the `Realm` returns an "`Avatar`", which is a remote reference to an API. A `Realm` in Twisted gets passed along with a list of Twisted credential checkers (something that implements the `twisted.cred.checkers.ICredentialsChecker` interface), to the constructor of the `twisted.cred.portal.Portal` class to generate our `Portal`. From the Twisted API documentation [twi] for `twisted.cred.portal.Portal`:

> A `Portal` is associated with one `Realm` and zero or more credentials checkers. When a login is attempted, the `Portal` finds the appropriate credentials checker for the credentials given, invokes it, and if the credentials are valid, retrieves the appropriate `Avatar` from the `Realm`.

That about says it all as far as `Portal`s are concerned. The `Portal` is then given to the server factory for perspective broker (`twisted.spread.pb.PBServerFactory`), which is passed to the `twisted.application.internet.SSLServer` method (in `app.tac`) to connect the factory to the network and start the Twisted event loop.

Getting down to the gory details, when a client connects to the controller (We restrict the discussion to `CliRealm` since it is basically the same for `LNMRealm`), the `requestAvatar` method of the `CliRealm` instance gets fired. This method returns a reference to a `pb.Avatar` (a remote API reference) to the client. This method does a few things, like add the actual `Avatar` to the client's dictionary in the `UsherCtrl` instance if it does not already exist, and calls the `Avatar`'s attached method.

More interesting is the `mind` parameter to the `requestAvatar` method. This is a remote reference back to the client which is passed by the client when running login. Through this, the controller is able to call back methods which the client exposes to the controller (by prepending the method names with `remote_`). This is quite handy for sending notifications back to the client. Finally, notice that the `requestAvatar` returns a 3-tuple. The first item of the tuple is simply the interface which the `Avatar` implements. The second is the remote reference to the `Avatar`. The third is a no-argument function which the protocol should call when the client connection has been

lost.

Now, the fun begins. Continuing with the client, when a client connects (via `login`), the `CliRealm` instance returns a reference to an instance of the `Client` (which is an `Avatar`). This `Avatar` is essentially a remote reference to an API exposed by the controller to the client. With this remote reference, the remote client can call any methods beginning with "`perspective_`" in the `Client` class (i.e., the `pb.Avatar` class). These methods, in turn, generate `Request` objects which are passed to plugin chains before being carried out.

In Twisted, all remote method calls are asynchronous, immediately returning a `Deferred` (i.e., an instance of the `twisted.internet.defer.Deferred` class). This `Deferred` is then called back with the result of the remote procedure call when it becomes available. Handling this result is done via a callback chain which is created on the caller's side using the `Deferred`'s `addCallback` method. A callback chain can be made arbitrarily long by simply adding more and more callbacks with this method. The first callback in the chain is passed the result of the RPC as its first parameter. Subsequent callbacks are called with the result of the previous as their first parameter. Errors can be handled in a similar manner using the `Deferred`'s `addErrback` method. The fine details of Twisted `Deferred` can be found in the main Twisted manual.

So, nearly all remote API calls are followed by the creation of a callback chain to handle the results. Hence, all the real work is done by functions added to the callback chain in the API method.

A.1.3 Local Node Manager (`usher/lnm/`)

Next, we give a brief explanation about the purpose of each of the LNM files and any important details. For further information, see the code itself.

`app.tac`: See the explanation for the controller's `app.tac` file above.

`config.py`: See the explanation for the controller's `config.py` file above.

`server.py`: The main LNM server code can be found here. It also provides the API used by the controller to request VM administrative operations. These methods are found in the LNM class beginning with "`remote_`".

`vm.py`: The `LNMVM` class is implemented here. This is a `pb.RemoteCache` object for the `VM` objects created at the controller. This file handles setting up the cached copy and methods for keeping it synchronized with the real `VM` object at the controller.

