5-2010

Virtual Organization Clusters: Self-Provisioned Clouds on the Grid

Michael Murphy

Clemson University, mamurph@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the [Computer Sciences Commons](#)

Recommended Citation

Murphy, Michael, "Virtual Organization Clusters: Self-Provisioned Clouds on the Grid" (2010). *All Dissertations*. 511.
https://tigerprints.clemson.edu/all_dissertations/511

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

VIRTUAL ORGANIZATION CLUSTERS: SELF-PROVISIONED CLOUDS ON THE GRID

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Michael A. Murphy
May 2010

Accepted by:
Dr. Sebastien Goasguen, Committee Chair
Dr. Jim Martin
Dr. Walt Ligon
Dr. Robert Geist

Abstract

Virtual Organization Clusters (VOCs) provide a novel architecture for overlaying dedicated cluster systems on existing grid infrastructures. VOCs provide customized, homogeneous execution environments on a per-Virtual Organization basis, without the cost of physical cluster construction or the overhead of per-job containers. Administrative access and overlay network capabilities are granted to Virtual Organizations (VOs) that choose to implement VOC technology, while the system remains completely transparent to end users and non-participating VOs. Unlike alternative systems that require explicit leases, VOCs are autonomically self-provisioned according to configurable usage policies. As a grid computing architecture, VOCs are designed to be technology agnostic and are implementable by any combination of software and services that follows the Virtual Organization Cluster Model.

As demonstrated through simulation testing and evaluation of an implemented prototype, VOCs are a viable mechanism for increasing end-user job compatibility on grid sites. On existing production grids, where jobs are frequently submitted to a small subset of sites and thus experience high queuing delays relative to average job length, the grid-wide addition of VOCs does not adversely affect mean job sojourn time. By load-balancing jobs among grid sites, VOCs can reduce the total amount of queuing on a grid to a level sufficient to counteract the performance overhead introduced by virtualization.

Dedication

This dissertation is dedicated to my grandparents: Marge Tyner, Jim and Edrie Murphy, and the late Dr. Calvin Tyner. Throughout my entire academic career, from grade school through university, my grandparents have provided the encouragement and support that has made this journey to the doctorate possible, and to them I am exceedingly grateful.

Acknowledgments

I would like to express my sincere thanks to my research committee for their assistance with this project. My advisor, Prof. Sebastien Goasguen, has spent countless hours assessing my work, providing assistance with the research, and reading my long-winded e-mail messages. His patient guidance throughout this process, along with his encouragement to publish results in challenging venues, has inspired me to pursue loftier objectives in my research. Prof. Jim Martin, Prof. Walt Ligon, and Prof. Robert Geist have assisted in the process by dedicating their time and expertise to reviewing and aiding in the improvement of my research and its presentation.

Outside my committee, I would also like to thank the faculty members who have assisted me with research guidance, pursuit of funding, and day-to-day administration issues. Prof. Jason Hallstrom, Prof. Chris Post, and Prof. Steve Stevenson have guided my past research projects, advised me on career and professional issues, and were instrumental in my securing the NSF Graduate Research Fellowship. Prof. Wayne Madison not only assisted me with the fellowship pursuit but also contributed significant time and effort to handling administrative issues, advising me throughout my undergraduate and graduate careers, and providing me with teaching opportunities in the School of Computing. Prof. Pradip Srimani graciously reviewed paper submissions and mathematical estimation work, providing helpful comments for improvement of the material. The late Prof. Per Brinch Hansen of the Syracuse University Department of Electrical Engineering and Computer Science encouraged me with insights in distributed computing and computational problem solving, which greatly shaped my approaches to the research problems described in this dissertation.

For their assistance with numerous administrative and technical issues, I would like to thank the staff of the Clemson University School of Computing. Scott Duckworth and Nell Kennedy have provided exceptional assistance with the provisioning of networking and other services required to support our prototype systems, hardware and vendor advice, and assistance with hardware orders.

April Bowen, Lea Benson, and Kim Keasler have generously provided their assistance with school and university policy compliance, fellowship administration, facilities support, and other administrative issues.

Throughout the process of researching Virtual Organization Clusters and preparing this work, I have been privileged to work with a team of graduate and undergraduate students, who have assisted in the collection of data, administration of computer systems, and co-authoring of papers. I would like to express my sincere appreciation to Linton Abraham, Michael Fenn, Lance Stout, Brandon Kagey, Joshua Canter, and Emmett McQuinn for their contributions to this research project and to the publications that have resulted from it.

During my entire academic career, I have been supported by my relatives and friends as I have pursued challenging and often time-consuming courses of study. I would like to thank my grandparents, to whom this dissertation is dedicated, my parents – Mark Murphy and Ginny Murphy, and the remainder of my family for their support and encouragement. I also owe a debt of gratitude to all my friends through graduate school, including Dru and Heather Sepulveda, Katelyn Howay, Shawn Becker, Tim and Kristen Rabideau, James Bardoner, Claire Wellborn, Sarah Peck, Jordan and Shana Upham, Ben Sterrett, Christine Marusich, Myric Lehner, and Shawna Martell.

Simulation data sets used in this work have been provided by the Grid Observatory (grid-observatory.org). The Grid Observatory is part of the EGEE-III EU project INFSO-RI-222667.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Title Page	i
Abstract	ii
Dedication	iii
Acknowledgments	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Vision	2
1.2 Motivation	5
1.3 Thesis Statement	12
1.4 Dissertation Organization	13
2 Grid Systems	15
2.1 Computational Grids	16
2.2 Composition of Grids	18
2.3 Virtual Organizations	20
2.4 Computational Jobs	22
2.5 Service-Oriented Architecture	24
2.6 Cloud Computing Systems	25
3 Operating System Virtualization	29
3.1 Levels of Virtualization	31
3.2 Instruction Set Architecture Requirements	32
3.3 Dynamic Recompilation	34
3.4 Paravirtualization	35
3.5 Hardware Virtual Machine Extensions	38
3.6 Applications of Virtualization Technology	41
4 Related Work	44
4.1 Virtualization of Grid Systems	44
4.2 Simulation Systems	55
4.3 System Resources	58
4.4 System Management	64
5 Virtual Organization Cluster Model	68
5.1 Separation of Administrative Domains	69

5.2	Terminology	72
5.3	Definition of a Virtual Organization Cluster	74
5.4	Self-Provisioned Clouds	77
6	Simulator for Virtual Organization Clusters	79
6.1	Motivation for Simulation	80
6.2	Grid Trace Data	83
6.3	Simulator Design	84
7	Prototype Implementation	90
7.1	Cluster Setup	91
7.2	Dynamic Provisioning of Virtual Organization Clusters	94
7.3	Management of Systems	97
8	Simulation Test Results	103
8.1	Preliminary Simulations	104
8.2	Input Data Processing	106
8.3	Control Simulation	106
8.4	Standard Grid Architecture	107
8.5	Virtual Organization Cluster Architecture	109
8.6	Effects of Virtualization Overhead	111
9	Prototype Test Results	117
9.1	Performance Tests	117
9.2	Dynamic Provisioning System	127
9.3	Overlay Scheduling	133
9.4	Operational Tests	138
9.5	System Management with Stoker	140
10	Conclusions	145
10.1	Utility and Viability of Virtual Organization Clusters	147
10.2	Direct Impacts	148
10.3	Broader Impacts	148
10.4	Future Work	151
	Appendices	153
A	Electronic Attachments	154
B	License Agreements	155
	Bibliography	161

List of Tables

3.1	Levels of Virtualization	32
3.2	Common Linux Virtualization Systems	41
5.1	Examples of VOC Component Technologies	77
8.1	Simulation Results	114
9.1	Observed Boot Times	119
9.2	Performance Comparison Between Slackware Linux 12 and CentOS 5.1	120
9.3	Comparison of VOC and Physical System Performance	120
9.4	Compute-Bound Virtualization Overheads	122
9.5	Network-Bound Virtualization Overheads (Dual-Core)	122
9.6	Network-Bound Virtualization Overheads (Single-Core)	123
9.7	Bandwidth and Latency on Physical and Virtual Clusters	126
9.8	Effects of Overlay Scheduling	135

List of Figures

1.1	Vision of a Virtual Organization Cluster	3
1.2	Traditional Grid Systems	6
1.3	Virtualized Grids	8
1.4	Leasing Architectures	9
1.5	Virtual Overlay Cluster on a Grid	11
2.1	Traditional Grid Layers	17
2.2	Cluster Systems	18
2.3	Grid Systems as Networks of Clusters	19
2.4	Abstract View of Grid Systems	19
2.5	Virtual Organizations on the Grid	21
2.6	Regular Jobs	22
2.7	Pilot Jobs	23
2.8	Service Oriented Architecture	24
2.9	Cloud Computing	27
3.1	Virtual Machines	30
3.2	Popek and Goldberg Virtualization Requirements	33
3.3	Virtualization with Dynamic Recompilation	34
3.4	Type I Paravirtualization System	36
3.5	Full Virtualization with Hardware Support	39
3.6	Staged Virtual Machine Images	40
3.7	Non-Staged Image in KVM Snapshot Mode	40
4.1	Leasing Model	46
4.2	Autonomic Model	51
5.1	Administrative Domains	70
5.2	Virtual Organization Clusters on a Grid System	76
6.1	High-level components of SimVOC	85
7.1	Cluster Construction Process	92
7.2	Initial Test Cluster Setup	93
7.3	Watchdog Daemon Functions	95
7.4	Overview of Stoker	99
7.5	Integration of Stoker with DHCP and DNS Services	101
7.6	Overview of Pulley	102
8.1	Simulation of 2 VOCs Sharing 16 CPUs	105
8.2	Simulation of 2 VOCs Sharing 4 CPUs	105
8.3	Control Simulation	107

8.4	Trace of the Control Simulation	108
8.5	Standard Architecture Simulation	109
8.6	Trace of the Standard Architecture Simulation	110
8.7	Representative Trace of a VOC Simulation	112
8.8	KVM Simulation without Target Level	112
8.9	KVM Simulation with Target Level 1024	113
8.10	Xen Simulation without Target Level	113
8.11	Xen Simulation with Target Level 1024	115
9.1	Average Boot Times	119
9.2	Virtualization Overheads for Compute-Bound Processes (Single-Core)	121
9.3	Virtualization Overheads for Network-Bound Processes (Single-Core)	121
9.4	Virtualization Overheads for Network-Bound Processes (Dual-Core)	124
9.5	Virtual Machine Bridged Networking	125
9.6	Prototype System Architecture	127
9.7	Virtual Machine Boot Delays as Recorded by Condor	129
9.8	Direct Execution of Batches of Jobs	131
9.9	Long-Period Direct Submission of 10 Jobs (10 Second Length)	131
9.10	Short-Period Direct Submission of 10 Jobs (10 Second Length)	131
9.11	Short-Period Direct Submission of 10 Jobs (1 Second Length)	131
9.12	Globus Execution of Batches of Jobs	132
9.13	Long-Period Job Submission Through Globus	132
9.14	Short-Period Job Submission Through Globus (10 Second Jobs)	132
9.15	Short-Period Job Submission Through Globus (1 Second Jobs)	132
9.16	Transparent VOC Performance without Overlay Scheduling	135
9.17	Participating VOC Performance with Overlay Scheduling	136
9.18	Performance of the Simple Greedy VOC Watchdog Algorithm	137
9.19	Performance of the Delayed Response VOC Watchdog Algorithm	137
9.20	Short Operational Test	138
9.21	Transparent Operational VOC: Engage VO	139
9.22	Transparent Operational VOC: NanoHub VO	140
9.23	Stoker Performance on a Parallel Hostname Job	141
9.24	Stoker Performance on a Parallel Sleep Job	142
9.25	Overhead of Networking and SSH in Stoker	142
9.26	Stoker on a High Latency Network	143
9.27	Stoker SSH and Network Overheads on a High Latency Network	144
10.1	Revisiting the Virtual Organization Cluster Architecture	146

Chapter 1

Introduction

Grid computing systems enable users associated with a variety of application domains to run large-scale computational tasks that would overwhelm the capabilities of locally available hardware. In order to utilize these systems, users create and submit batch jobs to grid sites, which schedule the jobs for execution. Users, jobs, and computational resources are connected together via *middleware*, or software layers that facilitate interaction between different parts of the grid system. At a high level, the combination of middleware, resources, and utilization methodology employed on a grid system may be called a *grid architecture*. Existing grid architectures generally require the user to select computational resources directly and manage issues of job compatibility between resources manually, resulting in a system that is neither easy to use [33] nor truly transparent to the user in distributed computing terms [53]. Moreover, since users typically favor a small subset of grid sites when submitting jobs, the grid is not effectively load-balanced, leading to longer queuing delays while jobs wait for computational resources to become available.

Virtualization technology has been proposed as a mechanism for improving the usability of grid computing systems by providing environment customization to users or groups of users. By using virtual machines to run user jobs, the environment in which jobs execute becomes decoupled from the underlying hardware, allowing any physical site with a compatible Instruction Set Architecture (ISA) to supply any variety of code libraries required for current or legacy applications. Each environment becomes an isolated container for user applications, providing better security and resource control than is available with traditional grid systems. [57] To date, a key limitation in the deployment of virtual environments for grid computing has been middleware integration and the

paradigm selected for user interaction. In particular, existing grid virtualization systems, such as Shirako [78], Cluster on Demand [32], and Globus Nimbus [94], are designed around leasing models that require the user to make explicit resource reservations. As is often the case with traditional grid architectures, these lease-oriented systems compel the user to perform resource management tasks that otherwise could be automated by the system.

This dissertation presents a novel grid computing architecture that is designed to deliver the benefits of virtualized environments while simplifying grid utilization for the end user. The Virtual Organization Cluster Model (VOC Model) describes a grid architecture that provides execution environments that are homogeneous across grid sites, allowing applications to be run on any ISA-compatible resource, including those resources that do not provide the necessary software libraries to run user jobs directly on the hardware. VOCs remain transparent to end users and disinterested entities, permitting the new architecture to be deployed in conjunction with, and without disrupting, existing production grids. As demonstrated through simulation testing, large-scale deployment of VOCs would provide the benefits of virtualized grid systems without adversely affecting aggregate grid performance, since the overhead of virtualization systems would be offset by increased execution parallelism, leading to a reduction in total job queuing.

The remainder of this chapter introduces the reasoning behind Virtual Organization Clusters and the outcomes to be measured when VOCs are implemented and tested. Section 1.1 presents the vision for VOCs, after which a more detailed description of the motivation for creating this autonomic grid architecture is discussed in section 1.2. Specific features of the system to be demonstrated are presented in section 1.3. Finally, the organization of the rest of the dissertation is described in section 1.4.

1.1 Vision

Virtual Organization Clusters (VOCs) are autonomically managed virtual environments that are homogeneous across sites, transparent to end users, implementable in a phased and non-disruptive manner, optionally customizable by Virtual Organizations, and designed according to an architectural specification. As implied by the nomenclature, VOCs are *clusters* in a computational sense: each VOC can be represented as a star topology with a single head node and a collection of compute nodes. User jobs arrive at the head node and are scheduled to execute on the compute

Figure 1.1: Two Virtual Organization Clusters on two different physical grid sites. Of the two Virtual Organizations, VO₁ does not utilize the VOC technology; as a result, the use of virtual machines is completely transparent to this VO. In contrast, VO₂ participates in the VOC technology and has its own private resources, a private overlay network, and even control over job scheduling on its private cluster. Both VOCs are completely transparent to end users, who submit jobs through existing, unmodified grid middleware. User jobs are routed either to a physical cluster or to a VOC through a Compute Element (CE) interface between the grid and each cluster.

nodes by means of a private scheduler belonging to the virtual cluster. Like any computational cluster, VOCs may be administered by means of distributed management tools that execute at run time.

Unlike traditional computational clusters, VOC compute nodes are ephemeral, since they are spawned from virtual machine instances that are scheduled on underlying physical hardware. In addition, the network links between the nodes in a VOC are overlays on top of a Wide Area Network with heterogeneous link capabilities between different systems, rendering VOCs less desirable for High Performance Computing (HPC) applications. For High Throughput Computing (HTC) and Many Tasks Computing (MTC) applications, however, VOCs provide *cloud* environments: software systems that can be hosted entirely by off-site computational resources and accessed exclusively through the network [163]. These clouds are *autonomically managed* by middleware, implying that software agents manage, configure, and repair the virtual clusters without human involvement [96]. Furthermore, the software agents also adjust the size of VOCs in response to changing workload demands and resource availability, resulting in a *self-provisioned* system.

Following the model of grid computing presented in Foster et al., 2001 [66], Virtual Organization Clusters are dedicated to (and in non-transparent cases, owned by) a single *Virtual Organization* (VO), which is a collection of entities with a common purpose. In the case of VOCs, these VOs represent the end users, who are assumed to be domain experts, not computer scientists. VOs may employ computing professionals, however, to design and administer custom VOCs.

Since design decisions and issues in a grid system are by definition distributed and at scale, abruptly changing the architecture of an existing production grid would be impractical in terms of cost and complexity. Thus, it is necessary for a new grid architecture to be deployable in a phased manner that does not cause widespread disruption. As illustrated in figure 1.1, VOCs achieve phased deployability by leveraging existing middleware and permitting *participating* and *non-participating* (or *transparent*) deployments. In a participating deployment, a VOC is created for a single VO, which assumes administrative responsibility for the VOC. All participating grid sites must permit virtual machines to be spawned to create VOC compute nodes. Conversely, in a non-participating deployment, a single grid site transparently provides VOCs for VOs authorized to run jobs on the site, encapsulating the end-user workloads in local virtual environments. These non-participating VOCs are managed by the site administrators, and it is not necessary that VOs or end users even be made aware that VOCs are in use. In either case, middleware changes are needed only on the

sites that will host virtual machines. No changes to submission endpoint middleware (the software operated by the end user) are required.

Participating VOCs achieve cross-site homogeneity by utilizing virtualization systems to separate job execution environments from the underlying hardware. Each participating VOC provides a virtual cluster head node, which is exposed to the grid as a standard grid site. Users affiliated with the Virtual Organization that owns the participating VOC then submit jobs to this one special VOC-specific site, instead of selecting one of a potentially large number of possible target sites at job submission time. By freeing the user from the requirement to select computational resources manually for each task, the grid system becomes more accessible to the user.

1.2 Motivation

Existing grid systems require the user to adapt his or her application to run in the environment provided at each grid site, and they require the grid site system administrators to install software to support specific user groups or to maintain a common software stack. Users are constrained to specific and possibly outdated application support libraries, and administrators face the challenge of trying to support multiple user groups with potentially conflicting software requirements. Software homogeneity across grid sites, while a desirable property, is essentially impossible due to hardware constraints, site policies, and cost constraints. [47] As a result, users encounter different software stacks on different grid sites, and they must ensure that jobs are submitted to grid sites containing the libraries necessary for job execution.

Requiring users to target specific sites when submitting jobs to a grid system results in a less transparent [53] system than would otherwise be possible with an autonomic system that selects target sites for jobs automatically. Although metascheduling systems [10, 139] can perform automatic site selection, these grid-level schedulers still require the user to specify extended information about the nature of the job and its requirements. As is the case with any scheduling system that matches requests to providers based on explicit information provided by either party, the omission of important information could result in failure to match a job to a target site. Alternatively, the job could be matched to a site that does not have the necessary software environment for actual job execution, since the matching requirements could be erroneous or absent.

Existing grid systems require each physical site to maintain application support libraries as

Figure 1.2: In current grid systems, the user queries collective brokering services (1) to obtain an interface to an abstract computational resource (2). The user submits jobs directly to these abstract resources (3), which are actually software layers that communicate with actual physical resources using a set of well-defined network protocols (4). In this example, the Static Application is compiled statically with the Code Library and can run on any of the three sites in the Physical Fabric Layer. However, the Dynamic Application is supported only on Site 3, since the other two sites do not have the Code Library installed.

part of the local software stack. Since the number of applications that grid users desire to execute is always increasing [63], the software stacks available at different sites may differ greatly in terms of library support. When grid systems mandate common software stacks, user applications are limited to programs that can be supported by the common system, which is likely to be outdated [47]. Consider a dynamically compiled user application that depends upon an external code library. In the simple 3-site grid in figure 1.2, the user application is executable only on Physical Site 3, since the required code library is missing from the other two sites. Assuming the user manually and randomly submits her job to a site without *a priori* information about installed software at each site, she will have a 33% chance of selecting the site capable of running her application. If she uses a metascheduler instead of submitting her job directly, the metascheduler *might* select the proper site automatically, provided the metascheduler matching criteria are set correctly in the user's job description *and* in each site's local description. If the system administrator at Site 3 failed to add a data tag indicating the presence of the code library, or the user failed to require the code library in her job submission, then a failure would occur. Alternatively, the user could submit the application along with the code library, statically compiled into a single executable. However, for a large code library shared among several applications, this approach may not be practical due to the amount of data that would have to be transferred between sites.

A solution to both site heterogeneity and outdated per-site software on the grid is to provide virtual environments containing the software necessary to support the desired applications. Virtualization of grid systems has been proposed as a mechanism for providing custom environments to grid users, without imposing undue burdens on system administrators [57]. However, virtualized grid systems to date have taken the approach that new middleware should be developed for the leasing of physical resources on which to run virtual containers, since the existing middleware systems do not have native support for starting virtual machines (figure 1.3). The most widely published systems – Virtual Workspaces [94], In-VIGO [7], and Shirako [83] – all require the addition of system-specific middleware at both the execution and submission endpoints to create leasing architectures (figure 1.4). In some cases, these systems require replacement of entire middleware stacks. With such requirements imposed on both the hosting site and the user, these lease-oriented systems are not transparent and cannot be deployed in a non-disruptive fashion. Moreover, the user is responsible for instantiating his or her desired environment manually, which does not address the usability issue. It is perhaps for these reasons that leasing systems have not seen widespread use.

Figure 1.3: In a virtualized grid, the user performs job submission (steps 1, 2, and 3) using exactly the same procedures used in the current grid architecture (figure 1.2). However, the applications are executed in virtual environments hosted by the physical sites, instead of directly on physical sites. An issue with this architecture, as illustrated in step 4, is the method by which the virtual machines are started on the physical systems.

Figure 1.4: In a leasing architecture, the user queries the collective brokering services (steps 1 and 2) to locate a resource that supports virtualization. The user initiates a lease through middleware, which starts a virtual machine (step 3). Once the virtual machine has booted, the user interacts with it directly, running the application in a manner similar to execution directly on a local system (step 4). Note that the grid middleware is bypassed in the last step, violating the transparency principle from [53].

If each user were to create a custom virtual environment – on the order of 10 GB – for each application, and there were 1,000 replicas of a single application running on 1,000 different systems, a total of 10 TB of data would have to be transferred to instantiate the environment across all virtual machine hosts. Since transferring even one 10 GB virtual machine image over the Internet on a frequent basis would be impractical, it would be desirable for virtual environments to be created and administered by groups of users with similar interests, known as Virtual Organizations (VOs). Jobs submitted by VO-affiliated users could be scheduled to execute in the virtual environments provided by the VO. By utilizing this model, the example user’s dynamic application becomes executable on all sites, since actual job execution occurs within a virtual machine instantiated on the site. The virtual containers in this case are independent of the underlying hardware and effectively isolate end user jobs from physical systems [57]. Moreover, moving the application layer software libraries into the virtual systems eases administration for all groups on the grid, since a division of labor is created between the virtual system administrators and the physical site administrators [69]. By requiring that all virtual machine instances be spawned from copies of the same VM image, management of the virtual environments is greatly simplified. Unfortunately, the addition of virtualization alone is not sufficient to solve the problem of grid system usability, since the virtual environments must be scheduled and run on the physical systems.

Improving ease of use for the end user necessitates the implementation of an autonomic [96] middleware system to manage the virtual machine instances in which user jobs are executed. Since these instances are spawned from the same virtual machine image, they are effectively homogeneous from a software compatibility perspective. By adding an overlay network to the set of virtual environments, a virtual cluster system is created. Private overlay scheduling is used to execute jobs submitted to a dedicated server that functions as a head node for the virtual cluster. Since the set of virtual resources is dedicated to, and managed by, a single Virtual Organization, this cluster is called a Virtual Organization Cluster (VOC).

Virtual Organization Clusters improve the usability of the grid by providing a dedicated site for the Virtual Organization that supplies the VOC. As illustrated in figure 1.5, the user submits her application through the grid middleware, targeting the dedicated server provided by the VO. The VO Server functions as a head node for the VOC and schedules the user’s jobs using an overlay scheduler and a private overlay network that connects the head node to the compute nodes. Since the VO Server manages the virtual machine instances, the user is liberated from the task of managing

Figure 1.5: Virtual Overlay Cluster on a Grid. In this use case, the user queries collective brokering services to obtain a reference to the dedicated VO server, to which she submits her job. The VO Server autonomically starts virtual machines using pilot jobs [143] on the grid, then schedules and executes the user’s job through an overlay network with an overlay scheduler. Since the VM image contains the code library, the dynamic application is supported.

computational resources. Moreover, since the user always submits her jobs to the dedicated site provided by the VO with which she is affiliated, she is no longer required to select from a set of sites each time she submits a job. Instead, the VOC middleware handles site selection by starting virtual machines on grid sites and automatically routing jobs to the dynamically instantiated environments. In addition, since the VO represents the interests of the user, it should be easier for her to request installation of specific code libraries and tools required to run her application.

Two other important issues are illustrated in figure 1.5: homogeneity of the execution environments and interoperability with existing middleware. Since each VM instance is spawned from the same VM image, all compute nodes are homogeneous from a software compatibility perspective.

These VM instances are started through the use of *pilot jobs*, which effectively reserve a physical grid resource and then start another process on that resource by means of a request to an external server [143]. Pilot jobs are managed autonomically by the VOC middleware installed on the dedicated VO server system, and they are submitted through the *existing* middleware already in use on the grid, such as Globus [64]. Each physical site simply exposes a virtualization interface to the pilot job, which permits VM instances to be started or stopped. VM instances remain associated with their respective pilot jobs, permitting local site utilization policies to be enforced against the pilot jobs in the same way they are enforced against any other job. If a pilot job is killed by the site, its associated VM instance is terminated immediately.

Support for VOC implementations requires the addition of a virtualization interface on physical grid sites, as well as the creation of a dedicated VO server system with VOC management middleware installed. Importantly, no changes are required to the submission endpoint middleware installed by end users: users submit their grid jobs to VOCs in exactly the same way they submit grid jobs to physical sites. All required changes to the system are transparent to the users. Furthermore, the addition of a virtualization interface to one site does not affect the operation of other sites on the grid. Nor does the addition of a dedicated VO server for VOCs affect any other VOs on the grid. Thus, VOCs may be deployed incrementally, without disrupting the overall operation of existing production grids. Entities that choose not to support VOCs may continue to operate and coexist on the grid.

Finally, Virtual Organization Clusters are designed to be technology-agnostic, permitting the use of different virtualization technologies, networking systems, and computational grids. Since these technologies all change rapidly, VOCs are defined as an architecture, not as a specific implementation. The requirements and properties of all Virtual Organization Clusters are specified by the Virtual Organization Cluster Model (VOC Model).

1.3 Thesis Statement

Virtual Organization Clusters are dynamically self-provisioned virtual software environments that can improve usability of grid systems by enabling computational environments to be matched to the needs of the user, instead of compelling the user to adapt his or her application to the software available on the grid. The addition of VOCs to existing production grids does not

necessarily reduce aggregate performance, since the overhead introduced by virtualization systems can be offset by a reduction in average job waiting time. VOC implementations can load balance user jobs across a greater number of CPU cores than existing physical grid systems, since the overlay environments can make each CPU core compatible with each user job, eliminating the need for users to target specific grid sites to ensure job compatibility. Unlike leasing systems, users are not tasked with managing virtual machines themselves. Instead, autonomic middleware adjusts resource allocations in response to changing demands, allowing the user to focus on his or her particular domain application.

Although this document divides the theoretical framework and empirical results into separate chapters, the actual research process used to create this dissertation was not an empirical validation of existing theory. Instead, this process utilized an exploratory research model [2] to develop a novel architecture for virtualizing grid systems by way of a “middle-out” design. Unlike related grid virtualization projects (section 4.1), the objective of the Virtual Organization Cluster Model was to specify properties common to all VOC implementations, as opposed to providing a single implementation framework, in order to reach as broad an audience as possible by providing implementation flexibility. Thus, the primary objectives of the VOC research were to define the model and then demonstrate that VOCs implemented according to the model could be viable additions to existing production grids.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of grid computing systems, followed by a discussion of virtualization systems in chapter 3. These topics provide a framework for the discussion of virtualized grid architectures. Related works are then discussed in chapter 4.

Following the related work, a discussion of the design of Virtual Organization Clusters is presented in chapter 5. The SimVOC simulator, designed to evaluate VOC properties using actual grid trace data, is described in chapter 6. Descriptions of the components of a physical prototype implementation of an Open Science Grid site with associated non-participating VOCs are provided in chapter 7. Empirical test results are presented in chapters 8 and 9, using the SimVOC simulator and a prototype implementation, respectively. Finally, chapter 10 concludes the main portion of the

dissertation.

Appendices to the dissertation describe the software and data sets that accompany the published version of the document. Due to the considerable length of both the software code listings and software documentation provided, these resources are provided electronically. Appendix A lists the electronic attachments that accompany the dissertation. License agreements that apply to these attachments are included in appendix B.

Chapter 2

Grid Systems

Grid systems, such as the Open Science Grid (OSG) [125], enable different entities to share computational resources using a flexible and secure framework. These resources enable users to run computational jobs that exceed the capabilities of the systems available at any single location. To mitigate compatibility issues that result from resource heterogeneity, computational grids could be designed as Service Oriented Architectures, in which grid-enabled applications do not interact with low-level systems and resources directly. Instead, these applications communicate with abstract service libraries, which are built on top of standard network communications protocols such as TCP and IP. The services layer acts as a high-level operating system for the underlying physical resources, providing both abstraction and arbitration functionality. Since co-scheduling physical resources across sites is a complex task, such grid services are well-adapted for High Throughput Computing (HTC) applications, which tend to be compute-bound and do not require completion deadlines. [63] Bag-of-tasks applications, in which multiple parallel tasks are executed independently, are a class of HTC applications especially well-suited to the grid environment [17]. As such, grids do not replace High Performance Computing (HPC) systems and supercomputers [66] but can be composed of such systems.

In this chapter, an overview of grid computing is presented. Section 2.1 describes the high-level architecture of computational grids, after which section 2.2 discusses the components that comprise a grid system. Virtual Organizations, the units around which grid transactions are coordinated, are presented in section 2.3. Workloads submitted by users, in the form of computational jobs, are discussed in section 2.4. Section 2.5 presents a vision of a grid system designed as a Service-

Oriented Architecture, where all grid-level operations are effected through services with well-defined APIs. When services are utilized to create autonomically managed, dynamically provisioned execution environments, the resulting systems are examples of cloud computing, as described in section 2.6.

2.1 Computational Grids

Grid computing systems exist to permit different entities to share computational resources in a well-defined and flexible manner. The sharing of computational resources in this context is “more than simply document exchange” [66]: it is the provisioning of entire computing systems so that third parties may run computational tasks. Providing local computing services to external third parties requires a federated trust model, in which parties are organized into representative units known as Virtual Organizations (see section 2.3). Since the task of providing login credentials to each individual user across a large-scale distributed system is tedious, entities participating in the grid define access limits and sharing policies on a per-Virtual Organization (VO) basis. VOs, in turn, are responsible for managing their own user memberships. If a user is trusted by a VO, and the VO is permitted access to a particular resource on the grid, then the user has access to that resource. Resource sharing policies are not static and can change at any time. [66]

Resources on the grid are cataloged by collective brokering services, allowing users to discover the existence of grid sites and services by querying a resource broker. Once a resource has been located, a request to use the resource can be submitted directly to the site or service desired. Tasks on the grid run as they arrive. Due to the complexity of providing advance reservation capabilities, grid resources historically have been made available on a “best-effort” basis. [66] However, recent advances in grid virtualization technology also permit pre-arranged leases on some systems [78, 94].

As depicted in figure 2.1, the services provided by grid system middleware may be decomposed into three sub-layers: collective services, resource management services, and connectivity protocols. Collective services provide resource accounting and brokering, enabling end users to locate interfaces to available resources. Management services perform arbitration among the individual resources, providing role-based access controls. Standard network protocols provide connectivity both between services and to the underlying physical “fabric.” This low-level fabric includes servers, networking interconnects, storage devices, and the software systems that run directly on the hardware.

Figure 2.1: Conceptual layers in a traditional grid system. Applications do not interact directly with hardware in this model but instead utilize resource abstractions, built on top of commodity communications protocols, to provide and access services.