`xen_vmm.py`: This file provides a wrapper around the Xen virtual machine manager administrative API for use by the LNM. This wrapper is used by the LNM to manage locally running virtual machines.

LNMs are simply clients of the controller which connect to a different service than Usher clients. Being a client is slightly different from a server, and the main class, `LNM` in the LNM's `server.py` file, subclasses `pb.Referenceable`. This allows our main class to define methods beginning with `remote_` to be made available to anything holding a reference to the `LNM` instance. Hence, when our LNM server calls the `login` function, it passes a reference to itself (the `client` argument), to the controller. Now, any methods beginning with "`remote_`" can be called by the controller.

The controller is not a client of the LNMs, since the PB protocol is symmetric once a connection is established — each end has a reference to an API on the other. So we make the LNM the client in this connection since its much easier for the LNMs to find the centralized controller than the other way around. In addition, we want the LNMs to have to provide credentials to the controller before becoming a trusted member of the system. Users may not want their VMs running on unauthenticated nodes.

Finally, notice that all VMM specific code has been pulled out into a separate module. The module should be named `<VMM type>_VMM.py`. The `LNM` class of `server.py` subclasses the `LNMBase` class of the VMM specific module. Currently, only `Xen_VMM.py` exists.

A.1.4 Client API (`usher/cli/`)

Next, we give a brief explanation about the purpose of each of the client files and any important details. For further information, see the code itself.

`api.py`: This is a library which clients can use to access the Usher controller. Developers wishing to tie their application to the Usher system can use the API presented in `api.py` to do so. Clients create an instance of the `API` by simply importing api. From there, the instance can be grabbed using the `api.get_api` method.

`callbacks.py`: This provides a callback interface which applications can subclass to receive event notifications. Applications must override the methods there to receive callbacks for the respective event.

`config.py`: See the explanation for the controller's `config.py` file above.

`config_skel.py`: This is a skeleton `config.py` file for use by client application writers to define configuration options for their applications.

`vm.py`: See the explanation for the LNM's `vm.py` file above.

A.1.5 Utilities (`usher/utils/`)

Utility modules are located in the `usher/utils/` directory. These include:

`config.py`: Other modules get their configuration by importing this module and using the `get_cfg` function to receive a `ConfigObj` instance containing configuration parameters. This module also adds a few parameter types.

`credcheck.py`: Twisted's `checkers.ICredentialsChecker` interface is implemented here to allow plugins registered for the `client_authenticate` and `lnm_authenticate` events to authenticate users. See the Appendix C.2 for additional information on writing credential checkers.

`events.py`: This file contains a dictionary with all valid events for which plugins can be registered in the Usher system. These are listed in Appendix B. This file also defines the important `EventListDispatcher` class which is the object with which plugins are registered to be called for the respective event.

`misc.py`: A few miscellaneous methods which do not really belong anywhere else are implemented here.

`notify.py`: This file contains logging and error notification methods.

`plugin.py`: This file provides the `UsherPlugin` class which all plugins subclass. It also contains utility methods for finding available plugins at controller startup and handling plugin specifications.

`result.py`: The very important `UsherResult` class is defined here. An `UsherResult` instance is returned by the controller for most client API calls. It slso contains the `UsherResPkgr` class which packages results from a list of `Deferreds` from an API call into an `UsherResult`.

`upb.py`: This file is necessary because of Twisted's inadequate credential checking system.

`usherr.py`: All Usher exceptions are defined here.

Appendix B

Usher Events

Below is a complete list of events in the core Usher system. This list can be extended by plugins.

Client Events:

- **client_authenticate:** A client is attempting to authenticate.
- **client_connect:** A client has connected.
- **client_disconnect:** A client has disconnected.

Cluster Events:

- **cluster_register:** A new cluster has been registered with the controller.

Controller Events:

- **ctrl_start:** The Controller has started.
- **list_request:** A request to list VMs has been received.
- **periodic:** A periodic timer has fired. For scheduling periodic tasks.
- **timer:** A timer has fired. For scheduling 1-shot future tasks.
- **request:** A request has been received (any type).

- **lnm_list_request:** A request to list LNMs has been received.
- **api_extension:** An api extension method has been called (pseudo event nothing called)

LNM Events:

- **lnm_authenticate:** An LNM is attempting to authenticate.
- **lnm_connect:** A Local Node Manager has connected.
- **lnm_disconnect:** A Local Node Manager has disconnected.

VM Events:

- **cycle:** A VM has been cycled.
- **cycle_failure:** A VM cycle has failed.
- **cycle_request:** A request to cycle a list of VMs has been received.
- **migrate:** A VM has been migrated.
- **migrate_failure:** A VM migration has failed.
- **migrate_request:** A request to migrate a list of VMs has been received.
- **pause:** A VM has been paused.
- **pause_failure:** A VM pause has failed.
- **pause_request:** A request to pause a list of VMs has been received.
- **poweroff:** A VM has been powered off.
- **poweroff_failure:** A VM poweroff has failed.
- **poweroff_request:** A request to poweroff a list of VMs has been received.
- **reboot:** A VM has been rebooted.

- **reboot_failure:** A VM reboot has failed.
- **reboot_request:** A request to reboot a list of VMs has been received.
- **register:** A VM has been registered with the controller.
- **resume:** A VM has been resumed.
- **resume_failure:** A VM resume has failed.
- **resume_request:** A request to resume a list of VMs has been received.
- **shutdown:** A VM has been shutdown.
- **shutdown_failure:** A VM shutdown has failed.
- **shutdown_request:** A request to shutdown a list of VMs has been received.
- **unregister:** A VM has unregistered with the controller.
- **start:** A VM has been started.
- **start_failure:** A VM start has failed.
- **start_request:** A request to start a list of VMs has been received.
- **state_change:** The state of a VM has changed.

Appendix C

Writing Usher Extensions

In this Appendix, We present the details of writing Usher extensions. Recall that there are three ways to extend Usher: clients, plugins, and VMM API wrappers. We discuss each of these in turn.

C.1 Clients

Usher client applications utilize the Usher client API (`usher/cli/api.py`) to interact with the Usher controller. Clients import `usher.cli.api`, then create an `usher.cli.api.API` instance for interaction with the controller.

After that, it is a simple matter of having your application use the methods provided by your `API` instance. See the docstrings in the `usher/cli/api.py` file for information on using these methods (signatures, parameter descriptions, etc).

Concerning configuration files for your clients, first decide upon an appropriate name for a section title for your application's configuration file. For example, *ush* uses "`ush`" for its configuration file section. Then, using the `config_skel.py` file included with the Usher client API source code, create a configuration file handler for your application by specifying the `section_title` variable and your configuration specification and saving it as `config.py` in your application's source directory.

See the the ConfigObj Validation section of the ConfigObj [cona] documenta-

tion for details on writing a specification. A specification is not required, but is probably a good idea. It is fairly straightforward to write one using the ConfigObj docs and examples.

With the above, configuration files will be read from:

- `/etc/usher/<section_title>.config` where "`section_title`" is whatever you named your section.
- `/.usher/<section_title>.config`
- anything pointed to by the environment variable `<SECTION_TITLE>_CONFIG` (where "`SECTION_TITLE`" is whatever you named your section capitalized)

To get both your application's and the client API's configuration sections as a dictionary of dictionaries, just call the `get_cfg` method without specifying a section. Then, to get options from a particular section, it is just a matter of indexing with that section first. For example, to get the value of the client API's `ctrl_host` option, use

```
cfg = config.get_cfg()
ctrl_host = cfg['cli']['ctrl_host']
```

If your section (named "`mysection`") has an option named "`myopt`":

```
myopt = cfg['mysection']['myopt']
```

If you only need options from a particular section, you can specify that section in the `get_cfg` call. Then, you only need the option as an index. For example, if your section (named "`mysection`") has an option named "`myopt`":

```
cfg = config.get_cfg('mysection')
myopt = cfg['myopt']
```

For additional details of writing Usher clients, please refer to the Usher source code.