[66, 65, 63]

Since the grid is distributed and at scale, a wide variety of systems may be made available by resource providers. The total collection of resources available on the grid at any time tends to be heterogeneous, particularly with respect to software stacks. To overcome issues of compatibility between different systems, grid applications ideally should be designed to utilize services instead of direct operating system resources. Grid resources ideally should be provided as services that are accessed through a narrow and well-defined set of protocols, such that the architecture of the grid can be drawn in the shape of an hourglass, with a large number of applications accessing a wide array of resources through a narrow set of services. [66] In practice, however, many grid applications still require direct operating system support, and most grid resources are exposed to users as computational environments at the operating system level.

Figure 2.2: A cluster system represented as a star network with a head node and multiple compute nodes

2.2 Composition of Grids

Although the architecture of grid systems is typically conceptualized at a high level (or top-down view), actual construction of grid systems is accomplished by joining component parts (which may be viewed in a bottom-up model). The lowest level of granularity on a distributed system such as a grid is typically the individual system that serves as a compute node. Compute nodes are frequently Commercial Off The Shelf (COTS) systems that are linked together via a Local Area Network (LAN) to form a *cluster computing system*. A cluster provides its own scheduling services via a dedicated system known as a *head node*. Head nodes also may provide other services, such as name resolution, address leasing, and centralized management [114]. Conceptually, clusters can be represented as a star network, or as a rooted tree of depth one, as depicted in figure 2.2 [17]. Compute nodes within a cluster are typically uniform with respect to software configuration, varying only by minor system configuration settings such as host name or Internet Protocol (IP) address. These configuration details can be set dynamically at boot time, permitting the compute nodes to be stateless systems and improving ease of maintenance. [122]

By themselves, cluster systems typically have high-speed interconnects between nodes, making them suitable for High Performance Computing (HPC) applications [50]. In a grid system, multiple clusters share resources using Internet connections, which typically have high latency and low bandwidth, reducing suitability for HPC tasks [65]. Hierarchies of cluster systems can be developed, resulting in an arbitrary rooted tree topology for a grid system [17]. However, one simplification that is useful for the discussion of Virtual Organization Clusters is that grid systems may be conceptualized as a star network of clusters, where each cluster is a star network of systems (figure 2.3).

Figure 2.3: A grid system represented as a star network of cluster systems. Each cluster system forms a site on the grid.

Figure 2.4: At a high level of abstraction, a grid system may be viewed as a star network of sites.

Each cluster within the grid system is known as a *site*, while the interface between the site and the grid itself is often called a *Computing Element* (CE)¹. Special software installed on the head node of each cluster is used to connect the cluster to the grid.

From the user’s perspective, grid systems are accessed through *middleware*, or software that implements resource discovery, access, and communications protocols. Individual compute nodes are not directly visible to the user, since each grid site will implement its own scheduling and usage policies that abstract away the low-level details of the physical implementation. This abstraction is necessary, since each cluster system in the grid must support both its own local users and permitted grid users, whose access is arbitrated by the grid middleware. [66] In this view, shown in figure 2.4, the grid system itself becomes a star network. Unlike the cluster system, however, scheduling tasks are generally performed at the sites, or at the leaves of the tree.

¹This interface is also called a “Compute Element.”

One property of this cluster-based grid design is that it is difficult, if not impossible, for any one cluster system to contain the necessary software libraries and resources to support the diverse requirements of all VOs on the grid [57]. Enforcing a single software stack on each grid site is impractical in terms of management and tends to result in outdated library and application installations [47]. It is therefore not guaranteed that any particular software application will be able to run on any arbitrary grid site. Users can query metadata catalogs, such as Inca [138], to identify compatible sites, provided that site administrators maintain the accuracy of the catalog. However, even with correct metadata available to users, applications that require specific libraries may be limited to a subset of grid resources. For some applications, the size of this subset may be zero.

2.3 Virtual Organizations

Fundamental to the architecture of grid systems is the concept of a Virtual Organization (VO), or a group of users with a common interest who organize themselves over a Wide Area Network (WAN) such as the Internet. VOs on science-oriented grid systems like OSG typically have a science-oriented mission, with areas as diverse as genome mapping, environmental modeling, and nanoelectronics simulation. However, there is no fundamental requirement that VOs consist only of scientists; industry groups working on a new product, artists requiring distributed rendering capabilities, and engineers simulating behaviors of new structures are all examples of groups that could form VOs and utilize grid services. [66] In addition, organizations that provide resources to the grid are also organized into VOs, which try to deliver specific Quality of Service (QoS) targets to consumer VOs. [65] Figure 2.5 depicts several VOs on a single grid.

A distinction between “consumer” and “provider” Virtual Organizations is not always clear. For the purposes of discussing Virtual Organization Clusters, the provider and consumer designations may be viewed as roles within the grid. It is possible for a group of scientists to form a VO for the purpose of accessing grid resources to conduct scientific simulations, in which case the VO has the role of a consumer. However, the same group of scientists may provide their own resources to the grid for use by other VOs, thereby assuming a provider role. In the case of Virtual Organization Clusters, the VO assumes a consumer role for the purpose of obtaining grid resources, while it functions as a provider of administration services to its own users.

Figure 2.5: Virtual Organizations on the Grid: VOs may represent groups of consumers, groups of resource providers, or groups that both provide and consume resources.

Figure 2.6: Regular jobs consist of a batch-style task submitted by the user to the grid Computing Element (CE) at a grid site. These jobs may request data from a grid Storage Element (SE), but the executable application is packaged with the job itself.

2.4 Computational Jobs

Computational jobs executed on grid systems may be classified either as *regular* jobs or *pilot* jobs. Regular jobs, illustrated in figure 2.6, consist of a job description and application executable transmitted together to a grid site. Once the job arrives at the Computing Element interface on the site, it is placed into a scheduler pool for execution when resources become available. At execution time, the job may require data from external sources, which can be transferred directly or via a grid Storage Element. However, the actual application payload is contained within the job itself. Jobs are submitted directly to sites in the general case, although *metascheduling* of jobs is also possible, in which case jobs are sent to a grid metascheduler, which selects a site to which the job should be submitted. If a metascheduled job fails to execute properly on a site, the metascheduler may re-submit the job to a different site. [10]

Pilot jobs differ from regular jobs in that the actual application executable is not “pushed” to the grid site with the job description. Instead, a simple executable task is sent to a grid site, and this task borrows the computational resource from the site at execution time. The simple task then contacts a remote server and retrieves (“pulls”) the actual workload to be executed, which may require another network transaction to obtain data (figure 2.7). Most simple tasks submitted as pilot jobs are machine-generated grid jobs produced by Workload Management Systems (WMS) that

Figure 2.7: Pilot jobs are submitted to grid Computing Elements in the same manner as regular jobs. Once scheduled and executed, pilot jobs request one or more executable applications from an external server. These applications are then run on the worker node leased from the grid site by the pilot job, potentially transferring data via a Storage Element.

receive tasks directly from users, “-subverting the established accounting and auditing procedures” [143]. Virtual Organizations frequently utilize these workload managers to share resources among individual users, as regular grid middleware only arbitrates resource access among the VOs. WMS mechanisms have become so popular on the Open Science Grid that pilot jobs now account for the majority of grid jobs on OSG. As a result, sites are deploying additional accounting and security mechanisms that are WMS-aware. [143]

Although grid jobs resemble the types of task that can be executed on a single cluster by a regular scheduler, such as Condor [155], a fundamental property of grid systems is that the links between grid sites have high latency and low bandwidth, since the commodity Internet is often used for this purpose [65]. Thus, regardless of whether a task starts via a regular job or through a pilot system, significant use of network resources will result in reduced task performance when compared to execution on a cluster system. Bag-of-tasks applications, which consist of multiple tasks that execute independently without communication among themselves, are considered ideal applications for grid environments [17].

Figure 2.8: A grid system is an example of a Service Oriented Architecture, in which the low-level details of the physical fabric are abstracted by common middleware that presents a uniform interface to the applications.

2.5 Service-Oriented Architecture

“A service ... is defined solely by the protocol that it speaks and the behaviors that it implements” [66]. A *Service Oriented Architecture* (SOA) is a mechanism for abstracting the details of a collection of heterogeneous systems, so that the same interface may be used to access and utilize resources across different domains. SOAs thus function as a type of high-level virtualization by enabling applications to run on disparate platforms without modification. However, instead of using individual operating-system level processes as application containers, SOAs utilize web services as the basic components of the system. Complex systems are constructed by “orchestrating” simpler web services, integrating them into a larger solution. When this model of computation is applied to a grid system (figure 2.8), applications are constructed solely from abstract services, without low-level operating system dependencies. [65]

As originally conceptualized in early models of the grid, the Open Grid Services Architecture (OGSA) provides a framework for service-oriented application development based on the Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP). The Globus grid middleware is based on these technologies and includes the Grid Resource Allocation and Management (GRAM) and the Grid Security Infrastructure (GSI). GRAM provides the “gatekeeper” mechanism required to implement a grid Computing Element, enabling resources to be connected

to the grid system, while GSI supplies authentication services for users. Additional web services have been developed to move the grid into a large SOA-based resource. By implementing scientific applications as sets of Globus-aware services using WSDL, applications theoretically would be able to utilize a wide range of resources available on the grid. Since the services provided on a dynamic grid system may change frequently, such collections of services must be adaptable to the changing environment. Providing Quality of Service guarantees in applications that aggregate services is difficult. [65]

While grid systems constructed from collections of services are achievable, as evidenced by In-VIGO [7], in practice most grid systems utilize service-oriented middleware to provide application routing services to grid users [17]. These services include mechanisms for monitoring sites, submitting computational jobs, checking job status, and transferring data across the grid. A widely used service for job submission is Globus, which enables user jobs to be transferred to grid sites by means of an XML-based web service. Prior to job submission, user credentials are authenticated by the GSI middleware component of Globus, ensuring that users can send jobs only to authorized sites. Once a user job reaches a grid site through Globus, it is received by the local scheduler and queued for execution according to local scheduling policy. [64] Upon execution, the job runs as a regular process at the operating system level. If the job is compatible with the system on which it executes, then the execution should be successful, provided the job is not preempted by the local scheduler. Should the job require a library or other software resource not present on the compute node, then the job will fail. Thus, while the high-level grid may be considered to be an SOA, user applications are still typically process-oriented instead of service-oriented.

2.6 Cloud Computing Systems

When computational systems are designed to be accessed entirely through a Service Oriented Architecture, the resulting implementations are often marketed as forms of “cloud” computing. Cloud computing is a somewhat nebulously defined concept, the meaning of which may vary in different contexts. One definition of cloud systems follows directly from the decomposition of mainframe systems into data centers constructed of Commercial Off The Shelf (COTS) components, where the computational power of the data center is derived from the “cloud” of disparate systems used to replace the monolithic mainframe. Another variation of the definition follows from the availability

of fast Internet connectivity, which permits the data centers to be located in areas of relatively low energy cost, thereby maximizing the amount of computation that can be effected for the same amount of money. Consolidation of the infrastructure at a potentially remote site results in a “cloud” in the sense of an abstraction: computational tasks are sent away to the data center for remote execution. [163]

Another iteration of the cloud computing definition results from the deployment of operating system virtualization technologies (chapter 3). Large data centers are designed with excess capacity for burst availability and redundancy, which results in a large number of spare computing cycles. Virtualization permits these unused cycles to be harvested to perform useful work, provided that tasks can be matched to free resources by means of a dynamic scheduling system that is adaptable to changing workloads. Thus, the “cloud” becomes the collection of virtual machines, which provides an abstraction for the lowest level of resources available in the system. The autonomic systems that partition the workloads between virtual machines comprise a distributed system that performs the functions of the middleware in a grid architecture. [163]

As illustrated in figure 2.9, excess computational cycles in the data center can be sold to third parties to host virtual machines and associated applications. Large providers, including Google and Amazon.com, offer dedicated application and virtual machine hosting services that reduce wasted energy, since the exact resources required by an application can be provisioned in a virtual environment (see section 3.6.2). In this sense, the “cloud” is a computing model in which all the computational hardware, power distribution, cooling equipment, and maintenance are offered as a demand-based service (Infrastructure as a Service, or IaaS). [163]

Another model of cloud computing, also illustrated in figure 2.9, extends the IaaS concept to provide Software as a Service (SaaS). Since all processing, data storage, and complex software systems are moved to the data center, users may interact with this type of cloud system using a low-power device such as a thin client, netbook, or mobile phone. With resources and data both stored in the cloud, users can access their applications, documents, and other data from any location with Internet connectivity. One popular application of this type of cloud technology is web-based e-mail, which frees the user from maintaining copies of the same e-mail messages on different computer systems. Drawbacks to this model include availability resulting from limited network capacity, along with privacy and security concerns that result from the remote storage of personal information. Without enforced open standards for cloud systems, there is also the threat of vendor

Figure 2.9: Cloud computing systems provide computational resources, including processing capabilities, data storage, and system support to users. Users may access these resources using relatively low-power devices, allowing the bulk of energy and infrastructure costs to be centralized in less expensive locations.

lock-in, since the remote provider supplies both the computation and data storage services. [163]

For the purpose of this dissertation, cloud computing is defined as computation that occurs within an autonomic private software environment. Autonomic systems manage themselves via software agents to create computational infrastructure that is self-configuring, self-optimizing, self-healing, and self-protecting [96]. Thus, for the purposes of this discussion, cloud systems must adapt to their execution environments automatically, allocating and returning computational resources in response to changing application demands. Private software environments are realized through the use of virtualization systems, which are described in the next chapter.

Chapter 3

Operating System Virtualization

Operating system virtualization enables one operating system to “host” a second “guest” operating system. The guest system is provided virtualized hardware resources, as shown in figure 3.1, so that the guest appears to be indistinguishable from a regular physical system. Special software known as a Virtual Machine Monitor (VMM) or hypervisor¹ is installed on the host computer to provide this illusion. Virtualization systems allow multiple guest operating systems to be run in parallel on the same hardware, permitting legacy applications to be hosted on newer infrastructure. In addition, the ability to control the resources of isolated guests permits untrusted operating systems to be hosted inside a safe environment. Since these virtual environments become independent of the host hardware, virtualization enables scalable application deployment in response to changing resource requirements. [130] These same benefits can be realized on grid systems if virtualization technologies are widely deployed [57].

Virtualization technologies originally were developed by IBM Corporation to multiplex expensive mainframe systems [132]. In 1964, a group of IBM engineers started the design of a research system known as the CP-40, which featured the first implementation of virtualization software. Despite the efforts of some IBM managers to kill the project, the implementation was released to a small group of test customers for use on the IBM System/360 Model 67. Official support by IBM, and widespread availability of the virtualization software, arrived in November 1972 after the System/370 mainframe was released. [160] Although the System/370 was commercially successful, virtualization technology remained limited to mainframe systems until the 1990’s, when VMWare

¹In this document, the terms VMM and hypervisor are taken to be synonymous and are used interchangeably.

Figure 3.1: Virtual Machine (guest) hosted by a physical system (host). The guest system is presented with the same, or similar, hardware as the host by means of software emulation or multiplexing. Aside from timing differences, application execution within the guest system is indistinguishable from execution of the same application on the host.

Workstation and Microsoft Virtual PC became available for Intel desktop computers [56].

Since the original Intel x86 architecture was not virtualizable [130], dynamic recompilation techniques were required to implement virtualization on the desktop. These dynamic recompilers were not as efficient as their true virtualization counterparts on the mainframe, necessitating fast processors to provide acceptable desktop performance [150]. During the final years of the 20th century, significant clock speed increases occurred in commodity systems. Due to reliability concerns, these systems often each hosted a single application, leading to a large number of wasted clock cycles. Virtualization provided a mechanism for harvesting these cycles to run multiple applications in isolated containers on the same hardware. [132]

At present, desktop virtualization systems can be loosely divided into four classes: operating system containers, dynamic recompilers, paravirtualization systems, and hardware-assisted virtualization extensions. Both dynamic recompilers and hardware-assisted virtualization systems are able to execute unmodified guests that have an Instruction Set Architecture (ISA) supported by the host. These two approaches differ in that certain instructions (see section 3.2) must be trapped and substituted by a dynamic recompiler, while hardware extensions permit these instructions to run directly on the processor, with possible traps to a hypervisor program. Paravirtualization systems remove the need to trap these instructions by requiring that the guest operating system be modified to use the paravirtualization interface instead of the regular ISA. For practical purposes, this

modification requirement restricts the set of available guest systems to certain Linux distributions. [14, 56, 150, 159]

Performance of applications executed within virtualized guest systems varies by hypervisor and workload. To measure the overheads introduced by a particular hypervisor for a particular type of workload, benchmarking tests are required. These tests use an application, such as High-Performance Linpack (HPL), to measure the native execution speed of an application running directly on the host system (without virtualization). The same application is then executed, using the same parameters, in the guest system. By dividing the relative difference in benchmark performance by the host metric, an overhead of virtualization may be obtained for the hypervisor and workload combination. [55, 54] These metrics are required to evaluate specific implementations of Virtual Organization Clusters.

The remainder of this chapter is organized as follows. Section 3.1 provides an overview of the relationships between the different levels of virtualization discussed in this document. Following this discussion, section 3.2 describes the Popek and Goldberg requirements for efficiently virtualizable Instruction Set Architectures. Three techniques of providing virtualization support on the x86 and x86_64 architectures are then discussed: dynamic recompilation (section 3.3), paravirtualization (section 3.4), and hardware virtual machine extensions (section 3.5). Finally, section 3.6 presents several enterprise and cloud computing applications of virtualization technology.

3.1 Levels of Virtualization

Virtualization systems allow architecture-compatible software systems to be decoupled from the underlying physical hardware implementation, thereby allowing computation to be location independent. [56] These systems can be loosely divided into four levels: operating system containers, dynamic recompilers, paravirtualization, and hardware-assisted virtualization. The latter three systems can be termed “full virtualization systems,” since they permit entire operating systems, including the kernel, to be hosted as guests. Since Virtual Organization Clusters are designed to contain entire environments, including private kernel and networking services, full virtualization systems are preferred. However, the different types of systems have different general properties, as illustrated in table 3.1.

Containers isolate collections of guest processes that execute directly under the host system

Table 3.1: Levels of Virtualization. Different virtualization systems have varying support for separate guest kernels, require (or do not require) modifications to the guest system, isolate the guest system in different ways, and satisfy or do not satisfy the Popek and Goldberg efficiency requirements.

Level	Guest Kernel	Guest Modifications	Isolation	Efficient
Operating System Container	Shared	Not Applicable	Userspace	Yes
Dynamic Recompilation	Separate	Unnecessary	Kernel	No
Paravirtualization	Separate	Required	Kernel	Yes
Hardware-Assisted Virtualization	Separate	Unnecessary	Kernel	Yes

kernel with the same restrictions as any other process on the host. Example container systems include OpenVZ, VServer, Virtuozzo, XR Enterprise, chroot jails, and Solaris Zones. [104, 161] Container-based virtualization is subject to a phenomenon known as “QoS crosstalk,” which results from imprecise kernel resource accounting when several containers are running on one host. As a result, it is difficult to control guest resource allocations properly with containers. [14] Moreover, the requirement that all guest processes run under a specific kernel limits their utility as a useful virtualization abstraction on grid systems.

Of the full virtualization systems listed in table 3.1, hardware-assisted virtualization provides the greatest potential for use in grid environments, since entire guest systems can be run efficiently and without modification. Dynamic recompilation systems enable unmodified guests to execute, but efficiency is compromised by the need to substitute instructions at execution time. Paravirtualization systems are more efficient than dynamic recompilers, but modifications to the guest operating system are required. Since Virtual Organization Clusters are to remain technologically agnostic, support for guests that cannot readily be modified (such as proprietary operating systems) must be provided.

3.2 Instruction Set Architecture Requirements

A Virtual Machine, as defined in Popek and Goldberg, is “an efficient, isolated duplicate of the real machine” [124]. VMs are efficient provided that a “statistically dominant subset of the virtual processor’s instructions ... [are] executed directly by the real processor, with no software intervention by the VMM” [124]. Execution of an application within a VM must be indistinguishable from execution of the same application on the same host, except for differences in resource availability and timing. The VMM maintains control of the resources allocated to the VM, but it does not intercept or interpret every instruction executed by the guest: most guest instructions are passed

Figure 3.2: According to the Popek and Goldberg virtualization requirements, a machine is virtualizable if its sensitive instructions are a subset of its privileged instructions, as shown in the Venn diagram.

directly to the host CPU. In addition, the VMM isolates guests from the host, preventing guests from changing host system state. A computer (or Instruction Set Architecture) for which an efficient VMM can be constructed is called “virtualizable.” [124]

Instructions provided by an ISA may be classified as *privileged* or *unprivileged*, where privileged instructions require the CPU to be in a supervisory or elevated privilege mode. If an attempt is made to execute a privileged instruction in an unprivileged mode, a *trap* will occur, permitting the operating system (or VMM) to intercept and handle the instruction. Instructions may also be classified as *sensitive* or non-sensitive. Sensitive instructions are further subdivided into behavior-sensitive instructions, which depend upon a physical resource setting, and control-sensitive instructions, which change resource allocations. Since a VMM is designed to run in unprivileged mode, and VMs should not change the state of the host system, the VMM process must be invoked by the privilege trap to interpret privileged guest instructions. An ISA is virtualizable if the set of sensitive instructions in the ISA is a subset of the privileged instructions in the same ISA, as illustrated in the Venn diagram in figure 3.2. Furthermore, an ISA is recursively virtualizable if it is virtualizable and a VMM without timing dependencies can be constructed for it. [124]

The Intel x86 ISA, as implemented on the original Pentium processor family, is not virtualizable, due to the presence of 17 sensitive instructions that are not also privileged instructions [130]. If a VMM as described in Popek and Goldberg is constructed for this family of processors, the CPU will not trap when unprivileged sensitive instructions are executed, preventing the VMM from delivering correct state information to guests. [132] Thus, implementation of virtualization on

Figure 3.3: Virtual machine architectures based on dynamic recompilation employ a hypervisor application that runs as a process in the host operating system. Guest virtual machines are executed within this process, which emulates the computer hardware seen by the guest. Since a hypervisor process runs for each hosted virtual machine, this virtualization architecture is sometimes called a “Type II” system.

the original x86 ISA requires either the construction of a less-than-efficient dynamic recompilation system (next section) or the modification of guests to run in a restricted subset of the ISA (paravirtualization, as described in section 3.4). Subsequent extensions to the original x86 architecture permit some newer x86 and x86_64 systems to provide a virtualizable ISA; these extensions are discussed in section 3.5.

3.3 Dynamic Recompilation

In dynamic recompilation systems, illustrated in figure 3.3, the VMM is installed as a service inside the host operating system, and the guests run as regular user-space processes on the host, using special kernel-level drivers to intercept and dynamically recompile privileged instructions. The VMM emulates hardware including video cards, disk devices, and networking interfaces. By emulating common and well-supported hardware, such VMMs can support a wide range of unmodified guest operating systems, at a cost of reduced performance. [56, 150] A VMM (also known as a hypervisor) process runs for each hosted virtual machine, creating a setup known (controversially) as a *Type II hypervisor*. [130]

Dynamic recompilation works by replacing both sensitive and privileged guest code with equivalent, unprivileged host code that calls an interface provided by the VMM. This code replacement occurs the first time a block of guest code is encountered, at which time the translated block

of equivalent code is stored in a data structure known as a trace cache. Future execution of the same block of code results in a trace cache hit, substantially improving performance by amortizing the cost of re-compilation over all the calls to the same code block. Translated code is designed to be as efficient as possible: calls into the VMM are executed directly, without the overhead of traps and associated context switches. [132] All guest state is maintained in the unprivileged code of the VMM, bypassing the processor's native state and privilege enforcement. [159]

Virtualized devices are provided to guest systems via a hosted architecture, in which emulated implementations of common Input/Output (I/O) devices are provided. The VMM must intercept and handle all guest I/O calls in the host operating system, resulting in substantial overheads. As an example, the VMWare Workstation virtualization system requires two context switches (one in the host and one in the guest) for each network packet sent or received. Moreover, receiving a packet into the VMM requires polling the network interface with a select statement, since packets destined for each guest must be dispatched separately from those destined for other guests or for the host. VMWare Workstation thus incurs substantially greater overhead when receiving packets than when transmitting packets, which can result in an 80% performance degradation for network-bound tasks when a large number of VMs are running. In contrast, when specific hardware is installed on the host system, VMWare ESX server is able to bypass host I/O emulation for the network device, greatly improving performance. [150]

3.4 Paravirtualization

Paravirtualization enables virtualization systems to be supported on non-virtualizable platforms, such as the Intel x86 architecture, by exposing an alternate virtualizable interface to guests. Unprivileged sensitive instructions are removed from the guest and replaced with instructions provided by the hypervisor, creating an overlay architecture that satisfies the Popek and Goldberg virtualization requirements. These systems are typically (but not always) implemented as *Type I hypervisors*, or virtualization layers installed directly on the computational hardware (figure 3.4). All user-facing systems are then implemented as guests, where one or more guests have the ability to manage other guests. [130, 132, 161]

One popular paravirtualization platform is Xen, which is engineered to support 100 guests executing on a single host. Xen has significant performance advantages when compared to dynamic

Figure 3.4: Type I paravirtualization systems use a single hypervisor layer installed directly on the hardware. Guest systems are modified to use the hypervisor interface.

recompilers and fault-dependent VMMs, due primarily to the direct sharing of hardware I/O devices. For efficient use of the hardware Translation Look-aside Buffer (TLB), the Xen hypervisor directly manages hardware page tables and associated page faults, requiring its implementation as a Type I hypervisor system. As such, Xen replaces the host operating system with a relatively lightweight hypervisor layer, which is managed by a special privileged guest called Domain0 (dom0) that persists as long as the host is running. [14]

Xen implements a virtualizable system with user and supervisor privileges by porting privileged guest code to run in ring 1 on the x86 CPU. [14] Intel x86 and compatible platforms utilize four “rings,” or CPU privilege levels, numbered 0 to 3. With the exception of IBM OS/2, recent desktop operating systems utilize only rings 0 and 3 for privileged and unprivileged instructions, respectively. By utilizing one of the unused privilege levels, Xen partly relies upon the built-in CPU privilege checking to enforce guest security, avoiding the need to monitor guest instructions in user-level code (ring 3) – a situation known to Intel engineers as *ring compression*. This design improves performance when compared to dynamic recompilation systems. [159]

Host hardware is typically shared directly in Xen, although emulated devices are available via code borrowed from the Qemu project. Only the hardware actually required by the host OS is virtualized in Xen. [161] Shared hardware is made available to Xen guests via an I/O interface that uses Direct Memory Access (DMA) for performance, with request security checking logic to enforce

isolation. I/O requests use descriptor rings (circular buffers) that permit out-of-order execution and host-guest communications without copying. Network packets are delivered to guests via these same DMA buffers, which are allocated by the guest and provided to the host whenever a receive operation is requested. Memory for guest systems is statically provisioned at guest initiation but can be increased to a specified limit during execution. [14, 67]

Aside from Xen, several other paravirtualization systems exist, including Disco for the MIPS platform [132] and User Mode Linux (UML) for the x86 platform. User-Mode Linux is a paravirtualization system that utilizes a host Linux kernel to run a guest Linux kernel directly [14]. Since the host installation includes a full operating system, the architecture of UML is closer to that of a Type II hypervisor than a Type I hypervisor.

Measured performance of paravirtualization systems typically exceeds that of dynamic recompilers. While CPU and memory management tend to scale well with increasing loads on most VMMs, paravirtualization is typically advantageous when workloads involve a significant amount of I/O [126]. The Xen hypervisor has execution overheads as low as 3% for some workloads and network throughput overheads as low as 14% for some packet streams [14]. Tests of virtualization systems in grid contexts yield execution overheads as low as 5% [61]. HPL tests conducted with scaling show application throughput overheads of up to 30% with Xen for a 7x4 matrix. The same overheads for a CPU-bound 1x1 matrix on a single VM are measured at 6.6% with Xen. [55][54] Un-scaled HPL tests with User Mode Linux (UML) show a virtualization overhead of 20% in terms of application throughput [133].

The most significant disadvantage to paravirtualization systems is the requirement that guest operating systems be ported to run on the hypervisor. Backward-compatibility of system-level code is a concern during code modification, and vendor cooperation is required for proprietary operating systems. [132] While paravirtualization systems such as Xen do not require modification to applications running within the guest [14, 132], required modifications to the guest can be significant. Linux guests require approximately 3,000 lines of code changes, while Windows XP guests require a substantially larger number of adjustments [14]. For these reasons, the long-term suitability of paravirtualization systems for production applications is questionable, since “adequate hardware support can decrease overhead ... to the point that the value of having a fully compatible virtual machine abstraction overrides any performance benefits from breaking compatibility” [132].

3.5 Hardware Virtual Machine Extensions

Hardware Virtual Machine (HVM) extensions, available on many newer Intel and AMD processors, implement an extended x86/x86_64 instruction set that is virtualizable according to the Popek and Goldberg requirements. The Intel Virtual Machine Extensions (VMX), first released in November 2005 on the Intel Pentium 4 models 662 and 672 [81], add a top-level set of execution modes to the CPU: host and guest. Within each mode, all privileged instructions run in privileged rings (usually ring 0), while unprivileged instructions run in ring 3, eliminating ring compression. Whenever the CPU needs to be switched between guests, or back to the host, a mode switch operation is invoked by VM entry and exit instructions. These instructions save and restore state information in the VM Control Structure (VMCS), allowing state information for the host and all guests to be maintained by the hardware. Interrupt masking and other common operating system support routines are implemented directly in the CPU, thus requiring mode switches back to the hypervisor only for operations that cannot be handled directly by the CPU. [159] On CPUs with the first generation of virtualization extensions, mode switching can be more expensive than dynamic recompilation for some workloads [9].

In this hardware-assisted model, the user-space component resembles a Type II VMM and emulates certain hardware devices available to the guests by means of a VMM process (figure 3.5). [159] This device emulation has the same benefits of device uniformity available with dynamic recompilation, with the same drawbacks in I/O performance. Some direct device sharing is possible with code borrowed from paravirtualization systems, although this sharing is easier to accomplish on channel-oriented devices such as USB and SCSI hardware. As originally implemented in the first generation of virtualization extensions, shadow page tables are still implemented in software, which reduces total memory performance. [132] Newer AMD and Intel CPUs implement hardware page table virtualization to mitigate this problem [76]. Additional I/O performance improvements could be obtained if hardware devices were designed to support direct access from guest systems [132], a feature provided by some high-end network cards [106].

HVM capabilities have been added as an optional virtualization mode to Xen, allowing it to run unmodified guests. However, a newer application called the Kernel-based Virtual Machine (KVM) system relies on HVM extensions exclusively, eliminating dynamic recompilation and paravirtualization for features other than I/O. When requested, emulated hardware devices are provided

Figure 3.5: Full virtualization with hardware support. A hypervisor (VMM) process runs to manage each guest and provide emulated hardware, but CPU virtualization extensions manage the majority of guest instruction handling and sensitive instruction management.

by Qemu, one instance of which runs for each virtual machine hosted. Paravirtualized I/O devices, including a network interface, are supported for Linux guests. [129]

A major benefit of KVM for use in grid computing is its “snapshot” mode, inherited from Qemu. In this mode, KVM can spawn multiple instances of a virtual machine using a single image file, eliminating the need to have one image file per instance. [129] For cluster computing systems, this operating mode eliminates image staging to compute nodes (figure 3.6), a time-consuming operation [51]. With snapshot mode enabled, multiple VM instances can be booted from a single shared filesystem (such as PVFS [28]), as illustrated in figure 3.7. The shared image is maintained in a read-only state, with write operations redirected to temporary files on each compute node.

The virtualization performance of KVM on hardware with first-generation virtualization extensions is acceptable, although it is below that of Xen with paravirtualization. Scaled HPL benchmark tests indicate network-bound overheads of up to 85% using a 7x4 matrix, compared to 30% with Xen. Virtualization overheads for compute-bound processes, using a 1x1 matrix on a single VM, are measured at 8.8% for KVM – slightly higher than the corresponding 6.6% Xen measurement. It should be noted that a stable version of Xen is compared to an alpha version of KVM in these tests. [55][54]

Figure 3.6: Virtual machine images used in read-write mode generally must be copied (staged) to each host prior to VM instance invocation.

Figure 3.7: In KVM snapshot mode, a single VM image can be stored on a distributed filesystem and booted directly by each VMM, without staging. The shared image is maintained in a read-only state, with writes redirected to a temporary file on each local compute node.

Table 3.2: Some common virtualization systems supported by Linux hosts. Table adapted from [14, 105, 129].

Virtualization System	Type
VMWare	Dynamic Recompilation / HVM
Xen	Paravirtualization / HVM
KVM	HVM
Qemu	Dynamic Recompilation
VirtualBox	Dynamic Recompilation / HVM
UML	Paravirtualization
OpenVZ	Container
VServer	Container

3.6 Applications of Virtualization Technology

Virtualization technology enables the creation of customized environments for a wide range of applications, including server consolidation [56], migratable containers [132], local containers for experimental test procedures [141], grid-connected parallel computing systems [57], and cloud computing implementations [132, 163]. A number of virtualization systems are available to support these applications on x86 and x86_64 Linux hosts, as listed in table 3.2. In the remainder of this chapter, two major classes of virtualization applications will be discussed briefly as examples: server consolidation (section 3.6.1) and cloud computing (section 3.6.2).