C.2 Plugins

The `usher/utils/plugin.py` file provides the `UsherPlugin` class which all plugins must subclass. Check out this class to see what it does. The most important thing to see here is that subclasses of `UsherPlugin` must define the `entry_point` method.

When a plugin is registered for an event, an instance of its `UsherPlugin` subclass is placed on the callback list for that event. When the corresponding event fires, the `entry_point` method is called for each plugin on the callback list in order of priority (set by the order instance variable or order of registration). The arguments passed to the plugin are given in Appendix B (and also in the `usher/utils/events.py` file).

When the plugin completes execution, its `entry_point` method must return a tuple of the same type as it received. That is, the return type for the `entry_point` method must be the same as its arguments. This is necessary since this is passed to subsequent plugins on the same callback list. This is not to say that your plugin cannot modify these arguments (for mutable objects), or replace them (for immutable objects). On the contrary, being able to modify these arguments or return something different (but with the same type) is a very powerful feature of the callback list.

If your plugin will be registered for multiple events, the `entry_point` method will essentially be a dispatcher to the appropriate class method designed to handle the event. As a simple example, let's look at the `entry_point` method for the Usher DNS plugin:

```
def entry_point(self, *args):
    if self.event == 'register':
        vm = args[0]
        self.zone = vm.usherctrl.cfg.get('name_suffix')
        self.add_vm(vm)
    elif self.event ==  'unregister':
        vm = args[0]
        self.zone = vm.usherctrl.cfg.get('name_suffix')
        self.remove_vm(vm)
    return args
```

For the `register` and `unregister` events, a `VM` object is passed as the only item in the `args` tuple. The `entry_point` method calls the appropriate method based on the event for which its instance was registered. Also notice that this `entry_point` method simply returns the `args` tuple it received unmodified.

For a more complicated `entry_point` method example, see the Usher IP Management plugin.

Getting back to the `UsherPlugin` class, plugin writers also to set the following class variables in their `UsherPlugin` subclass:

- `name` - plugin name (often just `__name__`)
- `author` - plugin author
- `description` - brief description of the plugin (often just `__doc__` if you have added a docstring to your module)
- `version` - plugin version

Also, plugin writers should define their plugin's configuration specification in the `specification` variable at the top of their plugin module, or in the plugin's `__init__.py` file if the plugin is a package rather than a single file. See the ConfigObj Validation section of the ConfigObj documentation for additional details on writing a specification. A specification is not required, but is probably a good idea. It is also fairly straightforward how to write one using the ConfigObj docs and examples. Also, other plugins are a great place to look for sample specifications.

Plugins should also be shipped with a sample configuration file (at least in a README) which shows an example of registering the plugin and what events the plugin was written to handle. See the README that ships with the udns plugin for an example.

Optionally, plugins can create new events for which plugins can be registered. This is another powerful feature of plugins. To do this, simply create a dictionary named events in your plugin module or in your plugin packages `__init__.py` file with the

name of the events as dictionary keys, and a brief description of each event as values. Then, at the appropriate point in your plugin, call the controller's `fire_event` method, passing the new event name as the argument.

Two sample plugins have been included in the main distribution in the `usher/plugins` directory. The first `sample.py` is a single file plugin which writes a few messages to the controller's log file. In addition, this file creates a new event called "`sample_event`" which it fires after sleeping for five seconds. The second sample, sample2, is a plugin package illustrating multi-file plugin setup. This sample also merely prints a few messages to the controller's log file.

This should be all you need to get started writing your plugin. If your plugin needs to access or modify controller state, you will need to familiarize yourself with the controller code. Start by reading the Code Overview section above, then digging into the code and docstrings therein. Also, be sure to check out any existing plugins for additional examples. Plugins are typically short, single files, so going to the code is often the best course of action when you have questions.

Credential Checkers

Writing a credential checker is surprisingly easy. Credential checkers are simply plugins registered for the `client_authenticate` and `lnm_authenticate` events. These receive a tuple of the form: `(( username:str, password:str), valid:bool)` as input. Note that the return type of a plugin must be the same as its input type (see the Writing Plugins section above for information about writing plugins).

So, credential checkers merely need to return a tuple of type: `(( str, str), bool)` with the boolean field set to True for authentication successful, or False for authentication failed.

Caveats

When a plugin is registered, it is placed on a callback list for the event for which it was registered. Each of these callback lists runs in a separate thread so that slow plugins only slow down their callback chain (and not the entire controller). This also makes it easier for an administrator to identify slow, or broken plugins. This design does come at a cost.

Plugins are called in order on any given event callback list. However, since each list itself is run as a separate thread, there may be multiple event callback lists in execution at any one time. For this reason, plugins should be used with caution since they are free to perform arbitrary action, even actions which manipulate the internal state of the controller. In particular, proper locking should be considered when writing plugins which modify controller state.

Since most plugins do not directly modify controller state, the above is typically not a problem. Just be aware of it if you do write or use a plugin which does.

C.3 VMM Wrappers

Usher can be extended to support management of nearly any type of virtual machine. This is accomplished by wrapping the VM administration API of the underlying VMM to create an adapter for use by the Usher controller. The interface exposed by the adapter are the methods starting with "`remote_`" in the `usher/lnm/server.py` file. These methods, in turn, call corresponding secondary methods in the same file (usually of the same name with the "`remote_`" prefix stripped) in separate threads. In this way, LNMs do not become unresponsive to commands when one a previous command takes a long time to complete (e.g., migrate commands can sometimes take tens of seconds).

These secondary methods rely upon the existence of the methods in Table C.1 and Table C.2 in a file named <`vmm_name`>`_vmm.py`. These methods constitute the Usher VMM wrapper interface.

The “vmm_name” above is the name of the underlying VMM in lower case (e.g., `xen_vmm.py`). The VMM name is specified in the LNM’s configuration file to specify which wrapper should be used.

Table C.1 VMM wrapper VM methods.

Method	**Description**
start	start VM
migrate	migrate VM from this node
shutdown	soft shutdown VM
poweroff	hard shutdown VM
reboot	shutdown and start VM
cycle	poweroff and start VM
hibernate	hibernate a VM
restore	restart hibernated VM
pause	pause a VM (still in RAM)
resume	unpause VM
vmm_rename	rename a VM with VMM
get_status	get dictionary of attributes

Table C.2 VMM wrapper node methods.

Method	**Description**
get_current_stats	get list of all current stats
get_current_node_stats	get dictionary of current node stats
get_current_vm_stats	get dictionary of all current VM stats
get_node_info	get dictionary of static node attributes
get_details	list of dictionaries of all VM static attributes
get_vmm_states	dictionary of all VM states as seen by VMM

Notice that the VMM wrapper interface consists of two components: VM methods (Table C.1) and node methods (Table C.2). VM methods are members of the `VMOps` class. With the exception of the `get_status` method, all methods of Table C.1 manipulate VMs.

Methods which manipulate VMs either succeed, returning `None` (a Python type), or throw an exception with a descriptive message. Exceptions are logged and automatically propagated back to the controller by Twisted. The single VM query method,

`get_status`, returns a dictionary of VM attributes. See the code for what attributes are expected.

Node operations return dictionaries or lists of dictionaries of VM and node attributes. This allows the controller to query the state of all nodes and VMs under its management. These methods are used at controller startup to generate the global state of the system and periodically by the Monitor plugin to monitor resource usage.