3.6.1 Server Consolidation

One widely used application of virtualization is server consolidation, which involves the migration of physical servers into virtual machines. Security and reliability are increased when virtualization technology is used, since individual server applications are contained within their own private environments. [56] Since physical machines frequently host single services to mitigate availability problems resulting from an unstable application, virtualization can improve system utilization by allowing a single physical machine to host multiple highly available services. If one service becomes unstable and results in a system-wide error condition, that error will remain contained within the virtual machine hosting the service. Other services will be unaffected by such an issue. Furthermore, the hardware independence of virtual machines permits containers to be migrated from system to system within the data center, thus improving system load balancing while decreasing the need for redundant infrastructure. [132]

Server consolidation into single data centers also improves hardware utilization and energy

efficiency by reducing the number of wasted cycles. Since physical systems would otherwise host single applications without virtualization, extra cycles on individual systems would be wasted. In addition to the lost amortization of the hardware equipment cost, these wasted cycles also contribute to energy demand both directly and through increased load on cooling infrastructure. Placing virtual servers in such a way as to maximize utilization and minimize waste is thus a means to cost savings. Moreover, when entire server farms can be centralized into data centers and made geographically independent of corporate operations, the data centers can be located in areas with relatively inexpensive electricity. Significant cost savings can be realized in such situations. [163]

3.6.2 Cloud Computing

Server consolidation into data centers reduces, but does not completely eliminate, wasted computational cycles. Enterprise applications require a certain amount of redundancy and burst capacity to maintain high availability, and this capacity is unused most of the time. By utilizing virtualization to provision isolated environments, corporations can sell the excess capacity to third parties. Several large corporations, notably Amazon, take this model further and provide entire virtual machine hosting services for a fee. Companies that do not desire to maintain their own computational resources and infrastructure may simply purchase these services on a fee-for-use model, creating a cloud system (section 2.6). Energy and resource waste is minimized through clouds, since environments can be provisioned to match the requirements of the workload. These virtual environments provide burst capacity through on-demand resource reallocation. [163]

When virtualization systems are used to implement cloud computing services, the services begin to resemble computational grids. Cloud systems enable applications to obtain resources beyond those immediately available, just as grid computing promotes resource sharing for the same purpose. The distinctions between individual systems in the cloud begin to diminish, as effective use of cloud architectures requires middleware abstractions to dispatch workloads to different servers. By definition, the process of dividing these workloads intelligently – and, preferably, autonomically – is a distributed computing task. [163]

Cloud computing with virtual machines also provides a solution to the multi-core problem. Since a large number of existing production applications are written for a single CPU core, several single-core virtual machines can be hosted by one multi-core physical system, improving utilization while reducing hardware costs. In cases where high speed Internet access is ubiquitous, entire user-

facing systems can be placed in the cloud and accessed from any location by means of a low-power device (see figure 2.9). [132] With per-hour execution costs as low as ten cents on the Amazon Elastic Compute Cloud [11], cloud-based Software as a Service (SaaS) may be practical for some applications.

Chapter 4

Related Work

Since Virtual Organization Clusters comprise a new grid architecture, a fairly broad set of research projects is related to VOC design and evaluation. This set can be loosely divided into grid system virtualization (section 4.1), grid simulation (section 4.2), system resource analysis (section 4.3), and system management (section 4.4). Several of these broad areas may be further subdivided into related projects. However, all areas are important to consider when evaluating VOCs in the context of production grid systems.

4.1 Virtualization of Grid Systems

Virtualization of grid systems, first proposed in [57], offers substantial benefits to grid users and system administrators. Users of virtualized systems can be granted administrative access rights to their virtual machines, thereby allowing end-user customization of the software environment to support current and legacy applications. These customization features can be granted without imposing additional workloads on the hardware administrators, making customization on a per-user basis practical. Moreover, isolation of VMs from the underlying hardware can provide additional security when compared to traditional shared multiprogramming systems. Since the hardware administrators retain control of the Virtual Machine Monitors (VMMs) or hypervisors, coarse-grained resource controls can be implemented on a per-VM basis, allowing hardware resources to be shared among different VMs. [57]

A primary benefit of offering administrative access to VM users is that custom overlay sys-

tems can be developed, where the environment of each virtual system can be determined by its user base. Such capabilities have existed since the early days of computing, when the Regnecentralen RC4000 Monitor system enabled different users to run their own kernels with whichever process scheduling algorithm each user chose [19]. Virtual Organizations on modern grid systems are increasingly interested in performing their own resource management and scheduling functions, so that individual communities have finer-grained control over communal resources [62]. Virtualization of grid systems provides the needed mechanisms for community-specific customization [127].

Grid services, by definition, communicate with each other and with users via middleware, or software layers interposed between physical fabric and applications [65, 66]. Such middleware may include Globus [60], which is part of the standard middleware stack on the Open Science Grid [125], as well as custom middleware used for specific applications like grid filesystem access [173]. Most of the existing grid virtualization systems, described below, require all participating sites and users to install and enable system-specific middleware.

Many existing grid virtualization systems can be classified into lease-oriented systems (section 4.1.1) or autonomic systems (section 4.1.2). Related systems may then be grouped by project. One unique project that relies more heavily on services and does not fit well into a leasing or autonomic classification is In-VIGO, discussed in section 4.1.3.

4.1.1 Leasing Models

One approach to grid system virtualization is to treat grids as commodity sources of computational resources, which can be obtained to run virtual machines directly. This model of leasing systems, illustrated in figure 4.1, requires the user to make an explicit resource reservation to instantiate a virtual grid node. Since each user manages his or her virtual environment directly, this approach violates the transparency property [53] required of distributed systems. Several leasing models have been proposed for grid computing, including Globus Virtual Workspaces (also known as Globus Nimbus, section 4.1.1.1) and Shirako and its Cluster On Demand back-end (section 4.1.1.2).

4.1.1.1 Virtual Workspaces

Virtual Workspaces, implemented as part of the Globus Nimbus toolkit [77], are an extension of Dynamic Virtual Environments (DVE) – dynamically instantiated execution environments designed to isolate individual grid users. Allocation of a DVE is accomplished in its simplest form

Figure 4.1: In a leasing architecture, the user queries the collective brokering services (steps 1 and 2) to locate a resource that supports virtualization. The user initiates a lease through middleware, which starts a virtual machine (step 3). Once the virtual machine has booted, the user interacts with it directly, running the application in a manner similar to execution directly on a local system (step 4). Note that the grid middleware is bypassed in the last step, violating the transparency principle from [53].

by dynamically adding a user account to the local system [93]. Extended DVEs can utilize VMWare virtual machines as locally instantiated trusted containers, which must be trusted to return their allocated resources once limits have been reached. Untrusted users therefore cannot be given administrative access to these VMs. [89] Virtual Workspaces further extend DVEs by using Xen virtual machines to offer customized execution environments to the untrusted users, as Xen permits forcible recovery of resources allocated to the guests [89, 172].

Virtual Workspaces provide a lease-oriented mechanism for sharing resources on grid systems with fine-grained control over resource allocation. A Virtual Workspace (abbreviated VW) is allocated from a description of the hardware and software requirements of an application, allowing the workspace to be instantiated and deployed on a per-application basis. From the perspective of the end user, deployment of a VW first requires the explicit construction of the VW environment as a Xen VM, which is automated through user-provided specifications to a “VW Factory.” Following the construction of the environment, the VW must be explicitly deployed on a host site, after which it can be directly accessed to run computational jobs. [94]

The primary motivation for Virtual Workspaces is to provide high-level, fine-grained resource management to deliver specific Quality of Service (QoS) guarantees to different applications [90]. Resources managed by VWs are provided directly to the end user, who is responsible for initiating his or her own workspace VM for the job or set of jobs he or she desires to run on the grid. These leases allow for individual jobs or applications to be given different QoS priorities, such as best-effort or pre-reserved allocations [148]. Provisioning of workspace services directly to Virtual Organizations, instead of directly to end users, has been discussed only at the level of sharing workspace-hosted edge services [128] between multiple VOs [69].

Aggregation of Virtual Workspaces into Virtual Workspace Clusters has been proposed in [172], [61], and [95]. These clusters are constructed by aggregating sets of homogeneous VW descriptions into heterogeneous outer sets. The outer sets are then matched to physical resources in an aggregated operation, permitting the cluster to be scheduled. [95] In practice, creation of a VW Cluster requires a single VM image per cluster node. While VM images can be created using the standard VW tools, inefficiencies would result if multiple copies of the same image were transferred across the grid to a workspace hosting site. To reduce this wide-area copy overhead, an image staging system at each cluster site to make copies of the image for each hypervisor instance has been proposed [172]. Staging time is related to the size of the virtual cluster and reaches at

least 400 seconds for a VW Cluster with 8 virtual nodes, leading to the conclusion that staging is an operation to be measured in “minutes ... rather than seconds” [172]. Using shared nodes on a standard Network File System (NFS) share (without staging) was found to be impractical, owing to the large amount of bidirectional traffic involved with reading from and writing to the Xen images [61]. While the importance of the staging system is expressed in [172], [61], and [95], in-situ updates to the core workspace images are not discussed except to state their necessity [18]. Presumably, any changes to the Virtual Workspace Cluster environment would be effected by transmitting and staging a new image. Also, modifications to the cluster scheduling algorithm have been proposed as a mechanism for scheduling staging as a part of preparing the workspace for execution [148].

Selection and final customization of VM appliances to match the requirements of individual jobs are accomplished via “contextualization,” a process by which job requirements are used to make minor configuration changes, such as network and DNS settings, to an existing appliance [18, 92]. Contextualization is done once per VM invocation, just prior to boot time, and involves injecting configuration data into the VM image prior to initialization of the instance. A centralized “context broker” provides an XML description of the configuration parameters to be set on a per-appliance basis. [91] This contextualization process may occur during workspace scheduling [68].

For performance reasons, workspace jobs should have sufficient run length so as to make the overhead of leasing and starting a workspace relatively small. Given the non-trivial data copy overheads involved with constructing a workspace, it has been proposed that overhead be treated as a property of the job execution site, instead of charging the overhead to the user as part of the resource allocation [146]. Total execution overhead for Virtual Workspaces has not been measured; instead, the potential benefits of Workspaces as cloud computing containers have been demonstrated through comparison of local performance to the performance of a larger, more powerful system made available via the grid [80]. Providing a combination of pre-arranged and best-effort leases has been shown to have better aggregate performance than a traditional best-effort scheduler without task preemption, while approaching the performance of a traditional best-effort scheduler with task preemption [147][148].

Virtual Organization Clusters differ from Virtual Workspaces, in that the VO assumes responsibility for the creation, configuration, and maintenance of the virtualized execution environment. These VOCs are then autonomically scheduled on participating physical fabric sites, eliminating the need for the explicit leases used in VWs. VOCs are therefore transparent to the end

user, who submits jobs using standard grid middleware. A single VOC requires only one VM image to spawn as many compute nodes as are needed.

4.1.1.2 Shirako and Cluster-On-Demand

Shirako [44] and Cluster-On-Demand (COD) [43] are related projects that implement a lease scheduling mechanism and back-end cluster provisioning system, respectively. Shirako utilizes a lease system that enables resource providers and consumers to negotiate resource allocations through automated brokers [83], which are designed around a self-recharging currency model [82]. COD provides back-end provisioning of physical resources, so that physical sites may be divided among different user groups. This division is accomplished either by reinstalling the physical compute nodes with user-specific software [32] or by virtualizing the installations using Xen guests [78]. In either case, a customized software load is installed onto the provisioned compute elements, using a user-supplied resource provisioning request [32]. Xen VMs installed in this manner may be migrated between physical systems for load-balancing purposes [78].

An application of Shirako, called the Grid Resource Oversight Coordinator (GROC), permits physical resources to be leased by individual Virtual Organizations for the purpose of deploying completely virtualized grids. VO users interact with these virtual grids using existing middleware tools for job submission and retrieval, resulting in submission endpoint transparency. However, GROC adds a new layer of software and scheduling to existing physical systems, essentially constructing a new overlay grid on top of the existing grid. Hosted virtual grids utilize a virtual head node on each physical grid site, without spanning of individual clusters across grid sites. These virtual grids are scheduled for execution on physical fabric by means of Shirako leases, which may be best-effort or pre-arranged with temporal guarantees. Back-end provisioning of physical resources for hosting the virtualized grids is performed using COD, with virtual grids running directly on the hardware or in Xen containers. [127]

Virtual Organization Clusters differ from Shirako-leased systems, including GROC, in both design and intent. VOCs are *clusters* of virtual systems, which can span multiple grid sites on existing infrastructures, while GROC creates overlay grid sites on single physical sites without spanning. Shirako is also fundamentally a leasing model, like Globus Virtual Workspaces, while VOCs provide no mechanism for prearranged resource leases. In this regard, VOCs are designed to be strictly “best-effort” systems that can be deployed in a manner completely transparent to the users and

optionally transparent to the VOs. GROC permits user-level transparency, but it requires the explicit participation of both VOs and physical sites. In addition, VOCs require the use of VMs as containers, while the COD back-end in GROC makes container use optional.

4.1.2 Autonomic Models

In contrast to leasing systems, autonomic grid virtualization models are transparent to the user. Whenever the user submits a job, the system automatically provisions itself to handle the workload, as depicted in figure 4.2. Existing autonomic grid virtualization systems include VioCluster (section 4.1.2.1), Dynamic Virtual Clustering (also presented in section 4.1.2.1), and Violin (4.1.2.2). As VioCluster and Dynamic Virtual Clustering have the same general objective, they are presented together, although they are developed by unrelated groups.

4.1.2.1 VioCluster and Dynamic Virtual Clustering

Two similar projects from unrelated groups are VioCluster and Dynamic Virtual Clustering, both of which are designed to improve the utilization of independent research clusters within the same organization. The earlier project, VioCluster, is based on the concept of dividing each cluster into a physical and a virtual domain, where the virtual domain may be borrowed and administered by another research group. Virtual domains are transparently and autonomically resized by means of a broker application, which trades machine allocations between groups by following explicitly configured policies. This brokering process is transparent to end-users, permitting the virtualized cluster to be used as if it were a regular physical cluster. User Mode Linux (UML) provides virtualization capabilities, while the Violin overlay network (see section 4.3.3) joins the borrowed virtual nodes to the borrower’s physical cluster, providing a seamless private network. [134]

VioCluster base images are pre-staged onto the physical hosts to be borrowed, and these images are essentially permanent in nature. At lease initiation time, a small (on the order of 200 KB) binary patch file is transmitted to the physical hosts. This file contains configuration information for the virtual cluster, and it is applied to the static configuration image using a copy-on-write (COW) technique. COW allows the VM to be terminated by means of killing the containing virtualization process, resulting in fast shutdown performance. Boot times are measured to be 40 seconds, with shutdown times of 16 seconds. Execution overheads of 15% are observed using the High Performance Linpack (HPL) benchmark. [134]

Figure 4.2: In an autonomic grid architecture, the user submits jobs exactly as she would submit them to a traditional grid architecture (steps 1, 2, and 3). An autonomic management system automatically starts and manages virtual machine instances according to some algorithm. User jobs are then automatically routed into the virtual environments by some method, perhaps an extension to existing middleware.

Like VioCluster, Dynamic Virtual Clustering allows clusters of virtual machines to be instantiated on a per-job basis for the purpose of providing uniform execution environments across clusters co-located on a single research campus. Xen is used as a hypervisor, with a modified version of Moab employed for scheduling [48]. DVC facilitates sharing of different research clusters in a Campus Area Grid (CAG) by allowing for temporary software environment customization through virtualization, as well as forwarding and spanning jobs between clusters. [50] Clusters constructed with DVC can support VM-level checkpointing that is transparent to the job being executed, thus providing fault recovery without modifications to the application [49].

A distinguishing design element of DVC is the concept of the Campus Area Grid, which is defined as “a group of clusters in a small geographic area ... connected by a private, high-speed network” [50]. Latency and bandwidth properties of the private network are considered to be favorable for HPC jobs, thereby allowing a combination of spanned clusters to function as a single high-performance cluster for job execution. However, the software configurations of the different component clusters may differ, as the component clusters may belong to different entities with different management. DVC permits Xen VMs with homogeneous software to be run on federated clusters on the same CAG whenever the target cluster is not in use by its owner, thereby allowing research groups to increase the sizes of their clusters temporarily. In terms of performance, spanned clusters using DVC were found to have 20% overheads on the PTRANS benchmark and negligible overheads on the HPL benchmark (matrix sizes were not provided). Use of DVC permitted a synthetic test load of long-running jobs to achieve a 57% throughput increase with a corresponding 33% reduction in turnaround time. [47]

Like Virtual Workspaces, In VIGO, and Shirako, DVC clusters utilize one VM image per compute node, and this image is staged onto the host machine prior to booting the VM instance. Image staging is built into the modified Moab scheduler, which is also responsible for scheduling both the VMs and user jobs. Due to the need to stage a VM image and boot a VM instance on a per-job basis, overheads of up to 60% in terms of throughput, and 80% in terms of turnaround time, were observed for short-running “transactional” jobs. Compared to the 118 seconds required to stage an image, 23 seconds required to boot an instance, and 11 seconds needed to terminate a VM, these transactional jobs would run for only 5 to 10 seconds. An optimization to improve this behavior would be to pre-arrange a lease for a VM dedicated to the execution of short jobs, thereby avoiding the image management overhead. [50] Caching a VM image for a fixed period time after

first use could reduce overheads substantially [51].

Unlike VioCluster and DVC clusters, Virtual Organization Clusters are explicitly designed for use on wide-area grid systems, which lack high-bandwidth, low-latency interconnects between physical sites. VOCs are ideally designed to use a distributed filesystem for a single VM image, which is shared among all VM instances on the local site. This combination eliminates both image staging and the need to perform an orderly shutdown, which would save 129 seconds of overhead if employed in DVC (16 seconds in VioCluster). In addition, the VOC Model is designed to support High Throughput Computing (HTC) applications instead of High Performance Computing (HPC) applications, reducing concerns about job turnaround times. Like VioCluster, VOC nodes may share a private network across sites through the use of an overlay network, as described in section 4.3.3.

4.1.2.2 Violin

Although primarily a network overlay system, Virtual Internetworking on Overlay Infrastructures (Violin) is extended for provisioning isolated networks of virtual machines hosted on a physical cluster system. Violin and VioCluster are designed by the same research group, but Violin focuses on using an overlay network to span multiple sites, while VioCluster focuses on sharing resources at the same physical site. [133, 134] Violin-enabled virtual clusters may use either User Mode Linux (UML) or Xen as the underlying virtualization technology [136]. Violin clusters may be created on demand, customized for specific applications, and used to isolate virtual systems from physical systems. High Performance Linpack (HPL) benchmarking results show a virtualization overhead of 20% with UML; however, the matrix sizes used in the calculations are not scaled with increasing cluster sizes. [133]

Clusters constructed with Violin are capable of autonomic adaptation to their execution environments. Local autonomy is achieved through the use of monitoring daemons, which dynamically adjust CPU and memory allocation to individual VMs. Distributed autonomy is effected by means of live VM migration, permitting a running VM to be moved from one physical host to another. As is the case with other virtual clustering technologies based on Xen, Violin requires the use of a single VM image per VM instance, and that image may be live-migrated between physical hosts during execution. Performance improvements are observed for workloads that require or can utilize excess computational resources beyond those initially allocated at VM creation time. Autonomic increases in resource allocation, and migration of virtual execution hosts to more powerful hardware, permit

a reduction in total job execution time for these workloads. [135, 166]

Virtual Organization Clusters share some design similarity with Violin. VOCs are designed to be autonomically deployable across different grid sites, but the desired autonomy in VOCs arises from the ability to shrink or expand the total size of a VOC, rather than adjusting the resource allocations of its component VMs. While VOC nodes will use an overlay network to span physical sites, the design emphasis of the VOC Model is not the networking layer itself, while Violin is primarily a networking layer that supports the creation of virtual environments. VOCs may utilize any overlay networking technology, including Violin.

4.1.3 In-VIGO

The In Virtual Information Grid Organizations (In-VIGO) project attempts to raise the level at which grid middleware operates from sharing physical systems to sharing virtualized resources on physical systems. As presented in [7], In-VIGO conceptually defines three major benefits of grid virtualization: multiplexing, manifolding, and polymorphism. Virtualized systems can appear as multiple different systems (manifolding) with potentially different capabilities (polymorphism) through multiplexing physical systems to support multiple simultaneous virtual machines. Three separate layers are specified in the In-VIGO design: low-level operating system virtualization to create virtual machines, grid middleware to connect application interfaces to the back-end VMs, and a top-level application layer to present the specific Grid-enabled scientific application to the end user. To provide data and connectivity to these applications, In-VIGO specifies per-application virtual filesystems and overlay networks. [7] Virtual filesystems are realized through the use of the Grid Virtual File System (GVFS) [174], while overlay networking is provided by ViNe [109, 158].

Virtual machines used by In-VIGO are constructed using VMPlants [99], an automated VM installation and cloning mechanism. A VMPlant is a system that prepares and installs VM images using a cost model to provide a “bid” to a centralized coordinator called a VMShop. The user sends an XML specification with the desired properties of the installed system to the VMShop, which collects bids from the various VMPlants and schedules construction of the individual VMs via a minimum-cost algorithm. Different VMPlants have different image stores, and partial images are collected over time on each VMPlant system. Thus, one VMPlant may be able to produce a desired VM configuration at a much lower cost than another VMPlant, due to the ability to re-use existing image components when producing a new image. VMPlant-generated VMs have a

flexible configuration selectable from a pre-determined allocation of “golden” images, which consist of VM images with manually preinstalled operating systems. Actions specified in the VMShop XML specification are partially ordered so that the VMPlant construction process can emulate that of a human system administrator at coarse granularity. [99]

Once instantiated, In-VIGO systems are directly accessed by the end user via a Web-based portal that permits data transfer into and out of the the system [175] and enforces permissions on shared resources through the use of a single sign-on short-term credential mechanism [8]. While an In-VIGO application can be targeted to a single Virtual Organization, such as NanoHub [59], each individual application is typically realized as a separate In-VIGO instance with application-specific virtual systems [7, 59, 110]. Multiple applications can be co-scheduled on shared hardware via best-effort multiplexing [7] or through the use of a fault-tolerant per-application deadline scheduler [167].

A key distinction between In-VIGO and Virtual Organization Clusters is that In-VIGO provides a per-application end-to-end system with which the end user directly interacts via a custom interface. Virtual Organization Clusters provide services to end users transparently: that is, end users run applications using existing grid middleware technologies and existing grid interfaces. VOCs provide services to individual Virtual Organizations and multiplex physical resources to permit multiple VOs to share the same hardware. Individual VOCs are then multiplexed among the different end-user jobs by means of scheduling systems installed in the VOCs. Dedicated VOC administrators are responsible for making configuration decisions, including specification of VM image contents and virtual network overlays. The VOC Model does not specify a single mechanism for VM image creation, so VMPlants could be used to prepare the system image for any given VO. Moreover, the VOC Model does not limit the choice of middleware systems to be installed on either the physical or virtual systems. Thus, it would be possible to implement an application-level grid computing system such as In-VIGO by using Virtual Organization Clusters as a back-end.

4.2 Simulation Systems

Simulation systems for modeling grid architectures and testing solutions to grid-related problems may be classified into a taxonomy as described in [152]. This taxonomy, hereinafter called the Sulistio Taxonomy, consists of a set of sub-taxonomies that address specific features of

the simulation design. Of particular importance are the distinctions between static and dynamic simulations, discrete and continuous output, serial and multithreaded execution, deterministic and non-deterministic (probabilistic) behavior, trace-driven and event-driven simulation kernels, entity-based and event-based grid models, and the software interface utilized to run the simulation. Static simulations ignore the passage of time, emphasizing changes in the overall system from one state to another state, while dynamic simulations are time-driven and report measurements over time. These measurements may consist of discrete observations from a finite set of possible values, or of a measurement from a continuous range. Execution of a simulation experiment may be serial (single-threaded) or multithreaded. Simulation results may be deterministically reproducible or randomly variable from run to run. Internally, the simulator cores may be driven by discrete events or by input traces, while the grid systems under simulation may be internally modeled by software entities, events that represent communications between entities, or a combination of both. Finally, simulation systems may be further classified by the interfaces they present to users, which may consist of a library of components designed for programmers or a custom simulation language that encapsulates the simulator into a stand-alone application. The software architecture of the system is especially important in the case of simulation libraries; the Sulistio Taxonomy classifies this architecture as either structured or object-oriented. [152]

Among the grid simulation systems classified in Sulistio et al. [152] is SimGrid [29, 30, 70, 162]. SimGrid is a static, serialized, deterministic, trace-driven discrete event simulation system with both an entity-based and an event-based model [152]. SimGrid is implemented as a structured code library, around which different simulation experiments are implemented using the C or Java programming language [30, 152]. Four core Application Programming Interfaces are exposed to client applications: SimDag, for investigating task graph scheduling heuristics; MSG, for simulating concurrent sequential processes; SMPI, for simulating MPI applications; and GRAS, for emulating a grid system to run actual grid applications under simulation [30]. As originally designed, SimGrid provides a framework for evaluating different grid scheduling models [29] while simultaneously providing realistic macro-scale simulation of network behavior, particularly for Transmission Control Protocol (TCP) communications streams [70, 162].

Another simulation system for modeling grid scheduling is GridSim [25], which operates at a higher level of abstraction than SimGrid. GridSim is a static, multithreaded, deterministic, event-driven discrete event simulator, with both entity-based and event-based models, accessible either

by programming a driver using an Object-Oriented library or by utilizing a form-based Graphical User Interface [152]. GridSim is implemented in Java and provides core components for modeling computational resources and costs, including detailed simulations of process execution behavior. Networking aspects of the grid are abstracted to communications speeds between node sites. [25] GridSim has been extended to model the transmission and staging of large amounts of scientific data in a Data Grid architecture [151]. Further refinements to scheduling simulations are implemented in the Alea [98] simulator. A recent extension to GridSim, called CloudSim, enables the simulation of virtualized data centers to assess resource allocation algorithms, virtual machine migration for reliability concerns, and scalable resource management beyond what is available with existing physical cloud implementations [26].

The Bricks simulator [154] models scheduling algorithms for global High Performance Computing (HPC) applications. According to the Sulistio Taxonomy, Bricks is a static, serialized, deterministic, event-driven discrete event simulator. Bricks can utilize both entity-based and event-based models, and operation of the simulation system is performed by scripting experiments using a custom Object-Oriented interface. [152] Implemented in Java, Bricks simulates network and processing resources at a high degree of fidelity, for the purpose of testing different global scheduling algorithms.

At the opposite end of the spectrum from application simulators are grid system emulators. MicroGrid [145, 165] is an emulator for unmodified Globus [64] applications. The simulation kernel of MicroGrid is implemented as a dynamic, multithreaded, deterministic, event-driven discrete event simulator with continuous sampling. Entity-based and event-based models are supported for simulation drivers written to utilize a custom structured interface. [152] MicroGrid virtualizes resources such that Globus applications are presented with the same environment present on an actual grid system. Test results from micro-benchmark suites run against Globus applications both on actual grid systems and in the simulated environment indicate high simulation fidelity at the application level. [145] Globus applications, written in a wide variety of languages, can be tested on a wide variety of simulated grid systems to measure performance characteristics [165].

Other purpose-designed simulation systems are not classified in [152]. BeoSim is a simulation system for local grids created from different clusters on the same research campus, where compute nodes on each cluster are borrowed for use with other clusters [86, 85]. GSSIM [100] enables multilevel scheduler simulation using actual cluster traces in the Standard Workload Format or Grid Workload Format. OptorSim [27] is designed to simulate grid job and file interactions, so that scheduling

and file replication tasks may be optimized. GangSim [45] is yet another grid scheduling simulator, which emphasizes interactions between grid-level job scheduling interfaces and local site scheduling and resource allocation policies.

While each of these simulation systems has the capability to model scheduling algorithms for different grid architectures, a new simulation system has been developed in this dissertation for testing VOCs: the Simulator for Virtual Organization Clusters (SimVOC), which is designed to model the aggregate behavior of an existing grid system that uses commodity scheduling algorithms. As SimVOC is able to execute simulations both with and without the addition of virtual machines and extra middleware, differences between existing grid systems and grid systems with deployed Virtual Organization Clusters may be evaluated. Although SimVOC is a purpose-driven simulation system, its Object-Oriented implementation using the Python language permits extensibility to other simulation tasks.

Utilizing the Sulistio Taxonomy [152], SimVOC is a dynamic simulation system with continuous output capabilities. By providing a centralized random source with user seeding support, the serialized SimVOC kernel is able to provide both deterministic and non-deterministic execution, depending upon the requirements of each simulation experiment. While the SimVOC kernel is fundamentally event-driven, a subset of events directly corresponds to an input trace. Grid models used within the simulator are hybrid entities capable of both creating and receiving events. SimVOC is presented to simulation consumers as a modular, object-oriented Python library.

4.3 System Resources

Whether computational jobs are executed in virtual machines or directly on physical systems, hardware resources ultimately must be provisioned and assigned first to specific user groups, and then to individual jobs. Dynamic provisioning of resources is desirable, as it allows resource allocations to match job requirements more closely. However, resource allocation is a difficult issue because application resource demands change during the course of execution, typically exhibiting multiple phase transitions [171]. Ideally, provisioning should be both dynamic and autonomic, eliminating the requirement to specify resource needs in explicit lease requests. This dichotomy in resource provisioning is characterized by competing solutions such as Violin (autonomic allocation) [135] and Virtual Workspaces (leasing) [94].

Regardless of the resource provisioning scheme, several classes of resources are needed to effect virtualized clustering with site spanning. These resources include networking, migration capabilities, and distributed filesystem support for image hosting. The scheduling of jobs to utilize these resources is important, as the choice of scheduling system can have significant impacts on overall system performance. In the remainder of this section, scheduling is discussed (subsection 4.3.1), followed by a survey of pilot job frameworks (subsection 4.3.2). Descriptions of available technologies for implementing VOCs follow, including overlay networking (subsection 4.3.3), live migration (subsection 4.3.4), and distributed storage (subsection 4.3.5).

4.3.1 Scheduling Resource Use

Sharing cluster resources requires allocating those resources to individual jobs on a short-term basis, temporally multiplexing the resources over the set of jobs. High Throughput grid jobs may be viewed as a bag of tasks, where each task is an independent component of a common parallel application. Scheduling bag-of-tasks applications in order to minimize the makespan, or time from the start of the first job in the bag to the completion of the final job, is known to be NP-hard in the general case [13]. For systems representable by a star network, which includes most cluster systems with a single head node and multiple compute nodes, both makespan minimization (HPC) and throughput maximization (HTC) are NP-complete scheduling problems [102]. When different resource quantities can be scheduled for different users with different utilities, the problem of provisioning is an NP-hard knapsack problem [78].

A desirable alternative to makespan minimization scheduling is steady-state scheduling, which attempts to optimize resource usage for high performance over a period of time by adjusting fractional usage of specific resources by individual jobs [16]. While this type of scheduling shows benefits for high-performance applications, steady-state throughput maximization is still NP-complete in the general case [15]. When the computational resource has the form of a tree of depth greater than one, distributed scheduling becomes an alternative to centralized scheduling. In the distributed case, individual non-leaf child nodes in the tree perform local scheduling for the star network consisting of leaf nodes rooted at that tree. Although distributed scheduling may be preferable to centralized scheduling in terms of algorithmic complexity, testing indicates that centralized scheduling, when possible, is still more efficient. [17] Effective testing of general scheduling algorithms can be performed within grid simulation systems, instead of on actual grids, with reasonable accuracy

and much greater flexibility [103].

In practice, applications known to fit a bag-of-tasks model may not always be executed on grid systems, owing to the complexities inherent with matching application requirements to available systems through a manual job submission process. A solution to this problem is presented in [33], which describes MyGrid, a custom grid implementation for running bag-of-tasks applications on whatever hardware is available to the user. MyGrid features a replicated work queue scheduler, which matches tasks to available processors in an arbitrary order, without *a priori* knowledge of the environment [33]. The MyGrid middleware is Java-based, portable, and capable of interfacing with Globus for general purpose grid connectivity. However, it does not utilize virtualization. [36]

Virtual Organization Clusters appear to be ideally suited for bag-of-tasks applications with current technologies, as virtual machine performance testing has shown that overhead minima tend to occur in compute-bound situations [14, 54, 55], and communications within bag-of-tasks applications are generally limited to task initiation and completion [17]. However, future advances in virtualization technology may improve performance for other classes of applications.