Briefly, the `get_current_stats` call combines the information returned by both the `get_current_node_stats` and `get_current_vm_stats` methods which return 1, 5, and 15 minute resource usages for the physical machine and all VMs respectively. This call combines the two so that resource usage samples are taken at nearly the same time for both.

Refer to the code for what attributes are expected for each of these methods.

Bibliography

[ABD+07] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *Proceedings of the Twenty-first USENIX Large Installation System Administration Conference (LISA)*, November 2007.

[ACPtNT95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for Networks of Workstations: NOW. In *IEEE Micro*, February 1995.

[AFF+01] K. Appleby, S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Océano — SLA-based Management of a Computing Utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May 2001.

[ATSV06] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (SIGOPS-OSR)*, 40(1), January 2006.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, 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*, 2003.

[Beg06] Kyrre Begnum. Managing Large Networks of Virtual Machines. In *Proceedings of the 20th Large Installation System Administration Conference*, pages 205–214, 2006.

[BKSP04] G. Bruno, M. J. Katz, F. D. Sacerdoti, and P. M. Papadopoulos. Rolls: Modifying a Standard System Installer to Support User-customizable Cluster Frontend Appliances. In *IEEE International Conference on Cluster Computing*, 2004.

[BS99] Amnon Barak and Amnon Shiloh. Scalable Cluster Computing with MOSIX for Linux. In *Proceedings of Linux Expo 99*, pages 95–100, 1999.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM Computer Communications Review*, 33(3), July 2003.

[CDK+04] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, G. Marin, M. Mazina, J. Mellor-crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, and C. Mendes. New Grid Scheduling and Rescheduling Methods in the Grads Project. In *Proceedings of NSF Next Generation Software Workshop: International Parallel and Distributed Processing Symposium. Santa Fe, USA: IEEE CS*, pages 209–229, 2004.

[CFH+05] 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 Second ACM Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[CGK+06] Ludmila Cherkasova, Diwaker Gupta, Roman Kurakin, Vladimir Dobretsov, and Amin Vahdat. Optimising Grid Site Manager Performance with Virtual Machines. In *Proceedings of the 3rd USENIX Workshop on Real Large Distributed Systems (WORLDS)*, 2006.

[CIG+03] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[cona] Configobj. `http://www.voidspace.org.uk/python/configobj.html`.

[conb] Convirt. `http://xenman.sourceforge.net/`.

[CZBL00] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.

[D07] Gyrgy Dsa. The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I)11/9 OPT(I)+6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer Berlin / Heidelberg, 2007.

[ec2] Ec2. `http://aws.amazon.com/ec2/`.

[eno] Enomaly. `http://www.enomaly.com/`.

[euc] Eucalyptus. http://eucalyptus.cs.ucsb.edu/.

[Eus07] Alan Eustace. Personal communication, February 2007. Senior VP of Engineering, Google.

[fir] Firefox. http://www.mozilla.com/firefox/.

[FK98] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[GIYC06] Laura Grit, David Irwin, Aydan Yumerefendi, and Jeff Chase. Harnessing Virtual Machine Resource Control for Job Management. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W H Freeman and Company, 1979.

[Gra69] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[Ham07] Jeff Hammerbacher. Personal communication, October 2007. Engineering Manager, Facebook.

[HBD95] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, page 236, New York, NY, USA, 1995. ACM.

[ipv] Indexable ipv4 range extension. http://pgfoundry.org/projects/ip4r.

[JCH84] R. Jain, D.M. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems. Technical Report TR-301, DEC, 1984.

[Kel03] Terence Kelly. Utility-directed Allocation. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[Kel04] Terrance Kelly. Generalized Knapsack Solvers for Multi-unit Combinatorial Auctions. Technical Report HPL-2004-21, HP Labs, 2004.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[Kis07] Jay Kistler. Personal communication, November 2007. VP of Engineering, Yahoo!