4.3.2 Pilot Job Frameworks

Pilot job frameworks are a mechanism for leasing computational resources by means of submitting a grid job that contacts a server at execution time, retrieving a workload from that server and thus bypassing the normal mechanisms for running jobs on the grid (section 2.4). Users of grid systems, particularly those from the High Energy Physics (HEP) community, often prefer to submit jobs to Workload Management Systems (WMS) instead of directly to grid sites, since the WMS autonomically matches the job to a site. Since this process bypasses normal grid accounting and security systems, an accounting system for pilot frameworks has been developed for the Open Science Grid (OSG). This system essentially places a grid gatekeeper on each compute node, checking the authorization of each pilot job arriving from the WMS. [143]

Several Workload Management Systems have been developed for different grid applications, including DIRAC, Condor Glideins, and PanDA. DIRAC (Distributed Infrastructure with Remote Agent Control) is deployed at CERN as part of the LHCb project charged with large-scale simulation of data associated with the Large Hadron Collider. DIRAC is designed to support High Throughput Computing (HTC) applications using a pull model that finds jobs appropriate for available resources, instead of locating resources to match to jobs. A central WMS server is deployed at CERN, to which

user jobs are submitted. Each worker node using DIRAC runs an agent that functions as the pilot job, retrieving matching workloads from the WMS. [157] Condor Glideins support similar HTC applications on commodity grid systems, except that jobs are submitted to a local Condor pool and matched to compatible systems on the grid using Condor-G. [142]

PanDA differs from DIRAC and Glideins in that it chiefly supports High Performance Computing (HPC) applications. Used by the ATLAS High Energy Physics Virtual Organization, PanDA receives jobs into its WMS via a Python client. In the base implementation of the WMS technology, PanDA uses a pilot framework with Condor and Condor-G to find matching resources on which jobs are executed. [5] PanDA supports virtualization with the Xen hypervisor, using a CERN-developed interface called the Virtualization Resource Management (VIRM) API to request resources. In turn, VIRM uses the CERN vGrid layer to lease a physical node and start a virtual machine. VIRM is extensible to several back-ends including Globus Nimbus. [97] Coupled with the autonomic leasing back-end provided by PanDA, this technology provides a rudimentary Virtual Organization Cluster to the ATLAS VO.

4.3.3 Network Overlays

Several networking libraries have been developed for virtual machines, which permit virtual clusters to use networks logically isolated from the underlying physical hardware. Virtual Distributed Ethernet (VDE) [38], Virtuoso [153], and ViNe [158] provide low-level virtualized networks that can be utilized for interconnecting VMs. Wide-area connectivity of VMs can be achieved through the use of overlay networks such as Wide-area Overlays of virtual Workstations (WOW) [73, 164, 74] or IP over P2P (IPOP) [72].

Virtual Internetworking on Overlay Infrastructures (Violin) enables the creation of isolated virtual private networks through the use of virtual hosts, virtual switches, and virtual routers. Violins are isolated from the underlying IP network and are recursively layerable, thereby allowing one Violin to be hosted within another. Virtualized networks inside Violins have their own routing and protocol stacks, allowing for wide discretion in network configuration. Throughput overheads are measured at 5% using TCP data streams. [84] Traffic bounds and quality of service adjustments may be made on a per-virtual host basis [133].

IP over P2P (Internet Protocol over Peer-to-Peer: IPOP) is an overlay network that provides scalability, resistance to single bad links, and NAT traversal. IPOP works by encapsulating IP traffic

inside the Brunet peer-to-peer networking system. Brunet implements a version of the Address Resolution Protocol (ARP) to permit IP address mobility within IPOP, improving support for live migration of VMs connected to the overlay. Within a Local Area Network (LAN), IPOP performs poorly, with an order-of-magnitude increase in packet Round-Trip Time (RTT) and a reduction of available bandwidth to 29% of the physical total bandwidth. However, over a Wide Area Network (WAN), the packet RTTs are increased by only 33%, and available bandwidth is up to 75% of physical total bandwidth. [72]

For latency-sensitive applications such as MPI [108], network performance is a substantial concern when virtualizing cluster systems. As presented in [71], measurements of network performance obtained within VMs may be skewed by the virtualization process itself. Context switches, which occur whenever packets arrive at a network interface and generate interrupts, have a negative impact on network and system performance as a whole. Strategies for improving network performance include enabling interrupt coalescence and larger frame sizes [75], as well as bypassing the host operating system when delivering network packets to a virtualized guest [106]. Networking issues are less important for high-throughput bag-of-tasks applications, which have limited communications requirements [17].

4.3.4 Virtual Machine Migration Support

Live migration between physical nodes permits a virtual machine to be transferred from one physical host to another without terminating the VM or the applications running within the VM, allowing virtual grid systems to implement features such as checkpointing [49] or autonomic resource adaptation [136]. This migration is performed using an incremental memory copy procedure, and it produces only a momentary pause in VM execution on the order of a few tens to hundreds of milliseconds. [34] One limitation of live VM migration is that both the origin endpoint and the target endpoint must be on the same subnetwork, so that the VM does not lose networking connectivity during the migration process. This subnet limitation can be avoided by using IPv6 with mobility extensions to permit VMs to be migrated over a wide-area network such as the Internet [79]. Alternatively, use of an overlay network may allow a VM to remain in the same private subnet, even though it is migrated to a completely new network, as is possible with VMs that use the Violin overlay [135].

The Virtual Organization Cluster Model does not require, nor does it prohibit, the avail-

ability of live VM migration. Since individual VOC nodes are given a static resource allocation, and the mechanism for increasing the total resource allocation for a VOC is to expand the size of the VOC by booting additional VMs, there is no need for live migration for the purpose of increasing allocated resources. However, a VOC might use live migration to implement checkpointing or to move jobs to more optimal hardware if such hardware becomes available after a job starts. Specific implementations of VOCs may choose to provide or require migration capabilities.

4.3.5 Distributed Filesystems

A distributed filesystem allows virtual clusters to avoid staging images onto host machines, eliminating the delay caused by the image copy operation. Distributed image stores have been recommended since the first paper on grid system virtualization [57], but most grid virtualization systems still rely on one locally staged VM image per VM instance. Even where the hypervisor technology requires a single image per instance, reading the images directly from a Network File System (NFS) share, without staging, may show performance gains [175]. Since NFS shares single disks, and single disks may become bottlenecked under multiple concurrent reads, a multiple-disk parallel filesystem may provide better scalability. The Lustre filesystem [144], equipped with high-speed interconnects [169], could be used for this purpose. Other parallel filesystems, such as the Parallel Virtual File System (PVFS) [28], may also be used.

The Virtual Organization Cluster Model does not require the use of a distributed filesystem (DFS), but use of a DFS is recommended for improved performance when VOC sizes change frequently. Ideally, each physical grid site provides a DFS from which copy-on-write images may be accessed directly, with read-only permissions. All writes are non-persistent and are performed to the local disk on which the VM instance is executing. Any distributed filesystem may be used as an image store, as long as the read speeds are shown to be sufficient when multiple VMs are booted simultaneously from the same shared image file. As a result of testing the relative ease of maintenance of both Lustre and PVFS on the prototype cluster used for this research, PVFS has been chosen as the distributed filesystem for the prototype implementation.

4.4 System Management

Related work on system management can be divided into three categories: physical system installation (subsection 4.4.1), virtual cluster construction (subsection 4.4.2), and post-installation management (subsection 4.4.3). Physical fabric is needed to host Virtual Organization Clusters, and these physical clusters must be configured with software for virtualization, network connectivity, and possibly other support services. Construction of virtual clusters may be simplified by exploiting the single-image property of VOCs. For both administrative domains, post-installation management of the cluster systems will be needed to update installed software and add or change existing functionality to adapt to the changing needs of the hardware site or Virtual Organization.

4.4.1 Installation of Physical Systems

Among the most popular physical system installation toolkits is NPACI Rocks [131]. The core of the Rocks system consists of a rapid installer for an underlying Red Hat-derived Linux distribution along with cluster-specific tools packaged for rapid installation. Rocks is designed to minimize the scalability issues inherent with managing a cluster system by centralizing configuration on the head node and treating compute nodes as stateless systems. [122] Entire clusters can be installed with Rocks in a matter of hours, provided the basic setup of the cluster hardware is suitable for the installation [123]. Standard tools, such as Red Hat Kickstart, are leveraged by means of an XML-based specification, so that clusters can be automatically configured according to a pre-specified design [88], which can be modified and extended to permit customized installs for specific cluster applications [137]. Rocks provides a mechanism for easy addition of software application groups via “rolls,” or meta-packages of related programs and libraries, grouped according to software dependency graphs [20].

Other systems for rapid physical cluster installation include the OSCAR meta-package system, RISE, and Cluster-On-Demand. OSCAR is an automated installer that uses groups of packages (“meta-packages”) to add software to cluster compute nodes in a manner similar to Rocks rolls [112]. RISE is a Web services-based system imaging suite, which propagates binary software images to cluster compute nodes using the Preboot eXecution Environment (PXE) [101]. Cluster-On-Demand (COD) permits dynamic reinstallation of all or part of a physical cluster, allowing the cluster to be split into multiple virtual cluster slices [32].

Each of these systems provides rapid installation and initial configuration of physical compute fabric, provided the hardware to be used is supported by the rapid installer. Experiential tests with some of these systems, particularly Rocks and OSCAR, has led to intractable failures when the hardware was not fully supported by the installation suite. Furthermore, these systems make little provision for the ongoing maintenance and updating of cluster systems, short of performing complete reinstallations.

Virtual Organization Clusters require little in the way of support from the host operating system: virtualization support, access to shared storage, and networking connectivity are the principal requirements of the virtual machine. As a result, VOCs can be run on simplistic host installations using lightweight distributions such as Slackware Linux [55]. Many of the meta-packages managed by Rocks, OSCAR, and similar systems are simply unnecessary on physical hardware that will support VOCs. The software complexity arising from the various scientific software package sets is moved into the virtual machine space, eliminating the need for the vast majority of cluster software to be installed directly on the hardware.

4.4.2 Installation of Virtual Systems

Virtual Machine installation systems tend to be designed to produce a large number of identical, or mostly identical, VM images for use with virtualized grids. VMPlants generates a set of customized VMs by means of a directed acyclic graph specification. A cluster of VM-producing systems is used for actual image creation, with load balancing accomplished via a bid-cost model. [99] Virtual Workspaces may be created by a factory system using an XML specification for customization [94]. Factory systems themselves may be realized by specification-driven installers that permit software customization either at install time or by using standard grid tools [168]. Performance of these factory systems may be enhanced by using aggressive image and partial image caching strategies, significantly reducing installation time [120].

VM installation for Virtual Organization Clusters could be automated, and in such a case, one of these existing solutions could be utilized to produce the VM image. However, a single VOC uses only one compute node image, the creation of which would not be sufficient to produce full utilization of a cluster-based installer. Since the single image will be used to boot as many instances as are needed, and replication will occur only when the same image is used at multiple different physical sites, manual installation of the image will be practical. The only requirements for a VOC

image are that it be in a format compatible with the virtualization system in use, and that it be capable of dynamic configuration of its networking properties, such as IP address and hostname, at boot time using DHCP.

4.4.3 Post-Installation Management

Once a system is installed and operational, it is necessary to perform updates and make configuration changes as required for system security, stability, and feature additions. Software and configuration changes can be made through several mechanisms, such as the addition of ROCKS rolls [20], or the use of remote command execution systems such as the Cluster Command and Control Suite [58] or Tentakel [156] to push changes to the compute nodes. Alternatively, nodes can be completely re-imaged using a tool such as the System Installation Suite [37]. All these “traditional” techniques are best suited for the management of physical systems within a single site. With the exception of the ability of the Cluster Command and Control Suite to handle multiple individual clusters [58], these tools tend to focus on batch management of entire systems simultaneously, omitting fine-grained targeting of individual systems. Moreover, a standard practice for upgrading existing cluster systems is complete re-imaging of the compute nodes, which are presumed to be stateless [122].

A non-imaging mechanism for post-installation management of systems is Cfengine [24], which utilizes a descriptive language to configure autonomous agents. These agents communicate with a central policy server to effect configuration changes via a convergent process [22] in which each machine tends to move toward a desired state. Although Cfengine allows different groups of systems to be targeted by specifying policies around system classes, it is still designed around the concept of single-site management. Cfengine also requires dedicated services and transports to operate, which might cause difficulties when spanning grid sites. [24] However, Cfengine is able to detect and correct configuration anomalies [21], and it uses a hybrid feedback loop to move the system configuration toward the desired state [23]. Applications of Cfengine include middleware deployment and file synchronization [111], as well as network device management with added components based on the Simple Network Management Protocol (SNMP) [140].

An alternative agent-based configuration management system is Quattor. Developed by entities related to the Large Hadron Collider project at the European Organization for Nuclear Research (CERN), Quattor is capable of both automated installation (via Red Hat Kickstart) and

post-installation management. Quattor is designed around a large set of small agent components, each of which performs a single configuration task. The Pan language is used to specify configurations, while target machines may be specified in plain text, XML, HyperGraph, or GraphViz format. A central database aggregates configurations into profiles for retrieval by the Quattor-managed systems. [31]

Another automated mechanism for change detection and configuration updates is Bcfg2 [12], which was developed to unify the management of compute and service nodes in cluster systems [42]. Unlike Cfengine, which uses a custom descriptive language for expressing configuration states, Bcfg2 utilizes “generator” sub-applications to produce remote configuration specifications [39], which are matched to system groups expressed in an XML format [87]. Bcfg2 accommodates changes over time through integration with Subversion repositories and other change management tools, improving flexibility [40]. However, despite an explicit design goal of simplicity in comparison with other tools [39], a full-scale test deployment of Bcfg2 still required 4 months [41].

The division of administrative space that occurs with Virtual Organization Clusters presents new issues for system management. Virtual clusters are not tied to specific hardware or specific sites, but individual VOCs may be long-lived and require in-place updates to address security concerns or adjust capabilities. Complete reconstruction of a virtual cluster, while feasible given the single-image nature of VOCs, may not be desirable for small changes due to the need to propagate a large image over a wide area network. Moreover, simplification of the physical administrative domain may obviate the need for extensive change-management systems, instead suggesting simpler strategies for updates and reconfiguration.

Chapter 5

Virtual Organization Cluster Model¹

The Virtual Organization Cluster Model (VOC Model) describes a novel grid architecture that provides execution environments that are homogeneous across grid sites, allowing applications to be run on any ISA-compatible resource, including those resources that do not provide the necessary software libraries to run user jobs directly on the hardware. VOCs remain transparent to end users and disinterested entities, permitting the new architecture to be deployed in conjunction with, and without disrupting, existing production grids. Since design decisions and issues in a grid system are by definition distributed and at scale, abruptly changing the architecture of an existing production grid would be impractical in terms of cost and complexity. Thus, it is necessary for a new grid architecture to be deployable in a phased manner that does not cause widespread disruption. VOCs achieve phased deployability by leveraging existing middleware and permitting *participating* and *non-participating* (or *transparent*) deployments. In a participating deployment, a VOC is created for a single VO, which assumes administrative responsibility for the VOC. All participating grid sites must permit virtual machines to be spawned to create VOC compute nodes. Conversely, in a non-participating deployment, a single grid site transparently provides VOCs for VOs authorized to run jobs on the site, encapsulating the end-user workloads in local virtual environments. These non-participating VOCs are managed by the site administrators, and it is not necessary that VOs or end users even be made aware that VOCs are in use. In either case, middleware changes are needed only on the sites that will host virtual machines. No changes to submission endpoint middleware (the software operated by the end user) are required.

¹Portions of this chapter will appear in *Future Generation Computer Systems*.

Virtual Organization Clusters are formally defined according to a set of abstract specifications, which are presented in this chapter. This set of specifications is intended to be flexible, so that actual implementations of VOCs can be accomplished using a wide variety of technologies for virtualization, networking, job scheduling, and autonomic adaptation. Any implementation that satisfies the requirements of the Virtual Organization Cluster Model is a Virtual Organization Cluster. In other words, the term “Virtual Organization Cluster” describes a *concept* as opposed to a specific implementation or product.

A description of the Virtual Organization Cluster Model (VOC Model) depends upon a separation of administrative domains (section 5.1), which conceptually divides responsibility for the computational environments between the physical resource provider and the Virtual Organization or its representative (which may be the physical resource provider in transparent implementations). Definitions of terminology necessary to the model are presented in section 5.2. The abstract definition of Virtual Organization Clusters is presented in section 5.3. Finally, section 5.4 explains how VOCs are self-provisioned cloud systems.

5.1 Separation of Administrative Domains

The Virtual Organization Cluster Model specifies the high-level properties of systems that support the assignment of computational jobs to virtual clusters dedicated to single VOs. Central to this model is a fundamental division of responsibility between the administration of the physical computing resources and the virtual machine(s) implementing each VOC, permitting a division of labor between physical and virtual system administrators [69]. For clarity, the responsibilities of the hardware owners are said to belong to the Physical Administrative Domain (PAD), while responsibilities delegated to the VOC owners are part of the Virtual Administrative Domain (VAD) of the associated VOC. Although this model of domain separation builds upon the concept of physical and virtual domains described by the Violin project [136], a central distinction between the PAD and VAD is that both domains represent independent policy spaces, where decisions made by the administrators of each domain do not require cross-domain coordination. Each physical cluster has exactly one PAD and zero or more associated VADs. Figure 5.1 illustrates an example system designed using the VOC Model.

Figure 5.1: Implementations of Virtual Organization Clusters divide the grid into two administrative spaces: each grid site has Physical Administrative Domain (PAD) that includes the hardware and software services needed to execute virtual machines. Each VOC is an isolated Virtual Administrative Domain, into which scientific software applications, computational libraries, and supporting programs are installed. Different administrative domains in this model may have different administrative staff.

5.1.1 Physical Administrative Domain

VOCs are hosted on physical computing fabric made available by affiliated organizations, and perhaps third parties, over a standard grid computing platform such as the Open Science Grid [121]. Each of these physical sites is an isolated Physical Administrative Domain (PAD), managed independently from the VOCs it hosts. The PAD contains the physical computer hardware (see figure 5.1), which comprises the host computers themselves, the physical network interconnecting those hosts, local and distributed storage for virtual machine images, power distribution systems, cooling, and all other infrastructure required to construct a cluster from hardware. Also within this domain are the host operating systems and central physical-level management systems and servers. Fundamentally, the hardware cluster provides the hypervisors needed to host the VOC system images as guests.

An efficient physical cluster implementation requires some mechanism for creating multiple compute nodes from a single VO-submitted image file. One solution is to employ a hypervisor with the ability to spawn multiple virtual machine instances from a single image file in a read-only mode that does not persist VM run-time changes to the image file. Another solution would be to use a copy-on-write filesystem layer to allow the hypervisor to read from the centrally stored image, while writing to a temporary local file.

5.1.2 Virtual Administrative Domain

Each Virtual Administrative Domain (VAD) consists of a set of virtual machine images for a single Virtual Organization (VO). A VM image set contains one or two virtual machine images, depending upon the target physical system(s) on which the VOC system will execute. In the general case, two virtual machine images are required: one for the head node of the VOC, and one that will be used to spawn all the compute nodes of the VOC. For transparent implementations, only a compute node image with a compatible job scheduler interface is required.

VMs configured for use in VOCs may be accessed by the broader Grid in one of two ways. For participating implementations, the head node of the VOC functions as a gatekeeper between the VOC and the Grid. In transparent implementations, the VOC is constructed using a single image and needs to be configured with a scheduler interface compatible with the physical site. The physical fabric will provide the gatekeeper between the Grid and the VOC (figure 5.1), and jobs will

be matched to the individual VOC.

A major benefit of the VOC Model design is that few constraints are placed on the VM. The VO has great flexibility in selecting the operating system and software environment best suited to the requirements of its users. Of the few constraints that do exist, the primary ones are as follows:

- **Image Compatibility.** The VM image must be in a format usable by the Virtual Machine Monitor (VMM) or hypervisor software in use at the physical site(s) where the VOC will be executed.
- **Architecture Compatibility.** The operating system running in the VM must be compatible with the system architecture exposed by the VMM or hypervisor.
- **Dynamic Reconfigurability.** The guest system inside the VM must be able to have certain properties, such as its MAC address, IP address, and hostname, set at boot time.
- **Scheduler Compatibility.** When a transparent VOC is used with a shared scheduler provided by the physical site, the scheduler interface used by the VM must be compatible with the shared scheduler.

5.2 Terminology

To provide a framework for defining a Virtual Organization Cluster and evaluating its behavior, some preliminary terminology is necessary. These terms are presented in abstract grid computing contexts to avoid circular definitions.

Definition 5.1. A grid technology is *transparent* to an entity if the entity can utilize the technology through existing grid middleware services that are installed as part of a standard grid interconnection system, without the addition of any extra middleware. This definition extends that presented in Ensrow [53], which requires that access to the technology must be accomplished via services and not by direct connection to specific systems.

Transparency is a key contribution of VOC research compared to prior and related grid virtualization work. Most virtualized grid systems require the addition of specific middleware, and perhaps replacement of existing middleware, at both the execution endpoint and the submission endpoint in order to utilize virtualization capabilities through the acquisition of leases [7, 83, 94].

As a result, implementing these systems requires a substantial investment of resources for both cluster administrators and cluster users, along with substantial planning, coordination, and associated political will. Moreover, requiring users to access leased environments directly violates the original principle of transparency required in Ensrow [53]. By extending the concept of transparency in definition 5.1 to include middleware, systems that claim this property must be usable by an end-user equipped only with the standard grid access tools in existence and installed before the addition of the new technology. Ideally, the user would also be unaware of the existence of the new system. If this property can be satisfied, then administrative action is only required at the computational sites where the system is to be deployed, minimizing disruption to the existing grid infrastructure.

Definition 5.2. A job is *compatible* with a cluster if the job is executable using the hardware, operating system, and software libraries installed on the cluster. The cluster in this case may be physical or virtual.

For a given cluster system, the *compatible fraction* of jobs belonging to an entity is the ratio of compatible jobs belonging to that entity to the total number of jobs on the grid belonging to that entity. An entity is *compatible* with a cluster if the compatible fraction of jobs belonging to that entity is non-zero on the cluster.

Compatibility refers to the ability of a cluster system, whether physical or virtualized, to run a job with the available hardware (or virtual hardware) and installed software. Several different measures of compatibility may be considered when evaluating virtualized clusters, including the breadth of compatible VOs and the total proportion of compatible jobs across all VOs. Implementation of a new cluster technology might enable the cluster to support a larger number of different VOs; however, such an implementation might simultaneously reduce the previously compatible fraction of jobs for VOs already supported, reducing the total proportion of grid jobs that are compatible with the cluster. A trade-off may arise between these two measures for certain design decisions.

It is important to note that compatibility does not necessarily imply that a cluster is willing to run jobs from all compatible VOs. Local policies may restrict cluster usage to a specific subset of VOs, even though the cluster is compatible with a larger set of VOs. This distinction between capability and policy is formalized by the following definition:

Definition 5.3. An entity is *authorized* on a specific cluster if local cluster policies permit the entity to run jobs on the cluster. An *unauthorized* entity is denied use of a cluster system only by policy

and not by an insufficiency of mechanism.

The main purpose of definition 5.3 is to separate mechanism from policy when defining and evaluating technologies such as VOCs. It would be incorrect to treat a VO as incompatible if VO jobs were rejected by the cluster simply because local policy did not provide execution services to that VO. Technical compatibility, or the ability to run jobs within particular environments, is a separate issue from that of authorization to use a particular resource.

Definition 5.4. An entity is termed to be *participating* in a grid-enabled technology if the entity chooses to utilize the specific capabilities of the technology, including, but not limited to, the use of specific middleware or specific configuration settings. Entities that choose not to deploy the technology under discourse are termed *non-participating*.

In order to facilitate transparency and enable partial deployment of new grid-enabled technologies such as VOCs, it is necessary to accommodate the different schedules upon which different entities may choose to deploy the technology. Moreover, some entities may choose not to deploy the technology due to technical or policy constraints. Achieving interoperability of VOCs with existing grid systems requires that the grid remain able to support both participating and non-participating entities.

Definition 5.5. A grid technology is *conservative with respect to jobs* if jobs are neither created nor destroyed by the system. A grid technology is *conservative with respect to user jobs* if user jobs are neither created nor destroyed by the system.

Definition 5.5 implies that *all* jobs originate externally relative to a totally conservative grid technology. A grid technology that is conservative with respect to *user* jobs may create and destroy *pilot* jobs or *utility* jobs, but all *user* jobs originate outside the technology and are not destroyed by the technology.

5.3 Definition of a Virtual Organization Cluster

Utilizing the previous definitions, a Virtual Organization Cluster may be formally defined as a set of *homogeneous* computational resources that:

- Consists entirely of virtual machines that are scheduled and hosted by commodity physical resources on the grid;

- Autonomically changes size in response to changing workload demands;
- Is owned by, or dedicated to, exactly one Virtual Organization;
- Is transparent to end users;
- Is transparent to non-participating virtual organizations;
- Is conservative with respect to user jobs;
- Provides a Virtual Administrative Domain to participating VOs; and
- Optionally permits participating VOs to utilize a private overlay network to span resources across physical sites, receive jobs from a remote scheduler, and access private resources.

A VOC is effectively a set of *compute nodes* in the traditional cluster computing sense. The scheduling of jobs on these compute nodes may be performed by a shared local scheduler on the same physical site as the VOC, by a dedicated virtual head node on the same physical site as the VOC, or by a central scheduler accessible via an overlay network. The flexibility afforded by this definition permits a wide range of VOC deployments, from a transparent system provided on behalf of a Virtual Organization without any involvement of the VO itself, to a fully overlaid environment with a private scheduling and policy domain directly manageable by the VO.

Figure 5.2 depicts two VOCs dedicated to two separate VOs. In this case, VO₁ is non-participating and thus chooses not to deploy VOCs or any related middleware. Nevertheless, Site 2 provides VO₁ with a transparent VOC, so that all VO₁ user jobs on Site 2 run within virtual machines. Simultaneously, VO₂ chooses to utilize a VOC with a private overlay network. End user jobs from users affiliated with VO₂ are submitted directly to the private head node owned by VO₂, and VOC nodes are autonomically started on both sites in response to the size of the scheduler queue. VO₂ user jobs are then privately scheduled and routed by means of an overlay network.

It should be noted that the VOC Model is intentionally designed to be as technology-agnostic as possible, particularly with respect to the choice of virtualization system, network overlay, distributed filesystem, and grid middleware. Although the prototype implementation described in chapter 7 uses KVM for the virtual machine monitor, IPOP for the network overlay, PVFS for the distributed filesystem, and Globus for the purpose of connecting to the Open Science Grid, a cluster implemented according to this model should be deployable using any combination of services on any

Figure 5.2: Virtual Organization Clusters on a grid system. In this example, Site 2 transparently provides a VOC to VO₁, which does not deploy its own VOCs. VO₂ chooses to deploy VOCs and has a customized environment with private scheduling and resource control.

Table 5.1: Examples of virtualization systems, network overlays, and distributed filesystems that may be used to implement Virtual Organization Clusters. This table is for illustrative purposes and is not an exhaustive survey of available systems.

Virtualization Systems	Network Overlays	Distributed Filesystems
VMWare	IPOP	Lustre
Xen	Violin	PVFS
KVM	ViNe	GlusterFS

computational grid, provided the above constraints can be met. Table 5.1 lists a small set of possible services that could be used to construct a VOC.

5.4 Self-Provisioned Clouds

In traditional grid terms, Virtual Organization Clusters fill the role of Workload Management Systems from regular pilot infrastructures (section 4.3.2), in that they simplify the task of sending a job to a grid site for execution by providing a dedicated computational cluster for each Virtual Organization. End users simply submit jobs to the VOC as they would to any other cluster connected to the grid. Since there is assumed to be a single VOC for each VO, the user is not required to select from a potentially large number of grid sites to which a job may be submitted. Instead, all jobs associated with a particular VO are simply submitted to its corresponding VOC. As the user is not *required* to have any specialized software to send jobs to the VOC, the VOC head node must be connected to the grid for accessibility via standard middleware. It should be noted, however, that the provisions of the VOC Model do not *prohibit* VOC implementations from providing alternate mechanisms for user job submission, such as Web portals, provided that jobs also can be received via standard middleware. Furthermore, users could run local instances of the VM used to create the VOC and submit jobs directly to the VOC scheduler, bypassing the grid entirely.

Virtual Organization Clusters are cloud computing systems, since they are customized overlay clusters that do not require dedicated hardware. In the same way that virtual machines permit cloud services to be hosted by remote data centers (section 3.6.2), VOCs enable dedicated clusters to be hosted using the physical fabric provided by grid systems (chapter 2). Virtual Organizations are provided custom environments with administrative access, without the associated equipment and infrastructure costs that accompany physical cluster systems. By utilizing the grid as a commodity

source of computational infrastructure, smaller VOs that would otherwise be constrained by existing resources have access to the same level of environment customization available to larger VOs with dedicated infrastructure. In turn, these VOs can provide better service to end users by improving administrative responsiveness.

Unlike leasing systems that require users to make explicit resource reservations (section 4.1.1), Virtual Organization Clusters are *autonomically self-provisioned*. Implementations of VOCs must increase the size of the VOC to handle increased workloads automatically. Similarly, whenever workloads fall below some level specified by the management policy in a VOC implementation, the system must decrease the size of the VOC and return the excess resources to the grid. Regardless of the mechanism used to implement this dynamic sizing behavior (an example is presented in section 7.2), it must be autonomic and part of the VOC implementation.

Chapter 6

Simulator for Virtual Organization Clusters¹

Testing new paradigms for distributed computing at scale can be complex and expensive when a single entity owns the computing system. When the system consists of a large collection of resources owned by federated entities, such as the Open Science Grid (OSG) or Enabling Grids for E-sciencE (EGEE), reserving the entire grid for disruptive large-scale tests becomes impossible. Simulation testing provides a means by which large-scale tests may be run in a repeatable manner without the need to interfere with existing production applications [30]. When a research application requires substantial replacement of middleware, major changes to grid or site policy, or the addition of a common software stack to each grid site, simulation testing provides a means to evaluate the behavior of the proposed application prior to actual deployment.

One research area in which actual physical testing is challenging is that of *grid architectures*, where “architecture” refers to the combination of middleware, resources, and utilization methodology used to run end-user computational jobs across different grid sites. Since a major architectural change to a production grid, such as the system-wide addition of virtualization systems and virtual overlay clusters, could change the way the grid is used by its consumers, *a priori* assessment of the impacts of such a change is desirable from both research and management perspectives. Evaluation of the impacts of high-level architectural changes requires a high-level – or *aggregate* – assessment of a grid

¹Portions of this chapter have been published in [113]. Other portions have been submitted to the 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2010).

system as a single entity. Component parts of the system, such as the networking behavior and individual site scheduling algorithms, must be abstracted for reasons of simulation performance and result accessibility.

This chapter describes a novel discrete-event simulation system for assessing the aggregate impacts of changing the high-level architecture of an operational grid system. Specifically, the motivation for this simulation system is to describe the behavior of a production grid in which Virtual Organization Clusters (VOCs) [115] are utilized to execute all end-user jobs across all available hardware. Although VOCs are designed to be deployed non-disruptively with existing grid middleware, simultaneous grid-wide deployment would be impractical due to the inter-entity coordination that would be required to perform a large-scale test. Thus, the Simulator for Virtual Organization Clusters (SimVOC) has been developed to provide an evaluation environment in which actual grid trace data is utilized to instantiate dynamic simulated grids with replaceable architectures. By enabling repeatable experiments to be run from the same trace data, self-normalizing grid architecture experiments can be effected using a commodity workstation or laptop computer. SimVOC is available free of charge under an open-source license [3].

In the remainder of this chapter, the motivation for simulation is discussed in section 6.1. A short discussion of issues with available grid trace data for use as simulator input follows in section 6.2. Finally, the design of the SimVOC simulation system is presented in section 6.3. Results obtained from tests conducted using SimVOC are presented in chapter 8.

6.1 Motivation for Simulation

At a high level of abstraction, a typical production grid system has a “standard” architecture consisting of a collection of discrete computational resources to which end users may submit computational jobs. Access to these resources is controlled via policies specified by the entities – known as Virtual Organizations (VOs) – representing the resource providers. End users also organize themselves into VOs, typically according to research domains, and these consumer VOs obtain resource access by negotiating with the provider VOs. If an end user belongs to a consumer VO that is authorized to use resources on a site by the provider VO, then his or her jobs will be accepted for scheduling on the site. [66] However, acceptance of a job for scheduling at a site does not guarantee (or even predict) that the job will run correctly. Since each site may have its own software

stack, a job may arrive at a site on which the necessary software libraries are not present to enable proper execution. It is therefore the responsibility of the end user to select compatible sites for job submission.

Virtualization of grid systems is a possible mechanism for improving the usability of the grid by improving the job compatibility of execution environments [57]. Several virtual grid architectures are available and are based on the concept of explicit resource leases. These systems include Shirako [83], Cluster-On-Demand [32], and Virtual Workspaces via the Globus Nimbus middleware [94]. While these systems provide fine-grained resource control and provide each user with a customized execution environment, they each require additional middleware to be installed both at the execution and job submission endpoints.

Virtual Organization Clusters (VOCs) represent a different architecture for grid computing that is compatible with existing submission endpoint middleware. Instead of executing end-user jobs directly on the computational fabric, virtual cluster environments are instantiated for each VO, and user jobs are scheduled and executed within the virtual environments [115]. Since VOCs are constructed from virtual machines (VMs), and VM instances can be spawned from a single image, VOC environments are nominally homogeneous (and therefore software compatible) across grid sites. VOCs are autonomically managed and self-provisioned without explicit resource reservation requests from end users [117]; users instead simply submit jobs directly to a dedicated virtual grid site provided by the associated VO. Different technologies can be utilized to implement VOCs, since a VOC is simply an implementation of a system that conforms to the specifications presented in the VOC Model [115]. Thus, multiple mechanisms for VOC creation are possible. In the simple case, VOCs can be constructed by using pilot jobs [143] to lease resources from schedulers on existing grid sites, provided user-level virtualization systems (such as KVM [129]) are installed. Alternatively, VOCs may be created by autonomically managing leases from Shirako, Cluster-On-Demand, Nimbus, Amazon EC2 [11], or another leasing provider. Evaluating the viability of grid-wide VOCs is the primary motivation for developing SimVOC.

Unlike existing simulation systems (section 4.2), SimVOC is designed for the purpose of observing the aggregate behavior of an existing grid system using commodity scheduling algorithms. As SimVOC is able to execute simulations both with and without the addition of virtual machines and extra middleware, differences between existing grid systems and grid systems with deployed Virtual Organization Clusters may be evaluated. Key features of the system include:

- SimVOC is able to model dynamic grids that change over time. The internal grid representation within SimVOC is not limited to a static snapshot. Instead, sites, clusters, and CPU cores can join or leave the grid at any time during the simulation.
- SimVOC is designed to track the aggregate effects of adding dynamic overlay clusters to a production grid. Support for dynamic overlays is provided with the core simulator distribution. Per-overlay and per-entity resource allocation policies are supported.
- Unlike existing simulation systems, SimVOC contains explicit support for pilot jobs.
- SimVOC models machine capabilities, such as support for hosted virtual machines and file access resources, for each physical machine and virtual machine in the simulation.
- The SimVOC distribution includes models for both interval schedulers, such as Condor and the Load Sharing Facility (LSF), and generic scheduling systems such as the Portable Batch Scheduler (PBS).
- SimVOC provides a simple grid-level metascheduler that enables jobs to be dynamically routed to simulated grid sites based on site-reported resource availability.
- High-level simulations of data file transfers over Wide Area Networks are supported.
- SimVOC includes facilities for job and scheduler fault injection and system fault modeling.
- Input files to SimVOC are provided to the simulator core in an easy-to-use text-based input format that is designed for human readability. Conversion filters for Enabling Grids for E-science (EGEE) trace data sets (which may be obtained from the Grid Observatory [46]) are supplied with the SimVOC distribution.
- Output is produced by SimVOC using a flexible processing pipeline that allows results to be filtered, post-processed, aggregated, and directed into either plain text or SQLite database [4] files.

Following the Sulistio Taxonomy [152] (see section 4.2), SimVOC is a dynamic simulation system with continuous output capabilities. By providing a centralized random source with user seeding support, the serialized SimVOC kernel is able to provide both deterministic and non-deterministic execution, depending upon the requirements of each simulation experiment. While the SimVOC

kernel is fundamentally event-driven, a subset of events directly corresponds to an input trace. Grid models used within the simulator are hybrid entities capable of both creating and receiving events. SimVOC is presented to simulation consumers as a modular, object-oriented Python library.

6.2 Grid Trace Data

SimVOC utilizes a simple text-based input format for grid map, job trace, and observed Virtual Organization information. Each input file is time-based, and inputs are read at simulation-time whenever the simulator clock reaches the time stamp of the next input block. Thus, all inputs to the simulation system are completely dynamic. In particular, the grid map may change over time, with cluster sizes increasing and decreasing, and grid sites appearing and disappearing. Also, new VOs may appear later in the simulation, representing the addition of newly formed VOs to actual grid systems.

Although the input format to the simulator is simple enough that experimental grids could be designed by hand, it is expected that many simulations will utilize trace data mechanically converted from an existing production grid system. A mechanism has been developed to utilize trace data from the Grid Observatory [46], which provides monitoring and topological information about the Enabling Grids for E-science (EGEE) production grid system. The input conversion mechanism included with SimVOC is also easily extended to other input formats.

6.2.1 Grid Map Information

Information about the presence and size of grid sites is obtained from the Grid Observatory using gLite [52] data collected by the GRIF/LAL (Groupe Grilles du LAL, part of the Grille de Recherche d’Ile de France) organization. These files are distributed as compressed tape archives (tar files) and are formatted in a modified LDAP Data Interchange Format (LDIF). Each LDIF entry in each base-level file provides information about entities observed on the grid, including grid sites and their associated processor (CPU) counts.

Since multiple grid sites may share the same hardware to fill the resources properly, the same CPUs are reported multiple times within the data set. Without any more detailed information about the underlying physical systems, it is necessary to apply some CPU count reduction heuristics when processing the input data, in order to reduce the amount of over-count. The first of these

heuristics detects multiple Computing Elements (CEs, or grid sites) sharing the same underlying hardware by detecting duplicate sites with the same domain name and the same CPU count. A second heuristic is then applied to the processed data, in which all sites (which now have distinct CPU counts) on the same domain are presumed to be sharing the same hardware. The largest CPU count from the set of such sites is selected as the actual cluster size for the underlying physical system. Although this heuristic is trivially incorrect, since it is quite possible to have two different clusters from the same domain attached to the grid, the additional over-count reduction has been found to be beneficial (see section 8.2). Without these heuristics, the grid appears to be largely under-utilized, which is not the case [1].

6.2.2 Job Traces

Job trace data sets are obtained from the Grid Observatory as observed by the EGEE Real-Time Monitoring (RTM) system. These traces are provided in a compressed tar archive containing tab-delimited information fields. As the RTM system records jobs from the time of submission, the data set must be filtered to select only jobs registered as having completed, since *a priori* length information is needed for simulation purposes. Jobs are also filtered to select those with valid (nonzero) submission timestamps, which further reduces the number of jobs from the trace that are actually usable in the simulator. It should be noted that the contents of the (unfiltered) job traces do not agree with the published grid utilization information from available accounting systems [1], for unknown reasons. The RTM trace sets always contain fewer jobs than are reported by the accounting systems, and only a fraction of jobs in the RTM traces have valid timestamps.

6.3 Simulator Design

SimVOC is fundamentally designed to model dynamic grids that change over time, with dynamic overlay clusters (Virtual Organization Clusters) allocated using pilot jobs to obtain simulated physical resources. Abstract scheduling facilities are provided, which model common cluster scheduling systems at a high level. The simulation system is divided into several top-level components (figure 6.1): the simulator core, input format handlers, simulation drivers, and utilities. The core contains the simulation kernel and objects that work with the kernel to implement simulated grid computing technologies including schedulers, file caches, compute nodes, and virtualization sys-

Figure 6.1: High-level components of SimVOC

tems. In addition, the core also provides the mechanisms needed to monitor the simulations and to collect results. To provide input data for the simulations, the input format handlers parse files in various formats to extract information about the architecture of grid systems to be modeled, as well as job-specific data such as simulation time and run length. Raw input files are preprocessed into intermediate files using utility programs; driver programs are then used to join these intermediate files with the simulator core to effect simulation experiments.

6.3.1 Simulation Kernel

SimVOC utilizes a generic, reusable discrete-event simulation kernel as the central mechanism for dispatching events and receiving results. This kernel is dependent only upon other classes and functions within the same module and modules available in the standard Python library. As such, the simulation kernel can be reused easily in other simulation projects implemented in the

Python language. Central to the kernel is a main event loop, which runs until the kernel event queue is empty. At execution time, each event invokes a function call, the return value of which (if any) is directed to the simulator output processing pipeline. Periodic trace information is also collected during event execution.

Simulator events are flexible enough to adapt to a wide variety of requirements. However, in practice, three types of events are used frequently: *regular* events, *external* events, and *periodic* events. Regular and external events have a flag set in the simulator kernel, which causes the total number of these types of events to be counted separately. In contrast, periodic events do not set this flag. As the name suggests, periodic events are designed to support components that need to re-schedule events on a regular basis; by detecting the absence of regular events in the event queue, these periodic processes determine when the end of simulation input has been reached. To prevent infinite event loops, it is necessary for periodic client code to stop re-scheduling periodic events once the simulator has reached an empty steady state. Among non-periodic events, the only distinction between a regular event and an external event is that the current time and a reference to the simulator object are passed to the function invoked by a regular event, while an external event invokes an external function that is not necessarily simulator-aware.

6.3.2 Job Model

All jobs in the SimVOC system are uniquely identifiable objects containing methods for starting, stopping, and killing themselves. At start time, regular jobs schedule their own completions using *a priori* length information encoded in the job object. In addition, jobs may experience random errors, which are determined at start time using the centralized simulator random source and a specified error probability. Full job records are recorded in the simulation output at completion time. Completion of a job, with error status, may be effected at any time prior to scheduled completion by invoking the job kill method, which simulates a system-induced abnormal termination.

In contrast to regular jobs, *pilot jobs* [143] are a special subclass of jobs that do not have *a priori* length information. Instead, these jobs invoke a callback function at start time. The callback function initiates a simulated process representing a workload transmitted to the remote system executing the job. Whenever the process is complete, it notifies the pilot job via a termination method, which schedules job completion. Random pilot job initiation errors may be injected at start time, inhibiting execution of the callback function. Execution of the kill method on a pilot job

results in transmission of a kill signal to the external process via the callback function, simulating the procedure by which processes are killed on Unix systems. Pilot job simulation is a key distinguishing feature of SimVOC.

6.3.3 Resource Representations

Within the simulation system, machines comprise the lowest level of granularity. For simplicity, each machine is assumed to contain a single CPU core, such that there is always a one-to-one correspondence between machines and cores. Each machine can be claimed by a single scheduler to run at most one job; however, multiple schedulers may target a single machine. Machines can be shut down, at which point they no longer permit job execution. Furthermore, machines are extensible through the addition of *capabilities*, which are named references to objects that provide specific functionality, such as simulated Virtual Machine Monitors (or hypervisors). At job execution time, machines may dynamically adjust the length of the job to simulate the addition of execution overhead introduced by various software stacks. This property is used by the Virtual Machine subclass to simulate virtualization overheads using adjustable overhead bounds, which is a novel capability of the system.

Clusters consist of collections of machines, along with their associated schedulers. Each cluster may be resized from the top down, using methods to create or remove machines in a dynamic simulation. Virtual Organization Clusters are implemented by subclassing the base cluster definition, using pilot jobs to lease regular machines for the purpose of spawning virtual machines. In the process of instantiating virtual machines, pilot jobs must query the physical machine for a virtualization capability, request the virtual machine image from an associated file cache (obtained from a scheduler resource broker, next section), and then instruct the virtual machine image to boot. Once booted, the new virtual machine image joins the parent VOC and its scheduler pool. At any step within the boot process, faults may be simulated by making resources or host machine capabilities unavailable, resulting in abnormal pilot job termination. Although the VOC component implements the pilot job mechanism, a separate component – called a watchdog [117] – implements the autonomic VOC management policy.

Data files, including virtual machine images, are simulated by specifying the file size in bytes. Abstract network components simulate transfers of these files from stores (servers) to caches (clients) by computing the transfer time as a function of file size and link speed. At higher levels within

the simulation, site objects collect resources to simulate grid compute elements, which primarily provide a named interface to a single scheduler on a cluster. A grid object forms the top level of the simulation, which collects site objects, receiving and routing jobs from the job trace input mechanism.

6.3.4 Scheduling Models

Two primary scheduling models are implemented within the simulator: greedy first-come, first-served scheduling, and interval scheduling with machine property matching. The former type of scheduling is non-periodic except when no free scheduler slots are available, at which time the scheduler periodically re-checks its targets in case a machine becomes available. Since multiple schedulers may target the same machines, there is no guarantee that a machine will be available to any one scheduler at any given time. The second type of scheduling is implemented as a purely periodic scheduler, which only performs property-based matching of jobs to machines at regular scheduling intervals. This scheduler is designed to simulate the Condor [155] High Throughput Computing system, with simplified ClassAds.

Another type of scheduler, known as a metascheduler, is available at the grid level. This scheduler does not start jobs directly on machines. Instead, the metascheduler selects a grid site to which the job should be submitted, based solely upon a ranking algorithm designed to favor sites with at least one free core. When grid jobs are submitted to the metascheduler, they are immediately submitted to the best-ranked site for execution. Should a job fail to start on a particular site within a given period of time, the metascheduler will cancel the job submission and re-submit the job to a different site. As jobs are sent to sites, the metascheduler utilizes a circular queue to load-balance submissions among all sites on the grid.

In addition to providing scheduling services, the schedulers in SimVOC also provide resource brokering, local statistics collection, and fault simulation support. Resource brokering permits a scheduler to provide a named interface to an object providing a specific resource, such as a file cache, that is to be made available to running jobs. Statistics collection routines are implemented as a set of self-aggregations that maintain continuous counts of jobs waiting, running, completed, and in an error condition, both scheduler-wide and for all Virtual Organizations represented by all jobs observed by the scheduler. When a target machine is abruptly terminated, as is the case when a pilot job running a virtual machine is killed, scheduler fault behavior is reproduced by allowing

the scheduler to attempt to claim machines that are no longer available. When such faults occur, the schedulers must recover from the fault by selecting a different target.

6.3.5 Input and Output

Inputs to SimVOC are provided in a simple, extensible text input format that is designed for both human accessibility and simple machine generation. Three categories of input are provided: grid map data, job traces, and Virtual Organization records. Grid maps specify the sizes of clusters at any given time in the simulation, as well as the names of sites and schedulers that utilize the clusters. Job traces provide submission time, job name, VO affiliation, length, and (optionally) observed queuing delay information for each job in the trace. Virtual Organization records provide the names of each VO observed in the job trace data, along with optional policy configurations used by the watchdog component in VOC simulations. All three input types are time-oriented, and time-specific blocks of input are read only when the simulator clock reaches the time specified in the block.

SimVOC utilizes a pipeline model for simulator output processing. Pipeline components include filters, duplicate data reduction mechanisms, data aggregation components, and file output handlers. The primary file format utilized by the initial release of SimVOC is the SQLite database format [4], version 3. External scripts, supplied with the SimVOC distribution, are used to query the database and extract data for analysis purposes. In addition, external programs may read the SQLite databases directly to generate plots or perform statistical analysis.

Chapter 7

Prototype Implementation

In order to evaluate the actual components required to implement Virtual Organization Clusters, and to measure the overheads introduced by virtualization, a physical cluster was constructed for use as a test bed. Several prototype installations were performed using different software stacks, so that the cluster could be adapted to changing research needs. In the initial configuration, the prototype cluster utilized a minimal installation of Slackware Linux 12 on the hardware, placing the end-user software stack in a set of CentOS 5.1 virtual machines. When it was later determined that a 64-bit host operating system was required for memory block allocation reasons, CentOS 5.2 was installed on the hardware. Over time, both the host and guest operating systems were upgraded to CentOS 5.3.

Several system management challenges were encountered during the test bed installations and maintenance operations. Due to the heterogeneity of the compute nodes and storage nodes within the same cluster, existing batch administration tools were difficult to use. Regular cluster services, such as the Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS) servers, required duplication of configuration information, resulting in errors whenever configurations required adjustment. To address these problems, a cluster configuration tool called Stoker was developed to centralize all configuration settings in a Lightweight Directory Access Protocol (LDAP) database. For management of ephemeral virtual machine instances, a second tool – named Pulley – was envisioned to deliver configuration settings whenever a system became available. Due to the development of commercial applications for performing similar management tasks in virtualized data centers [170], the Pulley project was discontinued.

Both the prototype implementation and management systems are discussed in the remainder of this chapter. Section 7.1 describes the initial cluster configuration with Slackware Linux 12 hosts and CentOS guests. Section 7.2 discusses the process of refining the cluster services to support dynamically provisioned Virtual Organization Clusters. Finally, section 7.3 presents system management issues, Stoker, and Pulley.

7.1 Cluster Setup¹

An initial cluster implementation was performed to test the Virtual Organization Cluster Model. This section presents in detail the procedure (figure 7.1) that was followed to set up the physical cluster, configure the physical fabric to support virtualization, and to construct the VOC itself. The physical test cluster used the Kernel-based Virtual Machine (KVM) hypervisor, which was installed on physical hosts running Slackware Linux 12. A Virtual Organization Cluster was constructed around a single virtual machine image into which CentOS 5.1 had been installed. In this particular implementation, the head node for the VOC was provided as part of the physical fabric, even though it was actually implemented inside a virtual machine. This head node was connected to the Open Science Grid Integration Testbed.

7.1.1 Physical Cluster Construction

The hardware cluster for the test installation consisted of sixteen nodes: fifteen Dell PowerEdge 860 1U rackmount systems, and one Dell PowerEdge 2970 2U rackmount server. One PowerEdge 860 system was employed to host the VOC head node, while the other fourteen were each prepared to host two VOC virtual compute nodes. Each PowerEdge 860 machine used in this test was configured with a 2.66 GHz dual-core Intel Xeon CPU, 4 GiB of RAM, and an 80 GB hard disk drive. The 2U PowerEdge 2970 server was employed to host installation images, user home directories, network services, and a shared VM image store exported via a Network File System server. Prior benchmarks and considerable network test results [116] were obtained using a previous CentOS 5.0 installation on the same hardware.

To provide hypervisor services, the Kernel-based Virtual Machine (KVM) was installed on each compute node and on the physical head node. KVM was chosen primarily due to its

¹The contents of this section have been published in [115] and [55].

Figure 7.1: Process of constructing the original prototype physical cluster and hosted Virtual Organization Cluster. The VOC represented an independent administrative domain and was constructed separately.

compatibility with the most recent kernel release at cluster construction time, as the most recent drivers were needed for optimal performance of certain hardware components.

Network access was provided to each virtual machine by means of bridging the physical Ethernet card in each physical compute node both to the physical node itself and to each guest machine (two guests per host). Thus, three logical devices shared each physical device. MAC addresses were assigned to each VM instance by KVM, using a custom script to generate the MAC addresses deterministically based on the host machine. On the physical head node, two separate bridges were employed: one to the cluster’s private LAN, the other to the University network and public Internet. Network Address Translation and iptables firewalls were implemented on both the physical head node and utility system, allowing them to serve as edge routers for the entire private LAN. Since each VM obtained an IP address from the DHCP server, and each IP address was part of the same subnetwork without regard to physical or virtual host status, each VM instance had both Internet access and local connectivity to other VMs in the VOC.

In the test cluster, a common VOC head node was provided as part of the PAD. For administrative simplicity, this CentOS 5.1 node was implemented as a virtual machine that was bridged

Figure 7.2: Initial Test Cluster Setup. Each physical compute node hosted a VOC node as a virtual machine, using the KVM hypervisor. The single VM image file resided on a network filesystem that was provided by other hardware connected to the cluster. Job scheduling in the transparent VOC was performed using Condor.

to the public Internet. To supply job scheduling, Condor 7.0.0 was installed on the shared VOC head node as well as on all compute nodes. Open Science Grid Integration Testbed membership was achieved by installing the OSG Virtual Data Toolkit (VDT) and connecting to OSG by configuring an OSG compute element. The compute element that ran Globus GRAM was set up as a shared head node for both the physical and virtual compute nodes. Differentiation between the PAD and VAD was done through the attributes advertised by each compute node's Condor *startd*. This setup, shown in figure 7.2, provided a transparent VOC to the Engage VO.

7.1.2 Virtual Cluster Construction

Constructing the Virtual Organization Cluster for the test system was a straightforward task, since only 1 Virtual Machine image was required to implement the whole VOC. CentOS 5.1 was installed into a VM image, then Condor was installed and configured to run a *startd* process to enable jobs to be scheduled. The VM image was configured for DHCP networking, and the

primary assumptions made about the underlying Physical Administrative Domain were that jobs would arrive via the Condor scheduler and that the KVM hypervisor would be used to execute the VMs.

As a result of the hardware emulated by KVM, implicit low-level requirements were imposed upon the VOC system. In practice, these requirements were not substantial, since the Linux system used in the VOC was generic enough to support the emulated hardware. However, a different choice of guest operating system might have required additional configuration steps for the VO administrator. In particular, KVM could execute only 32-bit, x86-compatible operating systems.

7.2 Dynamic Provisioning of Virtual Organization Clusters²

Virtual Organization Clusters are configured and started on the physical compute fabric by middleware installed in the Physical Administrative Domain. Such middleware can either receive a pre-configured virtual machine image (or pair of images) or provision a Virtual Organization Cluster on the fly using an approach such as VMPlants [99] or installation of nodes via virtual disk caches [120]. Middleware for creating VOCs can exist directly on the physical system, or it can be provided by another (perhaps third-party) system. To provide participating VOs with full administrative access, VM images also can be created manually and uploaded to the physical fabric with a grid data transfer mechanism.

Once the VM image is provided by the VO to the physical fabric provider, instances of the image can be started to form virtual compute nodes in the VOC. Since only one VM image is used to spawn many virtual compute nodes in efficient implementations, the image must be read-only. Run-time changes made to the image are stored in RAM or in temporary files on each physical compute node and are thus lost whenever the virtual compute node is stopped. Since changes to the image are non-persistent, data corruption is not an issue, and VM instances started in this way can be safely terminated without regard to the machine state. As an example, VM instances started with the KVM hypervisor are abstracted on the host system as standard Linux processes. These processes can be safely stopped (e.g. using the SIGKILL signal) instantly, eliminating the time required for proper operating system shutdown in the guest. Since there is no requirement to perform an orderly shutdown, no special termination procedure needs to be added to a cluster

²The contents of this section have been published in [117].

Figure 7.3: A watchdog process monitored the size of the job queue on the shared gatekeeper node (1), booting virtual machines to expand the size of the Virtual Organization Cluster as necessary (2-4). Jobs from the Condor queue on the shared gatekeeper were executed on the VOC nodes as they became available (5). This process was repeated until either the VOC was large enough to run all the jobs (6-11) or no more physical hosts were available to run VOC nodes.

process scheduler to remove a VM from execution on a physical processor.

Once mechanisms are in place to reserve physical resources and start VMs, entire virtual clusters can be started and stopped by the physical system. VOCs thus can be scheduled on the hardware following a cluster model: each VOC would simply be a job to be executed by the physical cluster system. Once a VOC is running, jobs arriving for that VOC can be dispatched to the VOC. The size of each VOC could be dynamically expanded or reduced according to job requirements and physical scheduling policy. Multiple VOCs could share the same hardware using mechanisms similar to sharing hardware among different jobs on a regular physical-level cluster.

In order to provision Virtual Organization Clusters dynamically, grid-enabled middleware

was developed. This middleware enables physical systems to host VOCs that are connected to the Open Science Grid. Grid jobs arrive via Globus [64, 66] and are deposited in a job queue on a gatekeeper system, where the Condor job scheduler [155] runs. A watchdog process (illustrated in figure 7.3) periodically samples the Condor queue and starts virtual machines that belong to the VO with which each job is associated. As the number of jobs in the queue for a particular VO increases, the watchdog will attempt to start additional VMs to increase the size of the respective VOC, subject to the limitations imposed by the hardware and by site policy. When the watchdog observes fewer jobs in the queue than there are executing VOC nodes for a particular VO, VMs belonging to the VOC are terminated. This process ensures that provisioning of physical resources dynamically adapts to the job loads of the VOs supported by the grid site, without any manual intervention or need for external middleware.

7.2.1 Job Tagging

Upon the arrival of a grid job, a corresponding Condor job is created by Globus. The dynamic VOC system modifies the Condor ClassAd to add a requirement that the target system match the VO with which the grid job is associated. Within the ClassAd, this additional requirement takes a simple name equals value form, in which the name of the VO is prefixed to form the name, while the Boolean condition of truth is used as the value.

As an example, the Requirements field for a job associated with the Engage VO, targeting a 32-bit Linux system, might have the form:

```
Requirements = (Arch == "INTEL") && (OpSys == "LINUX") && (VO_ENGAGE == TRUE)
```

One requirement imposed upon each VOC compute node VM is that its Condor interface expose a corresponding ClassAd field to match the VO. Thus, a VOC node that is owned by the Engage VO should include the following field in its ClassAd:

```
VO_ENGAGE = TRUE
```

With this mechanism, the Condor scheduler will handle the matching process of a grid job to a corresponding VM belonging to an associated VO. The dynamic provisioning middleware, however, is still responsible for starting the VM. Only after a VM has booted and joined the pool does Condor proceed to run the job.

7.2.2 VOC Management

Management of the VM instances comprising the individual VOCs is performed entirely by the watchdog, using the Condor queue as its data source. As the watchdog observes an increasing number of jobs associated with a particular VO, it attempts to start additional VMs in order to increase the size of the corresponding VOC. The maximum number of available slots in which to start VMs is known to the watchdog, and it will not exceed the number of slots pre-specified by the system administrator. Furthermore, the total number of slots available across the physical fabric at a single site may be further subdivided among the VOs supported by that site. Thus, administrators can enforce flexible site policies to balance resource usage among different VOs.

Whenever the watchdog observes that a single VOC has more running VMs than there are jobs belonging to the corresponding VO in the queue, the watchdog will reduce the size of the VOC by terminating VMs that are not claimed by Condor. Since the VOC model [115] specifies the use of a single read-only disk image to spawn all VMs in a VOC, termination of VMs is accomplished instantly by means of killing the virtual machine monitor process. Once a VM has been terminated, its slot is re-claimed by the watchdog and added to the pool of unclaimed physical Condor slots.

7.3 Management of Systems³

In order to instantiate Virtual Organization Clusters, underlying physical hardware is needed to run virtualization software and host the VOC nodes. While this hardware could be owned by the Virtual Organizations themselves, a cloud computing model would favor the use of specialized hosting providers that make physical fabric resources available on a grid system. Each hosting provider would thus be responsible for the acquisition and maintenance of all the hardware necessary to construct physical clusters, including the computing systems, local storage, network systems, power conditioning and distribution, and cooling infrastructure. Physical site administrators would be responsible for local site policies and maintenance tasks, but their administrative responsibility would extend only to the physical fabric at the specific site. Thus, each individual physical site would be a separate administrative domain, known as a Physical Administrative Domain (or PAD).

A simplifying assumption made about each physical site is that physical resources are devoted exclusively to hosting VOCs: no scientific jobs are executed directly on the hardware. As a

³The contents of this section have been published in [114].

result, the software set required on each compute node is minimal and consists of the host operating system, virtualization applications, and any needed driver software to support installed hardware. Standard networking technologies, such as the Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS), are employed in a centralized manner to enable connectivity between systems, while simultaneously maintaining the scalability of the physical cluster as a whole. By centralizing host-specific settings, such as the IP address and host name of each compute node, management inside the PAD is simplified.

One issue that does arise when using standard networking services is the replication of identical information across different services. For example, if fixed IP and host name assignments are to be made using DHCP, then the same mapping of host names to IP addresses must be entered into the DNS records for resolution of host information to work correctly within the system. A solution to avoiding this replication is to centralize the duplicate information, using an external database such as a Lightweight Directory Access Protocol (LDAP) server. Services utilizing the information in the database are then configured to use the external database directly, provided the services have the required integration capability. Otherwise, the services are adapted, through the use of middleware layers, to update their local configurations from the centralized database upon request.

7.3.1 Stoker

Stoker is a scalable remote management tool, whose overall architecture is shown in figure 7.4. Stoker differs from prior management tools such as Tentakel [156] and the Cluster Command and Control Suite [58] in that it can obtain system information directly from a centralized database with system grouping capabilities, thereby avoiding replication of host information in a configuration file. Stoker also has an extensible, modular design with three major components: *warehouses*, *core*, and *actors*. These components handle retrieving configuration information for a node or group of nodes, spawning and joining actor threads, and performing some type of action on the target nodes.

Stoker Warehouses

Stoker can use multiple data sources or *warehouses* to gather data about target nodes. Central to the warehouse data retrieval task is the concept of a *resolver*. Resolvers take as input a logical node or group name and return a data structure containing addressing information potentially

Figure 7.4: Overview of Stoker. Multiple configuration warehouses may be used to provide target machine information, including LDAP databases, MySQL databases, and text configuration files. Stoker uses SSH to run commands on machines described by these warehouses.

including, but not limited to, hostnames, IP addresses, and MAC addresses. Stoker’s grouping feature allows machines to be organized into arbitrary groups and subgroups for the convenience of the administrators.

A separate resolver is required for each type of data warehouse. Resolvers currently exist for LDAP databases, MySQL databases, and plain text files. An administrator wishing to implement a new resolver needs only to write a small Python class that retrieves the information from the warehouse and inserts it into a simple data structure defined by Stoker.

Stoker Core

A design goal of the Stoker *core* was to encapsulate the complexity inherent in a multi-threaded application, thus allowing a system administrator to extend or create, with minimal effort, new warehouses and actors to meet his or her needs. The primary function of the core is to invoke resolvers and spawn an actor thread for each target. The core then manages each thread, collecting any output from the actor, and generates an activity report for the user.

Since the core does not have *a priori* knowledge of whether or not a user-supplied target is a single node or a group of nodes, it must be flexible enough to handle a situation in which the

user inadvertently specifies a target multiple times. For example, if a user toggles a configuration option on two groups of machines with overlapping membership, the core must ensure that the configuration option is toggled exactly once on each system.

Stoker Actors

Stoker *actors* are analogous to Stoker warehouses in the sense that they are (potentially) simple scripts that perform a simple task at the direction of the core. Actors execute in their own threads and have well-defined data structures that specify all known information about a single target. The actor's task is to apply its argument, also provided by the core, to its target. Actors supplied by the core Stoker distribution perform such tasks as remote command invocation via SSH, ping, Wake-on-LAN, and local (relative to the user's machine) command invocation. New actors are easily created by the system administrator to perform new functions.

7.3.2 DHCP and DNS Middleware

When managing physical clusters, it is convenient to have a single repository for all node-specific information. Unfortunately, many software packages do not include support for a centralized configuration repository, requiring instead a product-specific configuration file. To eliminate needless data duplication and possible inconsistencies between services that share data, database-aware middleware was developed (illustrated in figure 7.5) using a Lightweight Directory Access Protocol (LDAP) server as the data source. As a proof of concept, two network services were adapted to use this middleware: ISC *dhcpcd*, a Dynamic Host Configuration Protocol (DHCP) server; and *dnsmasq*, a Domain Name System (DNS) server.

The middleware is implemented as a Python script that accesses the LDAP database and writes a configuration file for the given software package. This script is then integrated with the service initialization scripts so that every time the software package is started or restarted, the configuration file will be regenerated from the LDAP database, ensuring that any changes will be propagated throughout the system.

Figure 7.5: Automatic integration of Stoker with DHCP and DNS services when an LDAP database is available as a warehouse

7.3.3 Management of Virtual Machines⁴

Unlike the physical fabric systems, which are comprised of single, isolated sites, the Virtual Organization Clusters potentially execute on arbitrary sites, or even across physical sites. As a result, it is not practical to assume that virtual machines will necessarily always be directly reachable, since physical sites may be constructed using private networks with Network Address Translation (NAT) used on routers connecting the private networks to the commodity Internet. Without complex arrangements for forwarding connections to individual VOC nodes, the use of client-initiated transports like SSH is not feasible. As a result, configuration tools that operate on a “push” model, such as Stoker, are not suited for managing these virtual clusters.

An alternate system that can traverse NAT boundaries automatically is in the initial design stages. This middleware utility, called Pulley, will use a central database and application server to publish virtual cluster node configurations on the Internet. Client VMs will periodically poll the application server using standard HTTP Web service mechanisms. Since the polling requests will originate from the VMs themselves, they will be able to traverse NAT and firewall boundaries at the physical sites, without requiring any additional software or services to be made available on the

⁴This section is presented as originally published in [114]. Since that time, the Pulley project has been discontinued.

Figure 7.6: Overview of Pulley, which consists of a configuration database, a configuration server, and clients located on each VOC node. Each client periodically polls the Pulley server, making a connection through the NAT router and across the Internet. The Pulley server responds with configuration data obtained from the Pulley database, enabling the Pulley client to effect configuration changes on the VOC node.

physical fabric. This design is illustrated in figure 7.6.