[KKP+06] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic Placement for Clustered Web Applications. In *Proceedings of the 15th International Conference on World Wide Web*, pages 595–604. ACM, 2006.

[kod] Kodiak. `http://www.bluebearllc.net/kodiak/`.

[Kur] Roman Kurakin. Personal communication. Email dated 5/10/2007.

[KUS+04] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. SoftUDC: A New Adaptive Management Paradigm. *IEEE Computer*, November 2004.

[kvm] KVM - Kernel-based Virtual Machine. `http://kvm.qumranet.com/`.

[Law01] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Courier Dover Publications, 2001.

[lbv] load balancing of virtual machines. `http://lbvm.sourceforge.net/`.

[lcg] LCG project. `http://lcg.web.cern.ch/LCG/`.

[LGW04] Hui Li, David Groep, and Lex Wolters. Workload Characteristics of a Multi-cluster Supercomputer. In *Job Scheduling Strategies for Parallel Processing*, pages 176–193. Springer Verlag, 2004.

[lib] libvirt. `http://libvirt.org/`.

[liv] LiveCapacity. `http://virtualiron.com`.

[LO86] Will Leland and Teunis J. Ott. Load-balancing Heuristics and Process Behavior. *SIGMETRICS Performance Evaluation Review*, 14(1):54–69, 1986.

[Mer] Phil Merkey. Beowulf History. `http://www.beowulf.org/overview/history.html`.

[MRR+53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[NS97] Mark Nuttall and Morris Sloman. Workload Characteristics for Process Migration and Load Balancing. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 133, Washington, DC, USA, 1997. IEEE Computer Society.

[NS07] Ripal Nathuji and Karsten Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proceedings of Twenty-first ACM Symposium on Operating Systems Principles (SOSP)*, pages 265–278, New York, NY, USA, 2007. ACM.

[OCP+06] David Oppenheimer, Brent Chun, David Patterson, Alex C. Snoeren, and Amin Vahdat. Service Placement in a Shared Wide-area Platform. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 2006. USENIX Association.

[O'H08] David O'Hallaron. Personal communication, July 2008. Director of Intel Research Pittsburgh.

[pos] PostgreSQL. `http://www.postgresql.org/`.

[pyt] Python. `http://www.python.org/`.

[RRX+06] P. Ruth, Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *IEEE International Conference on Autonomic Computing (ICAC'06)*, June 2006.

[SCB04] F. D. Sacerdoti, S. Chandra, and K. Bhatia. Grid Systems Deployment and Management Using Rocks. In *IEEE International Conference on Cluster Computing*, 2004.

[SY99] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society, 1999.

[sys] System center. `http://www.microsoft.com/systemcenter/`.

[tas] Tashi. `http://incubator.apache.org/tashi/`.

[TC00] Kenjiro Taura and Andrew Chien. A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 102, Washington, DC, USA, 2000. IEEE Computer Society.

[twi] Twisted. `http://twistedmatrix.com/`.

[VD00] Werner Vogels and Dan Dumitriu. An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2000.

[VD02] Sathish S. Vadhiyar and Jack J. Dongarra. A Framework For Migrating Applications Under Changing Load Conditions in the Grid, 2002.

[VD03] Sathish S. Vadhiyar and Jack J. Dongarra. A Performance Oriented Migration Framework For the Grid. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 130, Washington, DC, USA, 2003. IEEE Computer Society.

[vmwa] VirtualCenter. `http://www.vmware.com/products/vi/vc/`.

[vmwb] VMware Workstation. `http://www.vmware.com/products/ws/`.

[Wal02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[WSVY07] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007.

[xen] XenEnterprise. `http://www.xensource.com/products/xen_enterprise/`.

[xvm] xVM. `http://www.sun.com/software/products/xvmopscenter/`.

[ZKY96] Lotfi Asker Zadeh, George J. Klir, and Bo Yuan. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems*. World Scientific, 1996.