Since Pulley is in a formative stage of design, the exact mechanism of its management functionality is not yet known. Two models of operation are planned for investigation. One of these possible architectures would be to adapt Stoker actors to draw operational information directly from the configuration database via middleware, instead of relying on human system administrators to provide command input. The second potential architecture would be to implement a policy-based mechanism such as Cfengine, adapting the agent-based system to operate across Grid sites via middleware protocols. It is even possible that a hybrid of both architectures will be used in the final system. In the meantime, simple management scripts that obtain configuration information from an HTTP server are in use to install extra packages, set package repository priorities, and make firewall changes.

Chapter 8

Simulation Test Results¹

Although the prototype system was available for testing, its size (reaching a maximum of 16 compute nodes with a total of 32 CPU cores) limited investigation of the aggregate performance of production grid systems with VOCs widely deployed. Since an actual physical deployment of an experimental architecture across an entire production grid was impractical, a simulation system was developed, called the Simulator for Virtual Organization Clusters (SimVOC) [3]. Following the taxonomy provided by Sulistio et al. [152], SimVOC was designed to be a dynamic, trace-driven, discrete-event simulation system with continuous output, hybrid grid model, and a serialized (single-threaded) kernel. As utilized in the experiments described here, SimVOC was operated via Python driver applications in a deterministic execution mode. For simplicity, the simulation system did not model cross-site authentication and authorization. Instead, it was assumed that each physical grid site permitted jobs from users affiliated with any Virtual Organization observed in the trace inputs. For the purpose of simulating VOCs, a synthetic VO named “_pilot_” was created, and it was assumed that every grid site would accept virtualization jobs from the “_pilot_” VO.

A number of simulation experiments were performed to evaluate the viability of VOCs on the production Enabling Grids for E-sciencE (EGEE) system. All simulation experiments were effected using a commodity Dell Latitude E6400 laptop computer with an Intel Core 2 Duo P8400 CPU at 2.26 GHz clock speed and a reported 4523.39 BogoMIPS per core. A total of 4 GiB physical system memory was installed, and virtual memory swapping behavior was not observed during any

¹Portions of this chapter have been published in [113]. Other portions have been submitted to the 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2010).

simulation run. The software environment of the system consisted of a 64-bit installation of Arch Linux, with Python 2.6.2 utilized as the interpreter for the SimVOC code.

In this chapter, the results of early small-scale simulation tests are presented in section 8.1. Processing steps required to handle input trace data are discussed in section 8.2, followed by a description of the control simulation used to evaluate the raw data in section 8.3. Simulation test results are presented for standard grid architectures in section 8.4 and the VOC architecture in section 8.5. A discussion of the effects of virtualization overhead concludes the chapter in section 8.6.

8.1 Preliminary Simulations

Early discrete-event simulation results were obtained through the use of a preliminary version of the discrete event simulator discussed in chapter 6. A synthetic workload was engineered, along with a simple greedy scheduling algorithm, to test the process of sharing a single physical site among several Virtual Organization Clusters. This workload contained 8 jobs of varying length, divided into three jobs of 200 seconds in length for the Engage VO, three jobs of 400 seconds in length for the Engage VO, and two jobs of 600 seconds in length for the NanoHub VO.

As depicted in figure 8.1, both VOCs could be started and expanded to a sufficient size so as to provide execution capacity for all eight jobs in parallel, since the physical site had capacity for 16 virtual machines. When jobs completed, the sizes of each VOC could be reduced at the next watchdog interval, and eventually all VOC nodes were removed when all jobs finished. The total execution time for the makespan was 1200 seconds.

When the number of virtual machine slots available to run the VOCs was reduced to four (figure 8.2), the total time for completion of the makespan was increased to 2100 seconds. Both the Engage and NanoHub VOCs were able to claim 2 slots at the first watchdog cycle, permitting the NanoHub VOC to execute both NanoHub jobs in parallel as soon as the corresponding VOC nodes booted. The Engage VOC was able to execute two jobs in parallel until the NanoHub VOC completely vacated the physical cluster, after which the Engage VOC started a third node. Parallelism was increased only briefly, as completion of the third job led to the removal the third node after only 300 seconds. As the remaining two Engage jobs completed, the Engage VOC nodes terminated, leading to a stair-step pattern in the graph.

Figure 8.1: Sharing 16 virtual machine slots among two distinct Virtual Organization Clusters

Figure 8.2: Sharing 4 virtual machine slots among two distinct Virtual Organization Clusters

8.2 Input Data Processing

Before simulation of a full production grid, input data sets were obtained from the Grid Observatory [46] and converted into the text input format supported by SimVOC. The traces consisted of a dynamic map of the Enabling Grids for E-sciencE (EGEE) grid; the set of jobs, including submission times, lengths, and VO relationships, observed on EGEE by the gLite middleware [52] from 00:00 UTC on December 8, 2008, to 23:59 UTC on May 10, 2009; and a dynamic list of Virtual Organizations found on the grid over the same time period. Prior to application of the CPU over-count reduction heuristics described in section 6.2, an initial CPU count of 314,547 was observed in the map data. Once the heuristics were applied, the initial CPU count was reduced to 107,396. A total of 258,097 jobs from the associated job trace were found to have been registered as complete with valid time stamp data, as shown in table 8.1. This count represented a fraction of the approximately 4 million jobs present in the raw trace data; however, both valid time stamps and *a priori* job length information were required for simulation purposes.

An additional data issue resulted from the timing of job arrival and Virtual Organization registration, relative to the grid map data. Since the job and VO data were derived from the job traces, while the map data were extracted from a different data set, some discrepancies were found in the timing. In particular, jobs could not always be matched to target sites or to affiliated VOs, resulting in jobs receiving a zero execution length and an error status, since the simulator was unable to find the resources on which the jobs were supposed to run.

8.3 Control Simulation

To check the non-scheduler portion of the simulated grid model, a control simulation was effected, in which the actual job start and finish times from the EGEE job input trace were used to start and stop jobs on sites. As illustrated in figure 8.3 and table 8.1, most jobs executed on the actual EGEE grid were compelled to wait in the queue for some period of time, averaging 2,740 s, before starting execution. Combining the wait time and service time, jobs experienced an average sojourn time (time from submission to completion) of 11,700 s. However, the median sojourn time was only 402 s. This time discrepancy, along with the substantially smaller median service time compared to the mean service time, indicated that the distribution of job lengths was right-tailed, with an abundance of short jobs. Issues with the trace data also resulted in an absence of jobs

Figure 8.3: Control Simulation: Job scheduling was based upon observed time stamp information in the trace data. Simulated schedulers were not used, and multiple jobs could run on individual machines.

between December 29, 2008 and January 18, 2009, as evidenced between hours 500 and 1000 in figure 8.3.

As illustrated in figure 8.3, most jobs were queued for some period of time prior to processing. No valid jobs were found to be present in the trace data between December 29, 2008, and January 18, 2009 (approximately 500 to 1000 hours into the trace), resulting in non-utilization of the simulated grid during that period. The control simulation procedure for the six-month trace period required 484 seconds to execute, with the simulator kernel processing a total of 893,805 events. Event processing was consistently variable throughout the length of the simulation, as depicted in figure 8.4.

8.4 Standard Grid Architecture

Significant differences were observed between the simulator and the EGEE data once the scheduling portion of the simulated grid model was enabled. Results of this “standard” simulation, illustrated in figure 8.5 and summarized in table 8.1, indicated greatly increased job queuing behavior, with an order of magnitude increase in observed job waiting time and an attendant increase in job sojourn time. The greatly reduced performance of the simulated grid in this case was determined

Figure 8.4: Control Trace: Behavior of the simulator during the control simulation. Event processing rates fluctuated throughout the simulation.

to have been caused by several factors. A naive implementation of the simulated Condor scheduler, which had to be suspended and resumed on each site to maintain simulator performance, resulted in a mean wait time increase for those sites utilizing Condor. Furthermore, the control simulation permitted multiple jobs to run on single machines, while the simulated schedulers would not start a job on a machine that was marked as busy. Grid sites using the Condor scheduler were modeled using an interval scheduler optimized for High-Throughput Computing (HTC) applications, whereas sites specifying other schedulers were modeled using a zero-interval scheduler optimized for High-Performance Computing (HPC). Thus, the median wait time was reduced to zero as a large number of jobs targeted at sites with non-Condor schedulers started instantly. Nevertheless, the reduced performance of the simulated Condor scheduler resulted in a larger amount of aggregate queuing, indicating poorer performance than was actually observed on the physical EGEE grid.

In the standard architecture simulation, only the job submission time and length information was utilized for scheduling purposes. Simulated scheduler components were utilized to match jobs to free machines and simulate execution. Average job queuing delays increased to 4,470 seconds relative to the control simulation (table 8.1). This additional delay changed the alignment of the job trace and grid map data, permitting a total of 156,804 jobs to execute – an increase of 231 from

Figure 8.5: Standard Architecture Simulation: Jobs were scheduled directly onto the simulated hardware using simulated schedulers. At most one job was permitted to run on any machine at one time.

the control simulation. These additional jobs were slightly longer than the other jobs in the set, as evidenced by the increase in average job length from 9,000 s to 9,020 s. Average sojourn times increased to 13,500 s.

Aggregate scheduling behavior for the standard architecture simulation (figure 8.5) revealed a general increase in total job queuing compared to the control simulation, which contributed to the increased waiting and sojourn times. Peak counts of simultaneous job executions remained similar. Fewer simulator kernel events (816,075) were required to execute this simulation, resulting in a simulation experiment execution time of 388 s. Both the maximum event queue size and maximum event processing rate were observed near the middle of the simulation run, as illustrated in figure 8.6.

8.5 Virtual Organization Cluster Architecture

A final set of simulation experiments were run to evaluate the behavior of Virtual Organization Clusters applied to the grid. In these simulations, the same simulated physical grid is constructed, but every machine connected to the physical grid supports virtualization technology –

Figure 8.6: Standard Architecture Simulation Trace. Peak processing rates were observed near the middle of the run.

either the Kernel-based Virtual Machine (KVM) [129] or Xen [14], depending upon the experiment. Instead of submitting jobs directly to sites, the grid was configured to send jobs to the dedicated VOC created for each Virtual Organization present in the input data. A watchdog component (described in chapter 7.2) monitored the scheduler queue size on each VOC and determined when the VOC should be expanded or shrunk. VOC expansion was effected by submitting pilot jobs through a top-level grid metascheduler, which routed the pilot job to a site with free resources. Upon booting, virtual machines spawned by these pilot jobs joined the VOC scheduler pool and became available to run user jobs.

One simulated VOC was constructed for each Virtual Organization observed in the input job trace, for a total of 50 VOCs. Each VOC utilized the simulated Condor scheduler, again with a naive implementation, and was given a unique grid Compute Element (CE) named with a prefixed version of the VO name. Jobs from the EGEE input trace were modified so as to be submitted to the unique CE corresponding to the job VO, instead of submission directly to a simulated grid site. Each VO CE was equipped with a simulated watchdog, and experiment sets were constructed using both a naive greedy VOC adjustment algorithm and a greedy algorithm with a minimum target level set to 1024. Although this target level was regarded as high, the total number of physical CPU

cores that would be utilized at the target level for all 50 VOCs running simultaneously would have been 51,200, slightly under 50% of the CPU cores present in the simulated grid. Each set of VOC experiments was repeated for both the KVM and Xen hypervisors, using 8.8% and 6.6% overheads (respectively) for compute-bound jobs as measured in prior work [54].

As illustrated in figures 8.8 through 8.11 and summarized in table 8.1, the addition of VOCs to the grid was found to reduce total job queuing significantly. Average wait times for user jobs decreased to 457 s for both the KVM and Xen simulation tests. Since all sites in the simulated grid supported VOCs, and it was assumed that a VOC was always compatible with user jobs from associated VOs, the majority of user jobs could be executed. Sojourn times for end-user jobs remained similar to the standard architecture simulation, at 13,500 s for the KVM simulation and 13,300 s for the Xen test. Average job lengths increased to 13,100 s for the KVM simulation, and 12,800 s for the Xen simulation. Length increases occurred due to a combination of the addition of virtualization overhead and an increase in the average length of the jobs executed. Simulation run times increased to 912 s for the KVM test and 881 s for the Xen test, with higher event rates observed in the earlier phases of both tests (represented in figure 8.7). These simulation time increases were due to the greatly increased number of kernel events processed to effect the simulations: 1,922,895 and 1,924,227 for KVM and Xen, respectively.

An interesting result of the VOC simulation tests, shown in table 8.1, was that the addition of the minimum target level did not generally improve the mean job sojourn time when compared to the naive greedy algorithm. Modest increases (under 1.5%) were observed in this metric using either hypervisor. The cause of this sojourn time increase was attributed to the increased number of user jobs executed by both target level systems. Since the simulated grid accepted a job whenever the specified grid site existed, the earlier creation and persistence of the target level VOCs further reduced the negative effects of timing interdependencies between the VO data and job data in the input trace. As evidenced by the increased service times, this change resulted in an increased mean job length, which yielded an increased mean sojourn time.

8.6 Effects of Virtualization Overhead

The addition of virtualization overhead did increase both the mean and median job service (execution) times, as expected. However, the increase in service times was largely offset by the

Figure 8.7: VOC Simulation Trace (KVM). The highest event rates were observed in the earlier portions of the trace.

Figure 8.8: KVM simulation without the target level set. Note that a larger number of user jobs were executed, compared to the standard and control simulations.

Figure 8.9: KVM simulation with the watchdog target level set to 1024. Higher peak utilization of the grid was observed due to the minimum target level.

Figure 8.10: Xen simulation without the target level set. As in the KVM simulation, a larger set of jobs was executed relative to the standard and control simulations.

Table 8.1: Measured statistics for simulation experiments. Values are exact for job counts and to 3 significant figures for time measurements. All time measurements are in seconds. KVM-0 and Xen-0 refer to simulations without target levels set, while KVM-1024 and Xen-1024 refer to simulations with target levels set to 1024. Since jobs were recorded whenever jobs finished, and pilot jobs for target-level VOC simulations were left running when above the target level, pilot job counts for these simulations were not recorded.

Measure	Control	Standard	KVM-0	KVM-1024	Xen-0	Xen-1024
User Jobs	258,097	258,097	258,097	258,097	258,097	258,097
Pilot Jobs	0	0	149,888	N/M	150,189	N/M
User Jobs Discarded	101,524	101,293	9,151	120	8,939	115
User Jobs Executed	156,573	156,804	248,946	257,977	249,158	257,982
Min Wait Time	0.00	0.00	0.00	0.00	0.00	0.00
Median Wait Time	206	0.00	316	164	316	164
Max Wait Time	823,000	2,380,000	462,000	462,000	462,000	2,090,000
Mean Wait Time	2,740	4,470	457	417	457	418
Stddev Wait Time	15,100	62,800	6,240	6,400	6,240	7,610
Min Service Time	0.00	0.00	0.00	0.00	0.00	0.00
Median Service Time	111	111	885	1,000	878	982
Max Service Time	478,000	478,000	919,000	919,000	901,000	901,000
Mean Service Time	9,000	9,020	13,100	13,300	12,800	13,000
Stddev Service Time	23,200	23,200	31,900	32,300	31,400	31,700
Min Sojourn Time	0.00	0.00	0.00	0.00	0.00	0.00
Median Sojourn Time	402	111	1,190	1,200	1,180	1,170
Max Sojourn Time	1,040,000	2,430,000	920,000	919,000	901,000	2,090,000
Mean Sojourn Time	11,700	13,500	13,500	13,700	13,300	13,400
Stddev Sojourn Time	30,300	68,100	32,600	33,200	32,200	32,900

Figure 8.11: Xen simulation with the watchdog target level set to 1024. Peak grid utilization was observed near the center of the trace but was still less than 10% of the total number of CPU cores on the simulated grid.

decrease in waiting time, resulting in the same mean job sojourn time for both the KVM VOC (without target levels) and standard architecture simulations. A slight decrease in sojourn times was observed when the Xen hypervisor was simulated, as would be expected with the lower virtualization overhead of Xen. At scale, the addition of target levels demonstrated little performance improvement over the naive greedy watchdog algorithm, with an observed increase in mean sojourn times observed for both KVM and Xen. As noted in table 8.1, maximum sojourn times for any job decreased relative to the standard simulation whenever VOCs were in use. However, one job experienced an exceptionally long queuing delay when the Xen VOC test was conducted with target levels set.

To lease the simulated physical grid resources, pilot jobs were required. These jobs had the effect of doubling the utilization of the grid at any time at which user jobs were running, since there was a 1-to-1 correspondence between user jobs and pilot jobs when the naive greedy watchdog algorithm was used. Pilot job requirements for either hypervisor were determined to be nearly identical based on the experiments (table 8.1), with approximately 150,000 jobs required in both cases. This number was less than the total number of user jobs executed in either case due to re-use of existing virtual machines whenever possible. Since the simulation system recorded job results only upon completion of the jobs, and the simulations ended without completion of the pilot jobs

in cases where target levels were set, the total number of pilot jobs utilized was not measured when the target level watchdog algorithm was employed.

Another effect of adding Virtual Organization Clusters to the grid was that 58% more user jobs were able to run, as recorded in table 8.1. With VOCs, user jobs were not submitted to physical grid sites; instead, these jobs were submitted to virtual grid sites created for each Virtual Organization observed in the trace. As a result, discrepancies between the grid map data and job trace data did not result in job errors. However, timing issues between VO registration (which resulted in associated VOC head node creation) and job arrival did result in a small set of user jobs failing to execute (3.5% in the worst case, compared to 39% in the standard simulation). Since a greater number of jobs executed, while total queuing across the grid was substantially decreased, it was determined that the primary contributor to queuing on the actual grid system was a result of jobs targeting specific grid sites instead of being load-balanced among all sites.

Chapter 9

Prototype Test Results¹

Following completion of the prototype test bed (chapter 7), performance tests were conducted using synthetic and operational workloads. The purpose of synthetic workload testing was to observe the behavior of both the underlying physical system and hosted Virtual Organization Clusters when engineered tasks with specific properties were run on the system. To provide more robust results based upon real workloads, several VOCs were deployed operationally onto the Open Science Grid, where they transparently received and processed end-user jobs.

In this chapter, basic performance tests are presented in section 9.1. Test results from the dynamic VOC provisioning system are described in section 9.2, after which the addition of overlay scheduling to hosted VOC systems is discussed in section 9.3. Results from operational testing are presented in section 9.4. Finally, scalability tests on the Stoker management application are presented in section 9.5.

9.1 Performance Tests

Several tests were conducted to ensure that the performance of Virtual Organization Clusters was not unreasonable. In order to evaluate the behavior of the transparent test VOC, two major installations were performed: a Slackware Linux 12 installation directly on the physical hardware and a CentOS 5.1 installation into a virtual machine image. Following the installations, boot times were measured for both the physical and virtual systems. A High Performance Linpack (HPL)

¹Most of this chapter has been published in [54, 114, 115, 117].

benchmark was performed on the physical compute nodes, followed by a second HPL benchmark on the VOC. Several different process grid sizes were used in the benchmark tests. To determine the cause of observed poor performance with HPL on the prototype VOC, a set of network tests was conducted. These tests included bandwidth measurement and ping Round-Trip Time (RTT) measurements to assess network latency.

To effect the performance tests, the High Performance Computing Challenge (HPCC) benchmark suite was used, which included HPL. Boot times were measured manually for the physical boot procedure, while a simple boot timing server was constructed to measure VM booting time. Network bandwidth was measured using both the Iperf bandwidth measurement tool and the RandomRing bandwidth assessment in the HPCC suite. Latency in network communications under load also was assessed using the RandomRing benchmark. Measurement of Round-Trip Time (RTT) of ICMP Echo packets generated by the UNIX ping tool was used as an additional measure of network latency both under load (with the HPCC suite running) and without computational load on the VOC.

9.1.1 System Performance

Following system installation, boot times were recorded for both the physical and virtual systems. Since VM startup was scripted, automated means were devised to measure the VM boot times. A simple server was deployed on the physical utility node, which received boot starting notifications from the physical nodes and boot complete notifications from the associated virtual nodes. Timing of the boot process was performed at the server side, avoiding any clock skew potentially present between physical and virtual nodes, but possibly adding variable network latency. Boot times for the physical nodes were subject to greater variation, as these were measured manually.

Results of the boot time tests have been summarized in table 9.1 and figure 9.1. For the physical system, the boot process was divided into three phases: a PXE timeout, a GRUB timeout, and the actual kernel boot procedure. While total boot times ranged from 160 to 163 seconds, 105 to 107 seconds of that time were utilized by the PXE timeout, and 10 seconds were attributed to the GRUB timeout. Thus, the actual kernel boot time ranged from 43 to 46 seconds. In contrast, the virtual compute nodes required 61.2 to 70.2 seconds to boot. These virtual machines were configured with a different operating system (CentOS 5.1) and started approximately 10 additional processes at boot time, compared to the physical systems. As a result, not all the boot time discrepancy could

Figure 9.1: Average boot times of physical nodes (Slackware 12) and VMs (CentOS 5.1). Total physical boot times are increased due to BIOS and embedded controller menu access timeouts.

Table 9.1: Boot Times (seconds) for both the physical systems (Slackware 12) and virtual machines (CentOS 5.1). The physical system boot process involved several timeouts, including delays for the Power-On Self Test, disk controller management interface, Preboot eXecution Environment (PXE) for network booting, and a bootloader management menu.

Physical Node				VM
Statistic	PXE Timeout	Total Boot	Actual Boot	VM Boot
Minimum	105	160	43	61.2
Median	106	160.5	44	65.4
Maximum	107	163	46	70.2
Average	106.4	160.9	44.5	65.5
Std Deviation	0.63	1.03	1.09	2.54

Table 9.2: Performance of Slackware 12 compared to CentOS 5.1 when installed directly on the hardware.

Process Grid (PxQ)	14x2
Problem Size	77,000
CentOS GFLOPS	115.6
Slackware GFLOPS	129.6
Performance Increase	12.11%

Table 9.3: Physical Cluster vs. VOC performance for original cluster implementation (Slackware 12) with CentOS 5.1 VOC.

Process Grid (PxQ)	1x1	7x2	7x4
Problem Size	10,300	38,700	54,800
Physical GFLOPS	7.29	74.39	143.45
VOC GFLOPS	6.57	29.74	63.09
Virtualization Overhead	9.86%	60.02%	56.02%

be attributed to virtualization overhead. Nonetheless, the overhead was small enough that booting the VOC did not require an inordinate amount of time.

Following the boot procedures, HPL benchmark data were obtained for both the physical and operational VOC nodes (tables 9.2 and 9.3). First, an HPL benchmark previously conducted on the prior CentOS physical installation was performed on the Slackware hosts. A 12% performance increase was noted as a result of the Slackware installation. Although the cause of this increase could not be conclusively determined, it was believed that the customization of the installation – including Linux kernel optimization – and minimization of unnecessary services contributed to the additional performance. This result showed that keeping the host configuration as lightweight and simple as possible not only made it easier for the cluster administrator to maintain but also increased the overall performance, thereby benefiting all VOs using the cluster.

One limitation of the original implementation was the inability to allocate a large, contiguous block of memory for the KVM guest process on the Slackware 12 compute nodes. Although the version of KVM in use at the time permitted up to 2 GiB of RAM to be assigned to the guest, a segmentation fault was observed when attempting to map more than 1.2 GiB to the process. An investigation determined that the 32-bit host system was mapping its shared object libraries into the middle of the userspace region. Thus, while 3 GiB of virtual memory was available to processes, the memory space was bisected by the shared objects and was thus non-contiguous. In order to retain the ability to perform a contiguous mapping (for performance and locality of reference reasons), a

Figure 9.2: Compute-bound application performance as measured on the revised installation (CentOS 5.2 on both the physical system and VOC).

Figure 9.3: Single-Core VOC Performance (28 VOC Nodes on 14 Physical Nodes, CentOS 5.2 installed on both)

Table 9.4: Physical system performance compared to a VM for a compute-bound single HPCC MPI process (CentOS 5.2).

Process Grid	1x1 Physical	1x1 Xen VOC	Xen Overhead	1x1 KVM VOC	KVM Overhead
Problem Size	10300	10300	N/A	10300	N/A
G-HPL (GFLOPS)	7.913	7.393	6.566%	7.218	8.771%
G-PTRANS (GB/s)	0.729	0.588	19.415%	0.635	12.946%
G-Random Access (GUP/s)	0.002	0.001	35.519%	0.002	15.818%
G-FFTE (GFLOPS)	0.799	0.658	17.733%	0.461	42.370%
EP-STREAM Sys (GB/s)	3.866	3.375	12.704%	3.808	1.491%
EP-STREAM Triad (GB/s)	3.866	3.375	12.704%	3.808	1.491%
EP-DGEMM (GFLOPS)	8.348	7.689	7.892%	7.682	7.977%
RandomRing Bandwidth (GB/s)	N/A	N/A	N/A	N/A	N/A
RandomRing Latency (μs)	N/A	N/A	N/A	N/A	N/A

Table 9.5: Physical System vs. VOC, One Dual-Core VM per Physical Node (32 processes, CentOS 5.2)

Process Grid	7x4 Physical	7x4 Xen VOC	Xen Overhead	7x4 KVM VOC	KVM Overhead
Problem Size	58600	58600	N/A	58600	N/A
G-HPL (GFLOPS)	169.807	118.067	30.470%	25.178	85.173%
G-PTRANS (GB/s)	0.867	0.496	42.818%	0.069	91.985%
G-Random Access (GUP/s)	0.014	0.009	35.910%	0.004	73.082%
G-FFTE (GFLOPS)	2.287	1.717	24.899%	0.399	82.556%
EP-STREAM Sys (GB/s)	59.046	62.678	-6.151%	82.599	-39.889%
EP-STREAM Triad (GB/s)	1.845	1.959	-6.151%	2.581	-39.889%
EP-DGEMM (GFLOPS)	8.271	7.669	7.269%	6.901	16.559%
RandomRing Bandwidth (GB/s)	0.023	0.017	23.425%	0.007	67.419%
RandomRing Latency (μs)	74.444	150.831	102.611%	290.463	290.179%

64-bit operating system was installed on the host systems. Since Slackware Linux was available only in a 32-bit architecture, 64-bit CentOS 5.2 was chosen as the new host OS.

Following the host software change, the High Performance Computing Challenge (HPCC) benchmark suite[107], which includes HPL, was executed on the host systems and in the VOC, repeating the previous tests with process grid dimensions of 1x1 and 7x4. These tests were conducted using both the KVM and Xen hypervisors, and these tests made use of the multicore guest support features of both systems, allowing for both single-core and dual-core guests. The results of these tests, published in [54], are presented in figures 9.2, 9.3, and 9.4, and in tables 9.4, 9.5, and 9.6.

HPL and HPCC tests were performed on both the physical machines and the VOC, using the same process grid layouts and problem sizes across administrative domains. These tests were roughly divided into three categories: compute-bound, embarrassingly parallel, and latency-sensitive. The first of these categories, compute-bound, represented High-Throughput Computing (HTC) jobs –

Table 9.6: Physical System vs. VOC, Two Single-Core VMs per Physical Node (32 processes, CentOS 5.2)

Process Grid	7x4 Physical	7x4 Xen VOC	Xen Overhead	7x4 KVM VOC	KVM Overhead
Problem Size	58600	58600	N/A	58600	N/A
G-HPL (GFLOPS)	169.807	130.862	22.935%	81.401	52.063%
G-PTRANS (GB/s)	0.867	0.830	4.302%	0.447	44.968%
G-Random Access (GUP/s)	0.014	0.011	22.941%	0.004	70.643%
G-FFTE (GFLOPS)	2.287	0.746	67.380%	1.751	23.449%
EP-STREAM Sys (GB/s)	59.046	62.382	-5.650%	73.110	-23.818%
EP-STREAM Triad (GB/s)	1.845	1.949	-5.650%	2.285	-23.818%
EP-DGEMM (GFLOPS)	8.271	7.726	6.588%	7.114	13.979%
RandomRing Bandwidth (GB/s)	0.023	0.007	68.779%	0.027	-17.148%
RandomRing Latency (μ s)	74.444	125.258	67.259%	228.383	206.787%

the type of job that would run in a “vanilla” Condor universe. As shown in table 9.3, the original virtualization overhead in terms of HPL observed throughput was only 9.86% for this type of job, using Slackware host systems. Under CentOS 5.2 host systems (table 9.4), the overhead decreased to 8.77% with the KVM hypervisor. Xen was slightly more efficient, at 6.57%.

Embarrassingly parallel jobs (represented by EP prefixes in tables 9.4, 9.5, and 9.6) exhibited better performance inside virtual machines than they did when run directly on the physical hardware, except for the 1x1 process grid size. These jobs had good spatial locality of reference in data accesses, but low temporal locality [107]. KVM outperformed Xen on this job class. This result was attributed to the fact that KVM emulates all its hardware devices, including the hard disks, while Xen shares the physical system hard disk with extra buffering [14]. Since the emulated disk could be located in host system RAM, its performance would have been higher than the physical drive.

MPI jobs that utilized inter-node communications (HPL, Random Access, Fast-Fourier Transform, and RandomRing) incurred substantial performance overheads on VMs for a 7x4 process grid. With HPL, these overheads were observed to be 52% for the single-core KVM test, 23% for the single-core Xen test, 85% for the dual-core KVM test, and 30% for the dual-core Xen test. Network latency was suspected for this observed overhead, as latency has been implicated as a cause of performance reduction in prior studies involving MPI [108, 107], and RandomRing latencies were over 100% higher than the physical systems for all tests except single-core Xen (67%). With MPI applications comprising a significant fraction of all scientific computing endeavors, it was desirable to be able to deploy a VOC that had good MPI performance. Additional network investigations were undertaken to determine the source of the latency.

Figure 9.4: Dual-Core VOC Performance (14 VOC Nodes on 14 Physical Nodes)

9.1.2 Network Performance

Several networking issues were suspected in the initial setup. Two VMs shared a single Linux TUN/TAP bridge to a single physical Gigabit Ethernet port, which was also shared by the host for host-level network connectivity (figure 9.5). Each KVM instance also emulated an Intel 82540EM Gigabit Ethernet Network Interface Card (NIC), which was presented to the guest OS and utilized as if the card were an actual physical device. The physical NIC on the host was configured in promiscuous mode, bypassing the internal NIC packet filtering code and offloading the low-level network processing onto the host CPU. Furthermore, the bridge component of the kernel and NIC emulation components of KVM also relied upon the host CPU to effect communications. As a result, the host CPU was taxed not only with the computationally-intensive HPL routines, but also with low-level networking operations typically carried out in the NIC hardware.

Table 9.7 summarizes the results of cluster network testing using Iperf, ping, and the Random-Ring bandwidth and latency benchmarks in HPCC. Iperf showed 941 Mbps of available bandwidth when the cluster was not under load, decreasing to 882 Mbps when HPL benchmarks

Figure 9.5: Virtual Machine Bridged Networking: The physical hardware NIC was operated in promiscuous mode, allowing the virtual machines layer 2 connectivity to the physical network.

were running on the hosted VOC. This decrease could be attributed to inter-node MPI communications, which would have consumed a portion of the network resources. The decrease in measured bandwidth between VMs was more significant, dropping from 708 Mbps to 636 Mbps for communications between VMs hosted by different physical nodes. Communications between two VMs sharing the same bridge were found to have substantially lower available bandwidth, with only 499 Mbps (roughly half the nominal bandwidth of Gigabit Ethernet) available when not under load. During the HPL tests, this intra-bridge bandwidth fell to an available level of 206 Mbps. Bandwidth as measured under load by the Random-Ring benchmarking was substantially lower in all cases: 544 Mbps for the physical hosts, and 24 Mbps to 32 Mbps for the VMs. Lower bandwidth was observed when the MPI rings included intra-bridge links (SB column of the table) than when only inter-bridge links (links between VMs hosted by different physical systems) were included in the MPI rings. Unlike the Iperf tests, the Random-Ring test data for bandwidth across intra-bridge links is also averaged with the available bandwidth between bridges; without this averaging, it is likely that the intra-bridge links would have shown lower available bandwidth, based upon the Iperf tests.

Latency between nodes was found to be higher between virtual hosts than between the underlying physical hosts. Measuring the Round-Trip Time (RTT) of the ping (ICMP echo) operation yielded an average of 106 μs without load, increasing to 191 μs under load. Ping operations across a single bridge (intra-bridge) required longer times to execute: 215 μs in the absence of load, increasing to 360 μs under load. RTTs for ping operations between VMs on different hosts were the longest, beginning at 312 μs and increasing to 484 μs under load, suggesting that inter-bridge

Table 9.7: Bandwidth and latency (ping RTT) as measured on the physical and virtual clusters.

Condition	No-Load			Under Load		
Parameter	P	SB	BTB	P	SB	BTB
Iperf (Mbps)	941	499	708	882	206	636
RRB (Mbps)	N/A			544	24	32
Ping RTT (μs)	106	215	312	191	360	484
RRL (μs)	N/A			54	379	233

Key: P – Physical, SB – Virtual links across the Same Bridge, BTB – Virtual links from one bridge (physical host) to another, RRB – RandomRing Bandwidth, RRL – RandomRing Latency

communications incurred the greatest latency. However, the Random-Ring benchmarks indicated greater latency between VMs sharing a single bridge, with bridge-to-bridge latencies 146 μs lower at 233 μs . Both VM latency figures were an order of magnitude higher than the measured 54 μs latency on the physical network.

One significant limitation of the network architecture used for the first implementation was identified as a result of the test procedures. Two VMs and one physical host were configured to share one physical Ethernet NIC on each physical node. Thus, parallel communication between two pairs of VMs on two separate physical hosts would have been converted to sequential networking operations, with packet queuing needed either at the bridges or at the physical switch. Queuing, in turn, could have introduced added latency into the communications, which may have reduced MPI performance. Moreover, an increase in queuing could have increased packet transmission time, thus causing the TCP protocol used by MPI to place more packets in flight to fill the sliding sender window. Such an increase in packet saturation on the network used in the test cluster has been shown to increase queuing delays, thereby increasing latency and further aggravating communications difficulties [116].

The combination of virtual machine overhead, latency introduced by the bridged networking, and delay properties of the underlying physical network resulted in a network environment that could not support MPI or other latency-sensitive applications inside VOCs. Latency in the underlying physical network was already on the order of 50 μs for one-way unicast traffic. VOC traffic latency was greatly increased as a result of the addition of the emulated NIC, the use of the Linux bridge facility, and the reassignment of low-level network processing from the physical NIC to the host CPU. The unsatisfactory performance results obtained from this experiment indicated that an alternative mechanism for providing network connectivity to VMs, such as VMM-bypass networking [106], would be needed if VOCs were to support HPC jobs.

Figure 9.6: Testbed architecture for the dynamic provisioning tests

9.2 Dynamic Provisioning System

A prototype of the dynamic Virtual Organization Cluster scheduler was implemented, and tests of the system were conducted using synthetic workloads. For analytical simplicity, the system only supported a single VO, with a single virtual machine slot per physical host, for a total limit of 16 slots. This simplifying design assumption permitted the use of a minimalistic physical system policy, so that the performance and behavior of the unrestricted mechanism could be observed. As a further simplification, the VMs used for the VOC in this test were all spawned from a single 20 GB image on a shared filesystem. Figure 9.6 depicts the architecture of the test system hardware.

Several sets of tests were conducted using the prototype system, the first of which was an analysis of the approximate time required to boot each VOC such that it joined the Condor pool. Since all VMs started from an identical state — the result of using a single virtual disk image to spawn all VMs — the boot times were assumed to be constant for all members of the same VOC. Two test suites were employed to observe job scheduling behavior. In the first suite, jobs were submitted locally: that is, directly to the Condor queue, without any use of Globus. Globus was used as the vehicle for job submission in the second suite, allowing its effects to be observed.

Each test suite consisted of five tests, in which the periodicity of job submission, size of each job group submission, and run length of each job were varied. These tests were arranged as follows:

- Two submission groups of 50 jobs each were submitted with sufficient temporal separation so as to execute the vast majority of jobs from the first submission group, prior to execution of the second submission group. This test was designed to simulate submitting large groups of jobs in which the results of the first group were retrieved before submitting the second group.
- Periodic submission of 10 jobs, each with a 10-second execution time, 90 seconds apart
- Periodic submission of 10 jobs, each with a 10-second execution time, 30 seconds apart
- Periodic submission of 10 jobs, each with a 1-second execution time, 30 seconds apart

The purpose of the latter three tests was to observe the behavior of the system under regular periodic loads, with variations in the period, job size, and per-job execution time. Execution times were varied in order to determine the sensitivity of the system to the boot time latency, while period and size variations were performed to test the responsiveness of the watchdog.

9.2.1 VM Boot Times

Tests were performed using local job submission (directly to the Condor queue) to measure the boot times for the VMs comprising the VOC. Submissions were performed in groups of 10 one-second jobs, with a period of 30 seconds between groups. The boot process for a VM was considered to be complete once it joined the Condor pool, as observed by the watchdog. As shown in figure 9.7, the first VM booted in response to incoming jobs joined the pool approximately 60 seconds after the first job was submitted, or about 55 seconds after the watchdog observed the first job and started the VM.

Since the watchdog required approximately 6 seconds to start all 10 initial VMs, a corresponding delay of approximately 7 seconds was observed between the time at which the first VM joined the Condor pool and the time at which the tenth VM joined the Condor pool. At a test wall time of approximately 38 seconds, the watchdog responded to the second batch of submitted jobs and began to increase the size of the VOC, continuing until the 44 second mark, at which point the 16 VM slots were exhausted. The additional 6 VMs joined the Condor pool between wall clock times of 92 and 101 seconds, corresponding to boot times in the range of 54 to 57 seconds. No additional VMs could be started once the slots were exhausted at 101 seconds, after which point the 16 running VMs were able to complete the remaining 1-second jobs quickly.

Figure 9.7: Virtual Machine boot delays relative to job submission time. VMs are started in response to jobs arriving in the queue. Since each VM is a virtual Linux machine, there is a boot delay between VM start time and the time at which the VM joins the Condor pool and is ready to accept jobs. The total count of these booted machines corresponds to the size of the Condor pool for the VOC.

Variations in VM boot time were expected, owing to dynamic processes that must occur during VM boot. These processes include the allocation of memory to the VM, initialization of the VM kernel, acquisition of a DHCP lease by the VM, and the starting of run-time services. Based on the test results, a conservative upper bound of 60 seconds was attributed to the VM boot process.

9.2.2 Jobs Submitted Locally

To effect completion of the first test suite, batches of jobs were submitted locally. The first test utilized two groups of 50 jobs each, with a sufficiently long delay between submission to allow all but two of the first batch to complete before submitting the second batch. As shown in figure 9.8, the watchdog started the maximum number of VOC nodes by 20 seconds into the test. The majority of the first set of jobs completed rapidly between 162 and 198 seconds wall clock time, or about 178 seconds after submission. Given the 60-second boot time assumption, the total run time for 48 10-second jobs on 16 VMs was approximately 118 seconds. Between sets, the watchdog was observed to reduce the size of the VOC from 16 VMs to 2 VMs. The second set of jobs completed in approximately half the time as the first, with a clear “step-down” pattern observed in the Condor queue between 280 and 310 seconds. While this pattern might have been partly attributed to the need to reboot the VMs that were stopped during VOC contraction, it is important to remember that

Condor only schedules jobs on a periodic basis, as it is designed for high job throughput, not high performance. Therefore, some of the “step-down” behavior could have been attributed to Condor simultaneously starting fewer jobs than were slots available.

As shown in figures 9.9 through 9.11, the watchdog continued to exhibit predictable behavior similar to that observed in the first test. Whenever the number of jobs waiting in the queue dropped below the number of running VMs in the VOC, the VOC was contracted by terminating VMs. Conversely, whenever more jobs were waiting than VMs were running, as long as VM slots remained available, the watchdog expanded the VOC by adding VMs. A noticeable delay between queue size and VM count was observed in both cases, which was the exact behavior expected from the periodic sampling done by the watchdog. Longer delays were observed between initial job submission and job completion, owing to the time required to boot the VMs.

The effect of extremely short jobs, or exceptionally long periods related to job execution length, were evident in the results, as illustrated in figures 9.9 and 9.11. Since the watchdog employed a simplistic VM scheduling policy, it terminated VMs as soon as VOC sizes were found to exceed queue sizes. This aggressive VM termination had a negative effect on throughput, as illustrated by comparing figure 9.9 to figure 9.10 and figure 9.10 to figure 9.11. In the case of relatively long periodicity relative to execution time, it was necessary to re-expand the previously contracted VOC, thereby incurring VM boot delays. For exceptionally short-running jobs submitted regularly, the initial cluster of submissions completed quickly, allowing the entire VOC to be removed from operation. Two interesting, but mutually disadvantageous, phenomena were observed after the next group of jobs arrived in the queue (figure 9.11 at approximately 130 seconds wall time). Due to caching of the virtual machine monitor process, and its initial read-only VM data, on the physical host, the VOC nodes were able to boot somewhat faster during VOC re-start. However, the rapid job execution and simplistic scheduling algorithm in the watchdog combined to limit the total size of the VOC to 10 nodes following the re-start. The result of this combination of properties was exceptionally low throughput following the restart.

9.2.3 Jobs Submitted Through Globus

A second suite of identical tests was executed using the Globus job manager as the submission vehicle. In order to avoid Globus errors resulting from multiple simultaneous job submissions, it was necessary to introduce a small delay of two seconds between the submissions of individual

Figure 9.8: Two submissions of 50 jobs, 10-second execution time, submitted locally

Figure 9.9: Submitted 10 jobs every 90 seconds, 10-second execution time, submitted locally

Figure 9.10: Submitted 10 jobs every 30 seconds, 10-second execution time, submitted locally

Figure 9.11: Submitted 10 jobs every 30 seconds, 1-second execution time, submitted locally

Figure 9.12: Two submissions of 50 jobs, 10-second execution time, submitted through Globus

Figure 9.13: Submitted 10 jobs every 90 seconds, 10-second execution time, submitted through Globus

Figure 9.14: Submitted 10 jobs every 30 seconds, 10-second execution time, submitted through Globus

Figure 9.15: Submitted 10 jobs every 30 seconds, 1-second execution time, submitted through Globus

jobs. As a result, the process of submission of a batch of jobs through the Globus system was longer than the process of local submission, resulting in the slower queue size growth visible in figures 9.12 through 9.15. In addition, some non-constant delays were observed between submission of jobs to Globus and the time at which the jobs were delivered to Condor. This delay, especially evident in the queue size jitter shown in figure 9.13, was not particularly alarming, once again due to the emphasis taken by both systems on high throughput, as opposed to high performance.

Although net throughput was slightly reduced by the late arrival of the last set of jobs, the addition of Globus as a “buffer” reduced the extremity of the VOC contraction. While the size of the VOC was reduced, necessitating the rebooting of some nodes, the VOC was never completely removed from operation as it was in the local test. This buffering effect was not observed with extremely short jobs, as depicted in figure 9.15, where the VOC was briefly completely terminated.

9.3 Overlay Scheduling

Although the overhead of adding virtualization to grid systems had previously been evaluated [54], and the overhead of using the IPOP network overlay had been independently studied [72], the combination of both overheads in an overlaid scheduling environment with Virtual Organization Clusters had not been measured. Since the overhead of virtualization would primarily affect job service time in a grid system with sufficient physical resources to handle all jobs concurrently, a chief concern was the amount of latency that might be added by the overlay scheduling system, which necessitated the use of an overlay network. In order to ensure that this overlay overhead would not have unexpected detrimental impacts on compute-bound jobs running within the virtual machines, tests were conducted using an Open Science Grid (OSG) [125] site configured to support virtualization. The OSG grid site was configured with 16 dual-core compute nodes, each with an Intel Xeon 3070 CPU and 4 binary gigabytes (GiB) of Random Access Memory (RAM), with the Kernel-based Virtual Machine (KVM) [129] hypervisor running within a 64-bit installation of CentOS 5.2. Virtual machine images were configured with 32-bit CentOS 5.2 and located on a shared Parallel Virtual FileSystem (PVFS) [28] store. Virtual machine instances were booted directly from the image located on the shared filesystem, without first staging the image to the local compute nodes, using the “snapshot” mode of KVM. These shared images were made available in a read-only configuration, with non-persistent writes redirected to a temporary file on local disk storage at each

compute node. Internet connectivity for the test site was provided by an edge router using Network Address Translation (NAT), with the physical compute nodes isolated in a private IPv4 subnetwork. OSG connectivity was provided through the standard OSG software stack including Globus [64].

A VOC head node was constructed using a virtual machine hosted by an off-site laboratory workstation. Condor [155] was installed on the head node to serve as a scheduler, and synthetic workload jobs were submitted directly to the VOC local pool. A watchdog daemon process, using the naive greedy watchdog algorithm with added support to maintain a minimum number of running VMs at all times, was run on the same head node. This watchdog created pilot jobs, which were submitted through a Globus client to the OSG test site, in response to the arrival of synthetic workload jobs in the VOC Condor queue. The pilot jobs started virtual compute nodes, which joined an IPOP network anchored at the workstation by contacting its bootstrap service. Once connected to the private IPOP network, the virtual compute nodes joined the Condor pool created by the collector on the VOC head node (the workstation-hosted virtual machine, also joined via IPOP), and Condor scheduled and executed the actual test jobs. Whenever the watchdog daemon determined that an excess number of pilot jobs were running in comparison to the size of the Condor queue, the pilot jobs were instructed to terminate, causing the virtual machines to be terminated.

9.3.1 Overlay Scheduling and Networking

To measure the relative performance difference of using a VOC with overlay scheduling and IPOP overlay networking, two sets of tests were conducted. A synthetic workload consisting of a 10-minute sleep procedure was devised, in order to approximate compute-bound jobs without incurring potential variations in service times that could result from running an actual compute-bound job within a virtual machine. In the control trials, a batch of 50 sleep jobs was submitted directly to the local scheduler on the physical grid site head node. For the experiment trials, the same batch of 50 jobs was submitted directly to the Condor central manager running within the VOC. Total makespan times were collected for both sets of trials, and each trial was repeated 10 times to reduce the effects of random variation in observed makespan lengths. Descriptive and relative statistics were computed for the makespan times. Throughput measures in jobs per second were also computed.

Results of the trials, summarized in table 9.8, indicated a slight increase in average makespan time (less than one half of one percent) for jobs submitted through the overlay scheduling system, compared to jobs submitted directly to the physical cluster scheduler. This increased makespan

Table 9.8: Observed makespan lengths and system throughputs for 10 overlay experiment trials

Test	Overlay		Change	
	None	IPOP	Absolute	Relative
Minimum Makespan (s)	2706	2714	8.000	0.2956 %
Median Makespan (s)	2709	2720	11.00	0.4061 %
Maximum Makespan (s)	2710	2737	27.00	0.9963 %
Mean Makespan (s)	2708	2722	13.50	0.4985 %
Makespan Standard Deviation (s)	1.414	7.735	6.321	447.0 %
Minimum Throughput (Jobs · s ⁻¹)	1.845×10^{-2}	1.827×10^{-2}	-1.820×10^{-4}	- 0.9865 %
Median Throughput (Jobs · s ⁻¹)	1.846×10^{-2}	1.839×10^{-2}	-7.467×10^{-5}	- 0.4045 %
Maximum Throughput (Jobs · s ⁻¹)	1.848×10^{-2}	1.842×10^{-2}	-5.447×10^{-5}	- 0.2948 %
Mean Throughput (Jobs · s ⁻¹)	1.846×10^{-2}	1.837×10^{-2}	9.000×10^{-5}	- 0.4954 %
Throughput Standard Deviation	9.644×10^{-6}	5.207×10^{-5}	4.243×10^{-5}	439.9 %

Figure 9.16: Autonomic VOC size adjustment behavior when executing long jobs without an overlay network (average of 10 repetitions of the experiment).

length corresponded to a similarly small decrease in job throughput resulting from the addition of the overlay. In the worst case observed in all trials, the maximum makespan and minimum throughput were affected by less than one percent. Variations in makespan and throughput observations between trials was substantially increased by over 400% when the overlay scheduler and network were added, likely due to the addition of a second layer of interval scheduling with the additional Condor pool overlaid on top of the physical Condor pool. Plotted traces of mean observations (figures 9.16 and 9.17) further confirmed the minimal overhead of the VOC overlay system.

9.3.2 VOC Adjustment Policy

A second experiment was performed to evaluate the behavior of the watchdog daemon when monitoring the private scheduler queue and adjusting the size of the Virtual Organization Cluster. As illustrated in figure 9.18, the simple greedy watchdog algorithm proved to be over-responsive when batches of microbenchmark (10-second) jobs were submitted to the scheduler. The VOC was

Figure 9.17: Autonomic VOC size adjustment behavior when executing short jobs privately scheduled using the IPOP overlay network (average of 10 repetitions of the experiment).

rapidly expanded to use all 16 available processor cores at the first watchdog interval. A delay of approximately 60 seconds was observed while the VOC nodes booted, after which the short user jobs quickly ran to completion. Even though additional jobs arrived in the queue while the VOC nodes were booting, all jobs from the first two batches had completed within 140 seconds. At this time, the size of the VOC was shrunk to zero, causing the virtual machines to vacate the physical systems completely. When another batch of short jobs arrived at 180 seconds into the test, all 16 virtual machines had to be restarted, resulting in another boot delay.

To provide a buffer against excessively short jobs, a Delayed Response adaptation algorithm was devised. This policy resulted in the immediate creation of a pair of VOC nodes to remain active at all times for the handling of instantaneously short (by design or by failure) jobs. In addition, the VOC was expanded by only one node at a time, and expansion only occurred if the size of the Condor scheduler queue exceeded the VOC size for at least 10 watchdog intervals. Similarly, the VOC was decreased by one node at a time, to a minimum size of two nodes, only when the size of the VOC exceeded the size of the Condor queue for at least 10 watchdog intervals. The results of submitting two batches of short jobs have been illustrated in figure 9.19, which shows a slow increase in the number of VOC nodes in response to the first batch of jobs. A slow decrease in the number of VOC nodes was observed between batches, followed by another slow increase as the second batch of jobs arrived. Once all jobs from the second batch completed, the VOC size slowly declined to the minimum size (two) specified by the policy.

Figure 9.18: Simple Greedy Algorithm for autonomic VOC size adjustment: VMs are started whenever there are excess jobs in the scheduler queue and free physical nodes available. Once the jobs complete and the queue size decreases, VMs are terminated quickly.

Figure 9.19: Delayed Response Algorithm for autonomic VOC size adjustment: a minimum of 2 VMs are kept in operation at all times, and VMs are started and stopped in response to queue size trends over a time series of samples.

Figure 9.20: Short operational test (44 hours) with the physical cluster configured to support a 16-node Virtual Organization Cluster dedicated to the Engage Virtual Organization.

9.4 Operational Tests

After completion of the prototype performance tests with synthetic workloads, the prototype cluster was configured for operational tests. In these experiments, Virtual Organization Clusters were transparently provided to specific Virtual Organizations on the Open Science Grid, and jobs from those VOs were executed in 32-bit CentOS virtual machines. On June 1, 2009, a short operational test was started, in which a single VOC was placed into service on behalf of the Engage VO, using the delayed response watchdog provisioning algorithm with a minimum of two VMs and a maximum of sixteen VMs. After approximately 44 hours of testing, the VOC was removed from service on June 3, 2009. As visualized in figure 9.20, several bursts of jobs arrived during the test period, and these bursts were accommodated by increasing the size of the VOC to the maximum specified level (16 nodes).

Following the short operational test, a long operational deployment – approximately two months in length – was effected using the same watchdog algorithm. Two VOCs were attached to the Open Science Grid, with one VOC dedicated to the Engage VO and the other VOC dedicated to the NanoHub VO. Both VOCs were set to a minimum size of two VMs and a maximum size of sixteen VM, which resulted in utilization of all 32 physical CPU cores whenever both VOCs were at

Figure 9.21: Long operational test: Engage VO. A second operational VOC, dedicated to the NanoHub VO, was sharing the same hardware.

maximum size. The long-running operational experiment commenced on June 4, 2009 and completed on August 17, 2009.

As illustrated in figure 9.21, jobs associated with the Engage VO continued to arrive in bursts for the first two thirds of the test period, resulting in temporary increases in VOC size. Bursts of jobs associated with the NanoHub VO (figure 9.22) were less frequent, significantly smaller in size, and limited to the first third of the test period. Subsequent analysis determined that most NanoHub jobs, and an increasingly larger number of Engage jobs, required 64-bit operating environments. Since the prototype VOCs provided 32-bit environments, fewer jobs were sent to the prototype system as August approached.

After completion of the operational tests, the prototype system became obsolete for research purposes. The Intel Xeon processors installed in the physical compute nodes were equipped with the first generation of virtualization extensions, which did not include extended page tables or virtualized Input/Output devices. Rather than immediately discarding the hardware, a single Virtual Organization Cluster was deployed for the STAR VO [6], using a custom virtual machine image provided by the VO administrators. In this deployment, the VOC was still provided transparently on OSG by the system, although the VO provided the software stack in a sort of “semi-transparent”

Figure 9.22: Operational test: NanoHub VO. A second operational VOC, dedicated to the Engage VO, was sharing the same hardware.

arrangement. Thus, the prototype implementation was converted into a production system, and it was still in service as of October 2009.

9.5 System Management with Stoker²

Once the prototype system was constructed and placed into operational service, ongoing system maintenance was performed using the Stoker remote administration tool. In order to measure overhead created by the resolution and threading management processes and to assess the performance improvement of parallelizing the Stoker command execution process, several tests were conducted. The first test, with results shown in figure 9.23, parallelized a short-running `/bin/hostname` task. Parallel execution was found to be of limited utility beyond 4–6 threads in this case due to the extremely short run time of the program ($\approx 0.001s$) relative to the total time necessary to spawn a thread and execute a remote command via SSH ($\approx 0.225s$). This test was conducted on a low-latency network inside a private computing cluster.

The next test (figure 9.24) involved a longer running job: a ten-second process sleep that was engineered to simulate the restart procedure of a network service. This test was conducted on

²The contents of this section have been published in [114].

Figure 9.23: Stoker performance on a low-latency 16-node cluster when parallelizing a hostname job the same private, low-latency cluster as the first test. Performance was measured to the limit of one thread per target (16 nodes in this case). It should be noted that no performance improvement was measured for 8–15 threads.

Since the 10-second sleep procedure had a known run time, network and SSH overhead could be measured. As shown in figure 9.25, these overheads generally decreased in an absolute sense until 14 threads were utilized. Beyond this point, the overhead of spawning new threads began to increase. However, overhead as a percentage of the total time remained constant until 16 threads, the limit for this test, were utilized.

To determine whether overheads would be more significant across a higher latency network, another 10-second sleep test was conducted on 27 public laboratory workstations. Results of this test, summarized in figure 9.26, showed improvements in performance as threads increased from 1 to 6. Performance improvements declined beyond 6 threads, even though 27 machines were targeted with the 10-second sleep job. After further testing, including manual execution of the same job using shell scripts instead of Stoker, it was determined that SSH authentication latency was increasing as the number of simultaneous SSH processes was increased. The root cause of this behavior was found to be serialization of the SSH authentication requests resulting from the use of a single Network File System (NFS) share for storing the SSH keys used for authentication. Since the single NFS server was also utilized for unrelated purposes, additional latency was added to the authentication process. As shown in figure 9.27, overhead as a percentage of total execution time increased with increasing

Figure 9.24: Stoker performance on a low-latency 16-node cluster when parallelizing a 10-second sleep operation

Figure 9.25: Comparative overheads of network and SSH with respect to the remote job being executed on a low-latency 16-node cluster. The normalized overhead is expressed as a percentage of the maximum observed overhead, which occurred when the number of threads was equal to 1. Relative overhead is expressed as the percentage of network and SSH overhead present in the total execution time of a 10-second sleep job.

Figure 9.26: Stoker performance on a high-latency group of 27 public laboratory workstations

parallelism, effectively negating the benefits of additional Stoker actor threads.

Figure 9.27: Comparative overheads of network and SSH with respect to the remote job being executed on a high-latency 27-node group of public workstations.

Chapter 10

Conclusions

Virtual Organization Clusters provide a mechanism by which individual Virtual Organizations may operate and administer virtual computational clusters that support the computing needs of VO-affiliated users. As re-illustrated in figure 10.1, VOCs are deployable in both transparent (non-participating) and participating contexts. In the transparent case, grid sites are able to separate the complexity of user-facing software stacks from the underlying services required to provide network connectivity and basic computational services. When VOs choose to make use of VOCs directly, pilot jobs are used to lease resources from physical sites, and the leased resources are joined via an overlay network to form a virtual cluster. Within this overlay cluster, the VO is able to make scheduling and resource allocation decisions for its users, who submit jobs to a dedicated VOC Computing Element.

Virtual Organization Clusters enable the execution of long-lived customized computational environments, which may span Grid sites to make best use of available physical fabric to provide cloud computing resources for user applications. Although VOCs will directly benefit users by improving job compatibility across sites, the implementation details of this new architecture will remain transparent to the users. Instead of forcing users to create and manage explicit leases, VOCs automatically adapt to changing resource demands, allowing users to focus on their own domain-specific research. Since these systems consist entirely of dynamically allocated virtual environments executing on remote grid infrastructure, Virtual Organization Clusters exist as self-provisioned clouds on the grid.

Conclusions regarding the utility and viability of VOCs are presented in section 10.1. Sec-

Figure 10.1: The architecture of Virtual Organization Clusters, revisited.

tion 10.2 describes the direct impacts of VOCs on grid systems, after which section 10.3 discusses the broader impacts of this research. Finally, section 10.4 provides a brief overview of related opportunities for future research. Information about the software packages included as attachments to this dissertation follows in the appendices.

10.1 Utility and Viability of Virtual Organization Clusters

As demonstrated by the prototype results (chapter 9), VOCs can execute High-Throughput Computing (HTC) grid jobs with reasonable overheads under 9%. Although High Performance Computing (HPC) applications that use latency-sensitive MPI library routines experience substantially greater overheads, VOCs still enable execution on physical sites that lack MPI support. This overhead represents a trade-off between performance and environment flexibility: whenever a job is directly compatible with the physical system environment at a grid site, the addition of the VOC layer will reduce execution performance for that job. However, **when the requirements of a job cannot be met by the environment present on the physical site, the addition of the VOC layer enables job execution when it would be otherwise impossible**, as demonstrated by the simulation results (chapter 8). The reduced execution performance in this case would be superior to the lack of execution that would otherwise occur.

The use of pilot jobs and IPOP overlay networking enables the provisioning of Virtual Organization Clusters with overlay scheduling, permitting each Virtual Organization to make resource allocation and job priority decisions within its private virtual environment (chapters 6 and 7). In this regard, VOCs are similar to pilot job frameworks used by High-Energy Physics (HEP) experiments (section 4.3.2). As demonstrated through tests using the prototype grid-connected system, the added overhead of the scheduling and network overlay is negligible for compute-bound grid jobs. Simulation results using actual trace data from the Enabling Grids for E-scienceE (EGEE) grid indicate that widespread VOC deployment on a grid system would not adversely affect the aggregate behavior of the grid, even though virtualization systems add execution overhead. VOCs reduce total aggregate queuing by making all jobs compatible with all sites composed of machines with the same instruction set architecture. Moreover, the virtual head node for each VOC creates a single submission point for all jobs affiliated with a particular VO, simplifying job submission for grid users.

10.2 Direct Impacts

Virtual Organization Clusters are a promising mechanism for delivering the benefits of grid virtualization systems, including environment customization, VO isolation, and legacy application support [57], to existing production grids without large-scale disruption. Through the use of overlay scheduling, individual VOs would be able to make resource allocation decisions for their members, allowing site and VO policies to be independent. Furthermore, the customization capabilities offered by virtualization would empower VOs to provide the software stacks required by their users, instead of forcing users to adapt applications to the available software environments installed by site administrators. As a result, existing computational grids could be made more useful for domain applications and more accessible to domain experts, without forcing users into system administration roles.

VOCs enable VOs to create and customize entire computational clusters for end users, which must be reachable through existing middleware. However, since the VOC Model does not proscribe other interfaces for submitting workloads to VOCs, a VO could provide a computational cloud directly to a set of users. Submission of jobs through an alternate interface to this cloud could bypass the existing grid middleware altogether, further simplifying the experience for end users. However, such a mechanism also would bypass the existing grid accounting and security mechanisms, in the same way such mechanisms can be bypassed by pilot job frameworks [143]. Thus, **the new paradigm of grid utilization provided by this architecture requires a new conceptualization of the grid as a provider of resource abstractions.** Instead of providing full computational services to end users, a grid system using VOCs could provide back-end computational fabric to VOs, which would then serve users directly.

10.3 Broader Impacts

The deployment of Virtual Organization Clusters on accessible grid systems would have broader impacts on the scientific community and society at large. As defined by the National Science Foundation [119], these impacts can be classified into five main areas: integration of research and education, broadening participation, enhancing infrastructure, broad dissemination, and benefits to society.

Virtual Organization Clusters can be used to integrate research with education by enabling

cluster construction in the classroom environment. Students would be able to construct computational clusters on top of grid systems using nothing more than a laptop with commodity Internet access and virtualization software. The dedicated hardware, cooling, power, networking, and other physical infrastructure required to construct a physical computational cluster, along with the software challenges inherent with its setup, are not present. Instead, students would be able to construct a VOC image within a few class periods, after which they would be able to instantiate virtual clusters on a commodity grid or a dedicated educational grid. Computational jobs spanning a wide range of disciplines, from computer science to the physical sciences, can be submitted by different student groups as part of an educational experience that demonstrates the benefits grid computing provides to diverse disciplines.

A second benefit to the modest requirements of VOCs hosted by cloud systems is that VOCs would be widely accessible to a large audience, regardless of the availability of cluster hardware or specialized networking at any given site. VOCs are therefore suitable for all educational institutions, from research universities to technical and community colleges, without requiring the availability of infrastructure, space, or dedicated support staff. This technology is ideal for broadening participation among institutions that might not otherwise have access to customized parallel computing resources, including rural institutions, institutions primarily serving underrepresented minority groups, and institutions participating in the EPSCoR program [118]. Moreover, the modest requirements of VOCs would enable participation at the K-12 level, allowing teachers and students to deploy custom services to support specific lesson plans. Use of VOC technologies at the preparatory level would have the benefit of exposing young people to the breadth and power of cloud computing, increasing awareness and interest in computational technologies as key enablers in both future scientific discoveries and future industries. VOCs enable the creation of internationally distributed systems by individuals or small groups, enabling creative opportunities that could appeal to both genders equally while crossing cultural, ethnic, and racial barriers.

VOCs would provide new mechanisms to enhance and extend current grid infrastructures, without requiring disruptive middleware and system replacements. Existing physical cluster systems could be multiplexed among different custom virtual clusters instead of using a shared multiprogramming model to multiplex hardware-based systems among users. The resulting customization would increase the utility of these systems to end users, since the environments used by the domain scientists would contain self-selected libraries and application software sets. Domain scientists would

thus be able to match the systems to their needs, instead of adapting their research applications to the capabilities provided by the system. Moreover, the lower technical entry barriers of VOCs would enable groups of domain scientists could form new, highly specialized Virtual Organizations that are focused on narrow problem areas. The resulting increased collaboration among such scientists would yield greater research productivity.

Both VOCs and the research behind them are designed for broad dissemination. VOCs themselves exist in the cloud and may be replicated across a wide range of systems, effectively becoming a new form of scientific communication. Dissemination of initial software and VOC research results has already occurred. The Stoker distributed management system [35] and SimVOC simulator [3] are freely available under open-source licenses. In addition, the following publications have resulted from VOC and related research:

- M. Murphy and S. Goasguen. "Virtual Organization Clusters: Self-Provisioned Clouds on the Grid." To appear, *Future Generation Computer Systems*.
- M. Murphy, L. Abraham, M. Fenn, and S. Goasguen. "Autonomic Clouds on the Grid." *Journal of Grid Computing*, 8(1): 1-18, 2009. [113]
- L. Stout, M. Murphy, and S. Goasguen. "Kestrel: An XMPP-based Framework for Many Task Computing Applications." MTAGS 2009. [149]
- M. Fenn, M. Murphy, and S. Goasguen. "A Study of a KVM-based Cluster for Grid Computing." ACMSE '09. [54]
- M. Murphy, B. Kagey, M. Fenn, and S. Goasguen. "Dynamic Provisioning of Virtual Organization Clusters." CCGrid '09 (21% acceptance rate). [117]
- M. Murphy, M. Fenn, L. Abraham, J. Canter, B. Sterrett, and S. Goasguen. "Distributed Management of Virtual Cluster Infrastructures." SMTPS '09. [114]
- M. Murphy, M. Fenn, and S. Goasguen. "Virtual Organization Clusters." PDP 2009 (42% acceptance rate). [115]
- M. Fenn, M. Murphy, J. Martin, and S. Goasguen. "An Evaluation of KVM for Use in Cloud Computing." ICVCI '08. [55]

- E. Harney, S. Goasguen, J. Martin, M. Murphy, and M. Westall. “The Efficacy of Live Virtual Machine Migrations over the Internet.” VTDC '07. [79]

Additionally, two journal articles and one additional conference proceeding, all directly related to Virtual Organization Clusters, are under review at the time of this writing.

If widely deployed, Virtual Organization Clusters would have profound positive impacts on society. By separating parallel computational environments from the underlying physical hardware, it would be possible to locate computational clusters in geographic areas that optimize the use of energy resources and minimize carbon production. Since VOCs can be moved between physical sites, it would be possible to respond to changing resource availability, enabling optimal use of renewable energy. For example, with future efficient photovoltaic technologies, it might be possible to migrate grid computing jobs world-wide on a constant basis, exclusively utilizing solar energy. With existing energy technologies, virtualized cluster systems can be located in areas with higher nuclear and renewable components to grid power, minimizing the fraction of computing performed using fossil fuels. Similarly, VOCs can be migrated geographically to take advantage of excess energy available during regional off-peak periods. In addition to these environmental benefits, the direct benefit of software environments customized for domain scientists will be better results from scientific modeling and prediction systems, which will in turn translate into discoveries that improve quality of life.

10.4 Future Work

Virtual Organization Clusters provide a new architecture for grid computing, which improves the usability of grid resources through the addition of virtualization technology. However, this new architecture and VOC research to date raise new questions that are not answered by the VOC Model itself. These issues may be loosely aggregated into four major areas: evaluation of the impact of VOCs, efficiency of VOC implementations, system security, and migration of the VOC concept to generic cloud systems that are not part of a grid architecture.

Although the VOC Model provides a foundation for reasoning about the performance impacts of VOCs, *a priori* evaluation of the impacts of VOC deployment in specific situations is still difficult, while large-scale VOC deployments are not likely to be embraced on federated grid systems without some performance assurances. Improved simulation systems and the creation of analytical models could provide a more accurate estimate of VOC impacts than the current model and imple-

mentation of SimVOC are able to produce. Better heuristics for mapping production grids would improve simulation fidelity, as would a more accurate implementation of simulated scheduler components. Although simulation is the preferred mechanism for evaluating experimental architectures in the grid community [25], a rigorous analytical model based on queuing theory also could provide insight into the effects of virtualization overhead and overlay scheduling on production grids.

Implementations of VOCs could benefit from further research into optimal VOC sizing and adjustment algorithms, which would balance sizing responsiveness with the costs of booting virtual machines. More efficient virtualization systems and mechanisms for bypassing the hypervisor when performing I/O operations would reduce the performance impacts of VOCs. An adaptive, fault-tolerant scheduling system that is robust to continuously changing environments and tuned for VOC use would enable more efficient scheduling of user jobs within the cloud. In terms of human factors, the design of best practices for VO system administration could optimize procedures for managing the software stacks made available to end users, thus improving user experience.

Another major area of research opportunity for grid systems utilizing the VOC architecture – and for cloud computing systems in general – is that of security, authorization, and data privacy. The addition of virtual machines to a production grid site may raise security concerns when insecure services, such as the Network File System, are made available on the local private network. Mechanisms for enforcing isolation of VMs will be needed to address the concerns of physical system administrators. Procedures and systems for validating VM images provided by the VOs will be needed, in order to ensure that a VM image is not compromised by an intermediate attacker. Infrastructure for ensuring the privacy of potentially sensitive end user data will be necessary before VOCs (or other cloud systems) could be used for sensitive activities.

Finally, an ultimate objective of research into self-provisioned clouds for scientific computing would be to move the virtual clusters to generic cloud computing hosts, eliminating reliance upon the grid entirely. Although scientific clusters constructed in this way would not be VOCs according to the definition presented in chapter 5, the same transparency and autonomic management principles could be used to create entity-dedicated clusters without incurring direct hardware expenses. These clouds would utilize readily available hosting services, such as Amazon EC2 [11], to provide compatible environments for large-scale simulations and other domain-specific applications, leading to improved productivity for end users and decreased costs for system stakeholders.

Appendices

Appendix A Electronic Attachments

Several software applications and data sets are provided as electronic attachments to this dissertation. These attachments include:

- A copy of the Stoker distributed management application, in Python source form
- A copy of the SimVOC simulation system, in Python source form
- Pre-processed trace data sets for the Enabling Grids for E-science (EGEE) production system, for use with SimVOC
- Operational data logs from the Clemson University Cyberinfrastructure Research Group's Furnace cluster with transparent VOCs deployed

License agreements for these attachments are provided in the next appendix. Both Stoker and SimVOC are released under the Apache License, version 2 (appendix B.3). EGEE trace data is provided under a license required by the Grid Observatory, which is listed in appendix B.1. Finally, the operational data logs are licensed under an MIT-style license, provided in appendix B.2.

Appendix B License Agreements

EGEE simulation data sets are provided by the Grid Observatory subject to the agreement in section B.1. Trace data from the Furnace cluster is supplied by the Cyberinfrastructure Research Group in the Clemson University School of Computing. Data are copyright 2008-2009 Clemson University and released under an MIT-style license (section B.2).

The Stoker and SimVOC applications that accompany this dissertation are released under the Apache License, version 2. A copy of this license is provided in section B.3.

B.1 Grid Observatory Data

The following conditions apply to data provided from the Grid Observatory:

All information, software and documentation are provided "as-is". The use of the material is restricted to scientific research. Significant use of the material for publication requires acknowledgement by the following citation: The datasets used in this work have been provided by the Grid Observatory (www.grid-observatory.org). The Grid Observatory is part of the EGEE-III EU project INFSO-RI-222667.

B.2 MIT License

Data sets provided by the Clemson University Cyberinfrastructure Research Group are released under the following license:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY

CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B.3 Apache License, version 2.0

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or

bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this

License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, includ-

ing, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF

ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Bibliography

- [1] EGEE Accounting Portal. [Online]. Available: <http://www3.egee.cesga.es>
- [2] Models in the research process. [Online]. Available: <http://www2.uiah.fi/projects/metodi/177.htm>
- [3] SimVOC. [Online]. Available: <http://code.google.com/p/simvoc>
- [4] SQLite. [Online]. Available: <http://www.sqlite.org/>
- [5] The PanDA Production and Distributed Analysis System. [Online]. Available: <https://twiki.cern.ch/twiki/bin/view/Atlas/PanDA>
- [6] The STAR Experiment. [Online]. Available: <http://drupal.star.bnl.gov/STAR/>
- [7] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu, "From virtualized resources to virtual computing grids: the In-VIGO system," *Future Generation Computer Systems*, vol. 21, no. 6, pp. 896–909, June 2005.
- [8] S. Adabala, A. Matsunaga, M. Tsugawa, R. Figueiredo, and J. Fortes, "Single sign-on in In-VIGO: Role-based access via delegation mechanisms using short-lived user identities," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 2004.
- [9] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, San Jose, CA, 2006.
- [10] J. M. Alonso, V. Hernandez, and G. Molto, "Towards on-demand ubiquitous metascheduling on computational grids," in *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '07)*, Naples, Italy, February 2007.
- [11] Amazon Web Services, "Amazon elastic compute cloud (amazon EC2)." [Online]. Available: <http://aws.amazon.com/ec2/>
- [12] Argonne National Laboratory, "Bcfg2." [Online]. Available: <http://trac.mcs.anl.gov/projects/bcfg2>
- [13] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Scheduling strategies for master-slave tasking on heterogeneous processor platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 4, pp. 319–330, April 2004.
- [14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

- [15] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms," in *3rd International Workshop on Parallel and Distributed Computing/Third International Symposium on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, 5–7 July 2004, pp. 296–302.
- [16] —, "Steady-state scheduling on heterogeneous clusters: why and how?" in *Proc. 18th International Parallel and Distributed Processing Symposium*, 26–30 April 2004, p. 171.
- [17] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert, "Centralized versus distributed schedulers for bag-of-tasks applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 5, pp. 698–709, May 2008.
- [18] R. Bradshaw, N. Desai, T. Freeman, and K. Keahey, "A scalable approach to deploying and managing appliances," in *TeraGrid 2007*, Madison, WI, June 2007.
- [19] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Communications of the ACM*, vol. 13, no. 4, pp. 238–242, 1970.
- [20] 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*, September 2004.
- [21] M. Burgess, "Recent developments in Cfengine," in *Unix.nl Conference*, Waardenburg, Netherlands, 2001.
- [22] —, "A tiny overview of Cfengine: Convergent management agent," in *1st International Workshop on Multi-Agent and Robotic Systems*, 2005.
- [23] —, "A control theory perspective on configuration management and Cfengine," *ACM SIGBED Review*, vol. 3, no. 2, pp. 12–16, April 2006.
- [24] M. Burgess and R. Ralston, "Distributed resource administration using Cfengine," *Software – Practice and Experience*, vol. 27, no. 9, pp. 1083–1101, 1997.
- [25] R. Buyya and M. Murshed, "GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Journal of Concurrency and Computation: Practice and Experience (CCPE)*, vol. 14, pp. 1175–1220, 2002.
- [26] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable cloud computing environments and the CloudSim toolkit: Challenges and opportunities," in *7th High Performance Computing and Simulation Conference (HPCS 2009)*, Leipzig, Germany, June 2009.
- [27] D. G. Cameron, A. P. Millar, C. Nicholson, R. Carvajal-Schiaffino, F. Zini, and K. Stockinger, "OptorSim: a simulation tool for scheduling and replica optimisation in data grids," in *Computing in High Energy and Nuclear Physics (CHEP 2004)*, Interlaken, Switzerland, September–October 2004.
- [28] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *ALS'00: Proceedings of the 4th annual Linux Showcase and Conference*, 2000.
- [29] H. Casanova, A. Legrand, and L. Marchal, "Scheduling Distributed Applications: the SimGrid Simulation Framework," in *Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, 2003.

- [30] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.
- [31] CERN, "The quattor system administration toolsuite," March 2009. [Online]. Available: <http://www.quattor.org>
- [32] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual clusters in a grid site manager," in *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, June 2003.
- [33] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. Silva, C. O. Barros, and C. Silveira, "Running bag-of-tasks applications on computational grids: The MyGrid approach," in *Proceedings of the 32nd International Conference on Parallel Processing*, 2003.
- [34] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005, pp. 273–286.
- [35] Clemson University Cyberinfrastructure Research Group, "Stoker home page." [Online]. Available: <http://code.google.com/p/cu-stoker>
- [36] L. B. Costa, L. Feitosa, E. Araújo, G. Mendes, R. Coelho, W. Cirne, and D. Fireman, "MyGrid – a complete solution for running bag-of-tasks applications," in *22nd Brazilian Symposium on Computer Networks*, Gramado, RS, Brazil, May 2004.
- [37] S. Dague, "System Installation Suite: Massive installation for Linux," in *Ottawa Linux Symposium*, 2002.
- [38] R. Davoli, "VDE: Virtual Distributed Ethernet," in *First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom 2005)*, Trento, Italy, February 2005.
- [39] N. Desai, "Bcfg2: A pay as you go approach to configuration complexity," in *2005 Australian Unix Users Group (AUUG2005)*, Sydney, Australia, October 2005.
- [40] N. Desai, R. Bradshaw, J. Hagedorn, and C. Lueninghoener, "Directing change using Bcfg2," in *Twentieth Large Install System Administration Conference (LISA XX)*, Washington, DC, December 2006.
- [41] N. Desai, R. Bradshaw, S. Matott, S. Bittner, S. Coghlan, R. Evard, C. Leunighoener, T. Leggett, J. Navarro, G. Rackow, C. Stacey, and T. Stacey, "A case study in configuration management tool deployment," in *Nineteenth Large Install System Administration Conference (LISA XIX)*, San Diego, CA, December 2005.
- [42] N. Desai, A. Lusk, R. Bradshaw, and R. Evard, "Bcfg: A configuration management tool for heterogenous environments," in *5th IEEE International Conference on Cluster Computing (CLUSTER '03)*, 2003.
- [43] Duke Systems, "COD: Cluster-On-Demand." [Online]. Available: <http://nicl.cod.cs.duke.edu/cod/index.html>
- [44] —, "Shirako home page." [Online]. Available: <http://nicl.cod.cs.duke.edu/cereus/shirako.html>
- [45] C. Dumitrescu and I. Foster, "GangSim: A simulator for grid scheduling studies," in *IEEE Cluster Computing and Grid 2005 (CCGrid '05)*, Cardiff, UK, May 2005.

- [46] EGEE, “Grid observatory.” [Online]. Available: <http://grid-observatory.org/>
- [47] W. Emeneker, “Dynamic virtual clustering,” Master’s thesis, Arizona State University, April 2007.
- [48] W. Emeneker, D. Jackson, J. Butikofer, and D. Stanzione, “Dynamic virtual clustering with Xen and Moab,” in *Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC)*, 2006.
- [49] W. Emeneker and D. Stanzione, “Increasing reliability through dynamic virtual clustering,” in *High Availability and Performance Computing Workshop (HAPCW 2006)*, 2006.
- [50] —, “Dynamic virtual clustering,” in *2007 IEEE International Conference on Cluster Computing*, 2007.
- [51] —, “Efficient virtual machine caching in dynamic virtual clusters,” in *Third Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS 2007)*, 2007.
- [52] Enabling Grids for E-science, “gLite.” [Online]. Available: <http://glite.web.cern.ch/glite/>
- [53] J. Enslow, P.H., “What is a ‘distributed’ data processing system?” *Computer*, vol. 11, no. 1, pp. 13–21, January 1978.
- [54] M. Fenn, M. A. Murphy, and S. Goasguen, “A study of a KVM-based cluster for grid computing,” in *47th ACM Southeast Conference (ACMSE ’09)*, Clemson, SC, March 2009.
- [55] M. Fenn, M. A. Murphy, J. Martin, and S. Goasguen, “An evaluation of KVM for use in cloud computing,” in *2nd International Conference on the Virtual Computing Initiative (ICVCI ’08)*, Durham, NC, May 2008.
- [56] R. Figueiredo, P. A. Dinda, and J. Fortes, “Resource virtualization renaissance,” *Computer*, vol. 38, no. 5, pp. 28–31, May 2005.
- [57] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes, “A case for grid computing on virtual machines,” in *23rd International Conference on Distributed Computing Systems*, 2003.
- [58] R. Flanery, A. Geist, B. Luethke, and S. L. Scott, “Cluster Command & Control (C3) tools suite,” in *3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, 2000.
- [59] J. A. B. Fortes, R. J. Figueiredo, and M. S. Lundstrom, “Virtual computing infrastructures for nanoelectronics simulation,” *Proceedings of the IEEE*, vol. 93, no. 10, pp. 1839–1847, October 2005.
- [60] I. Foster, “Globus toolkit version 4: Software for service-oriented systems,” in *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, 2005, pp. 2–13.
- [61] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, and X. Zhang, “Virtual clusters for grid communities,” in *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, Singapore, May 2006.
- [62] I. Foster, K. Keahey, C. Kesselman, E. Laure, M. Livny, S. Martin, M. Rynge, and G. Singh, “Embedding community-specific resource managers in general-purpose grid infrastructure,” Argonne National Laboratory, Tech. Rep. ANL/MCS-P1318-0106, January 2006.

- [63] I. Foster, "The grid: A new infrastructure for 21st century science," *Physics Today*, vol. 55, no. 2, pp. 42–47, February 2002.
- [64] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputing Applications*, vol. 11, no. 2, pp. 115–128, 1997.
- [65] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the Grid," June 2002. [Online]. Available: <http://www.globus.org/alliance/publications/papers/ogsa.pdf>
- [66] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations," *International Journal of Supercomputing Applications*, vol. 15, no. 3, pp. 200–222, 2001.
- [67] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," in *1st Workshop on Operating System and Architectural Support for the On Demand IT Infrastructure*, 2004.
- [68] T. Freeman and K. Keahey, "Flying low: Simple leases with Workspace Pilot," in *Euro-Par 2008*, Las Palmas de Gran Canaria, Spain, August 2008.
- [69] T. Freeman, K. Keahey, I. Foster, A. Rana, B. Sotomayor, and F. Wuerthwein, "Division of labor: Tools for growth and scalability of grids," in *4th International Conference on Service Oriented Computing (ICSOC 2006)*, Chicago, IL, December 2006.
- [70] K. Fujiwara and H. Casanova, "Speed and accuracy of network simulation in the SimGrid framework," in *Proceedings of the First International Workshop on Network Simulation Tools (NSTools)*, Nantes, France, October 2007.
- [71] F. Galan, D. Fernandez, J. Ruiz, O. Walid, and T. de Miguel, "Use of virtualization tools in computer network laboratories," in *Fifth International Conference on Information Technology Based Higher Education and Training (ITHET 2004)*, 2004.
- [72] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo, "IP over P2P: Enabling self-configuring virtual IP networks for grid computing," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
- [73] —, "WOW: Self-organizing wide area overlay networks of virtual workstations," in *15th IEEE International Symposium on High Performance Distributed Computing*, 2006.
- [74] A. Ganguly, D. Wolinsky, P. O. Boykin, and R. Figueiredo, "Decentralized dynamic host configuration in Wide-area Overlays of virtual Workstations," in *IEEE International Parallel and Distributed Processing Symposium IPDPS 2007*, 2007.
- [75] R. Geist, J. Martin, J. Westall, and V. Yalla, "Performance tuning of gigabit network interfaces," in *31st Annual International Conference of the Computer Measurement Group (CMG 2005)*, Orlando, FL, December 2005.
- [76] M. Gillespie. (2008) Best practices for paravirtualization enhancements from intel® virtualization technology: Ept and vt-d. [Online]. Available: <http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d/>
- [77] Globus Alliance, "Nimbus." [Online]. Available: <http://workspace.globus.org/>
- [78] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase, "Virtual machine hosting for networked clusters: Building the foundations for 'autonomic' orchestration," in *First International Workshop on Virtualization Technology in Distributed Computing (VTDC '06)*, Tampa, FL, November 2006.

- [79] E. Harney, S. Goasguen, J. Martin, M. Murphy, and M. Westall, "The efficacy of live virtual machine migrations over the Internet," in *Second International Workshop on Virtualization Technology in Distributed Computing (VTDC '07)*, Reno, NV, November 2007.
- [80] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES 2008)*, Indianapolis, IN, December 2008.
- [81] Intel Corporation. (2005, November) Intel Pentium 4 Processor 6xx Sequence and Intel Pentium 4 Processor Extreme Edition Datasheet. [Online]. Available: <http://www.intel.com/design/pentium4/datashts/306382.htm>
- [82] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi, "Self-recharging virtual currency," in *Third Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, Philadelphia, PA, August 2005.
- [83] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum, "Sharing network resources with brokered leases," in *USENIX Technical Conference*, Boston, MA, June 2006.
- [84] X. Jiang and D. Xu, "VIOLIN: Virtual Internetworking on OverLay INfrastructure," Purdue University, Tech. Rep. CSD TR 03-027, July 2003.
- [85] W. M. Jones, L. W. Pang, D. Stanzione, and W. B. Ligon, "Job communication characterization and its impact on meta-scheduling co-allocated jobs in a mini-grid," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, April 2004.
- [86] W. M. Jones, L. W. Pang, and D. Stanzione, "Computational mini-grid research at clemson university," Clemson University, Clemson, SC, Documentation PARL-2002-009, November 2002.
- [87] M. Jung and R. Gogolok, "Bcfg2 - konfigurationsmanagement für heterogene umgebungen," in *German Unix User Group's Frühjahrsfachgespräch 2007*, March 2007.
- [88] M. J. Katz, P. M. Papadopoulos, and G. Bruno, "Leveraging standard core technologies to programmatically build Linux cluster appliances," in *IEEE International Conference on Cluster Computing (Cluster 2002)*, April 2002.
- [89] K. Keahey, K. Doering, and I. Foster, "From sandbox to playground: Dynamic virtual environments in the Grid," in *5th International Workshop on Grid Computing (Grid 2004)*, Pittsburgh, PA, November 2004.
- [90] K. Keahey, I. Foster, T. Freeman, and X. Zhang, "Virtual Workspaces: Achieving quality of service and quality of life in the Grid," *Scientific Programming*, vol. 13, no. 4, pp. 265–275, 2005.
- [91] K. Keahey and T. Freeman, "Contextualization: Providing one-click virtual clusters," in *4th IEEE International Conference on e-Science*, Indianapolis, IN, December 2008.
- [92] K. Keahey, T. Freeman, J. Lauret, and D. Olson, "Virtual Workspaces for scientific applications," in *Scientific Discovery through Advanced Computing*, Boston, MA, June 2007.
- [93] K. Keahey, M. Ripeanu, and K. Doering, "Dynamic creation and management of runtime environments in the Grid," in *Workshop on Designing and Building Web Services, Global Grid Forum 9*, Chicago, IL, October 2003.
- [94] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron, "Virtual workspaces in the Grid," in *11th International Euro-Par Conference*, Lisbon, Portugal, September 2005.

- [95] K. Keahey, J. Chase, and I. Foster, “Virtual playgrounds: managing virtual resources in the grid,” in *20th International Symposium Parallel and Distributed Processing (IPDPS '06)*, 2006.
- [96] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [97] O. Khalid, R. J. Anthony, P. Nilsson, K. Keahey, M. Schulz, K. Parrot, and M. Petridis, “Enabling and optimizing pilot jobs using Xen based virtual machines for the HPC grid applications,” in *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing (VTDC '09)*, Barcelona, Spain, June 2009.
- [98] D. Klusacek, L. Matyska, and H. Rudova, “Alea – grid scheduling simulation environment,” in *Seventh International Conference on Parallel Processing and Applied Mathematics*, Gdansk, Poland, September 2007.
- [99] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, “VMPlants: Providing and managing virtual machine execution environments for grid computing,” in *ACM/IEEE Conference on Supercomputing*, 2004.
- [100] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz, “Grid scheduling simulations with GSSIM,” in *Third International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS '07)*, Hsinchu, Taiwan, December 2007.
- [101] J. Lee, K. Jeong, H. Lee, I. Lee, S. Lee, D. Park, C. Lee, and W. Yang, “RISE: A grid-based self-configuring and self-healing remote system image management environment,” in *Second IEEE International Conference on e-Science and Grid Computing (e-Science '06)*, December 2006.
- [102] A. Legrand, O. Beaumont, L. Marchal, and Y. Robert, “Master slave scheduling on heterogeneous star-shaped platforms with limited memory,” in *IEEE International Conference on Cluster Computing*, September 2004.
- [103] A. Legrand, L. Marchal, and H. Casanova, “Scheduling distributed applications: the SimGrid simulation framework,” in *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)*, Tokyo, Japan, May 2003.
- [104] Linux Virtualization Wiki, “TechOverview,” January 2007. [Online]. Available: <http://virt.kernelnewbies.org/TechOverview>
- [105] —, “TechComparison,” April 2008. [Online]. Available: <http://virt.kernelnewbies.org/TechComparison>
- [106] J. Liu, W. Huang, B. Abali, and D. K. Panda, “High performance VMM-bypass I/O in virtual machines,” in *USENIX Annual Technical Conference*, June 2006.
- [107] P. Luszczyk, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi, “The HPC Challenge (HPCC) benchmark suite,” in *Supercomputing '06*, 2006.
- [108] M. Matsuda, T. Kudoh, and Y. Ishikawa, “Evaluation of MPI implementations on grid-connected clusters using an emulated WAN environment,” in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid03)*, 2003.
- [109] A. Matsunaga, M. Tsugawa, M. Zhao, L. Zhu, V. Sanjeevan, S. Adabala, R. Figueiredo, H. Lam, and J. A. Fortes, “On the use of virtualization and service technologies to enable grid-computing,” in *11th International Euro-Par Conference*, August 2005.

- [110] A. M. Matsunaga, M. O. Tsugawa, S. Adabala, R. J. Figueiredo, H. Lam, and J. A. B. Fortes, "Science gateways made easy: the In-VIGO approach," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 905–919, April 2007.
- [111] C. Morey, "Use of Cfengine for deploying LCG/gLite middle-ware," in *International Conference on Computing in High Energy and Nuclear Physics*, 2007.
- [112] J. Mugler, T. Naughton, and S. L. Scott, "OSCAR meta-package system," in *19th International Symposium on High Performance Computing Systems and Applications*, May 2005.
- [113] M. A. Murphy, L. Abraham, M. Fenn, and S. Goasguen, "Autonomic clouds on the grid," *Journal of Grid Computing*, vol. 8, no. 1, pp. 1–18, March 2010.
- [114] M. A. Murphy, M. Fenn, L. Abraham, J. A. Canter, B. T. Sterrett, and S. Goasguen, "Distributed management of virtual cluster infrastructures," in *Fifth International Workshop on System Management Techniques, Processes, and Services (SMTPS '09)*, Rome, Italy, May 2009.
- [115] M. A. Murphy, M. Fenn, and S. Goasguen, "Virtual Organization Clusters," in *17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, Weimar, Germany, February 2009.
- [116] M. A. Murphy and H. K. Harton, "Evaluation of local networking in a Lustre-enabled virtualization cluster," Clemson University, Tech. Rep. CU-CILAB-2007-1, 2007.
- [117] M. A. Murphy, B. Kagey, M. Fenn, and S. Goasguen, "Dynamic provisioning of Virtual Organization Clusters," in *9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '09)*, Shanghai, China, May 2009.
- [118] National Science Foundation, "Experimental program to stimulate competitive research (EPSCoR)." [Online]. Available: <http://www.nsf.gov/div/index.jsp?org=EPSC>
- [119] —, "Merit review broader impacts criterion: Representative activities." [Online]. Available: <http://www.nsf.gov/pubs/gpg/broaderimpacts.pdf>
- [120] H. Nishimura, N. Maruyama, and S. Matsuoka, "Virtual clusters on the fly - fast, scalable, and flexible installation," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007)*, May 2007.
- [121] Open Science Grid, "Open Science Grid." [Online]. Available: <http://www.opensciencegrid.org>
- [122] P. M. Papadopoulos, M. J. Katz, and G. Bruno, "NPACI Rocks: Tools and techniques for easily deploying manageable Linux clusters," in *IEEE International Conference on Cluster Computing (Cluster 2001)*, October 2001.
- [123] P. M. Papadopoulos, C. A. Papadopoulos, M. J. Katz, W. J. Link, and G. Bruno, "Configuring large high-performance clusters at lightspeed: A case study," in *Clusters and Computational Grids for Scientific Computing 2002*, December 2002.
- [124] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.
- [125] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, I. Foster, R. Gardner, M. Wilde, A. Blatecky, J. McGee, and R. Quick, "The Open Science Grid: Status and architecture," in *International Conference on Computing in High Energy and Nuclear Physics (CHEP '07)*, 2007.

- [126] B. Quetier, V. Neri, and F. Cappello, "Selecting a virtualization system for Grid/P2P large scale emulation," in *Workshop on Experimental Grid Testbeds for the Assessment of Large-scale Distributed Applications and Tools (EXPGRID'06)*, Paris, France, June 2006.
- [127] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, and J. Chase, "Toward a doctrine of containment: Grid hosting with adaptive resource control," in *19th Annual Supercomputing Conference (SC '06)*, Tampa, FL, November 2006.
- [128] A. S. Rana, F. Würthwein, K. Keahey, T. Freeman, A. Vaniachine, D. Malon, E. May, S. Martin, R. Gardner, B. Sotomayor, B. Holzman, R. Pordes, D. Skow, T. Wenaus, R. Popescu, J. Shank, E. Laure, I. Bird, M. Schulz, M. Litmaath, S. Campana, D. Smith, B. Blumenfeld, K. De, M. Vranicar, P. Smith, and S. Wang, "An Edge Services Framework (ESF) for EGEE, LCG, and OSG," in *Computing in High Energy and Nuclear Physics (CHEP '06)*, Mumbai, India, February 2006.
- [129] Red Hat, "Kernel-based Virtual Machine." [Online]. Available: <http://www.linux-kvm.org>
- [130] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *USENIX Security Symposium*, 2000.
- [131] Rocks Development Team, "Rocks clusters," Website. [Online]. Available: <http://www.rocksclusters.org>
- [132] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, May 2005.
- [133] P. Ruth, X. Jiang, D. Xu, and S. Goasguen, "Virtual distributed environments in a shared infrastructure," *Computer*, vol. 38, no. 5, pp. 63–69, 2005.
- [134] P. Ruth, P. McGachey, and D. Xu, "VioCluster: Virtualization for dynamic computational domains," in *IEEE International Conference on Cluster Computing*, Boston, MA, September 2005.
- [135] P. Ruth, J. Rhee, D. 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.
- [136] —, "Enabling autonomic adaptation of virtual computational environments in a shared distributed infrastructure," Purdue University, Tech. Rep. CSD TR 06-004, January 2006.
- [137] F. D. Sacerdoti, S. Chandra, and K. Bhatia, "Grid systems deployment and management using ROCKS," in *IEEE International Conference on Cluster Computing*, September 2004.
- [138] San Diego Supercomputing Center. Inca: User level grid monitoring. [Online]. Available: <http://inca.sdsc.edu/>
- [139] J. M. Schopf. (2001, July) Ten actions when superscheduling. Open Grid Forum Scheduling Working Group. [Online]. Available: <http://www.ogf.org/documents/GFD.4.pdf>
- [140] J. Schönwälder, V. Marinov, and M. Burgess, "Integrating cfengine and scli: Managing network devices like host systems," in *Network Operations and Management Symposium (NOMS 2008)*, Salvador, Bahia, Brazil, April 2008.
- [141] S. Seetharaman and K. Murthy, "Test optimization using software virtualization," *IEEE Software*, vol. 23, no. 5, pp. 66–69, 2006.

- [142] I. Sfiligoi, "Making science in the Grid world: using glideins to maximize scientific output," in *IEEE Nuclear Science Symposium Conference Record (NSS '07)*, Honolulu, HI, 2007.
- [143] I. Sfiligoi, G. Quinn, C. Green, and G. Thain, "Pilot job accounting and auditing in Open Science Grid," in *9th IEEE/ACM International Conference on Grid Computing (Grid '08)*, 2008.
- [144] S. C. Simms, G. G. Pike, and D. Balog, "Wide area filesystem performance using Lustre on the TeraGrid," in *TeraGrid 2007 Conference*, Madison, WI, June 2007.
- [145] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The Micro-Grid: A scientific tool for modeling computational grids," in *IEEE Supercomputing (SC2000)*, Dallas, TX, November 2000.
- [146] B. Sotomayor, K. Keahey, and I. Foster, "Overhead matters: A model for virtual resource management," in *2nd International Workshop on Virtualization Technologies in Distributed Computing (VTDC '06)*, Tampa, FL, November 2006.
- [147] B. Sotomayor, K. Keahey, I. Foster, and T. Freeman, "Enabling cost-effective resource leases with virtual machines," in *HPDC 2007 Hot Topics*, Monterey Bay, CA, June 2007.
- [148] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *17th International Symposium on High Performance Distributed Computing (HPDC 2008)*, 2008.
- [149] L. Stout, M. A. Murphy, and S. Goasguen, "Kestrel: An XMPP-based framework for many task computing applications," in *2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS 2009)*, Portland, OR, November 2009.
- [150] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *USENIX 2001*, 2001.
- [151] A. Sulistio, U. Cibej, S. Venugopal, B. Robic, and R. Buyya, "A toolkit for modelling and simulating data grids: An extension to GridSim," *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 20, no. 13, pp. 1591–1609, September 2008.
- [152] A. Sulistio, C. S. Yeo, and R. Buyya, "A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools," *International Journal of Software: Practice and Experience*, vol. 34, no. 7, pp. 653–673, June 2004.
- [153] A. I. Sundararaj and P. A. Dinda, "Towards virtual networks for virtual machine grid computing," in *Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.
- [154] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima, "Overview of a performance evaluation system for global computing scheduling algorithms," in *Eighth International Symposium on High Performance Distributed Computing*, 1999.
- [155] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.
- [156] Tentakel Development Team, "Tentakel homepage." [Online]. Available: <http://tentakel.biskalar.de/>
- [157] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees, "DIRAC: A Scalable Lightweight Architecture for High Throughput Computing," in *Fifth IEEE/ACM International Workshop on Grid Computing (GRID '04)*, Pittsburgh, PA, November 2004.

- [158] M. Tsugawa and J. A. B. Fortes, "A virtual network (ViNe) architecture for grid computing," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
- [159] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, vol. 38, no. 5, pp. 48–56, May 2005.
- [160] M. Varian, "VM and the VM community, past present, and future," in *SHARE 89*, 1997. [Online]. Available: <http://www.princeton.edu/~melinda/25paper.pdf>
- [161] S. J. Vaughan-Nichols, "New approach to virtualization is a lightweight," *Computer*, vol. 39, no. 11, pp. 12–14, 2006.
- [162] P. Velho and A. Legrand, "Accuracy study and improvement of network simulation in the SimGrid framework," in *2nd International Conference on Simulation Tools and Technique (SIMUTools'09)*, Rome, Italy, March 2009.
- [163] A. Weiss, "Computing in the clouds," *netWorker*, vol. 11, no. 4, pp. 16–25, 2007.
- [164] D. Wolinsky, A. Agrawal, P. O. Boykin, J. Davis, A. Ganguly, V. Paramygin, P. Sheng, and R. Figueiredo, "On the design of virtual machine sandboxes for distributed computing in Wide-area Overlays of virtual Workstations," in *First International Workshop on Virtualization Technology in Distributed Computing*, 2006.
- [165] H. Xia, H. Dail, H. Casanova, and A. Chien, "The MicroGrid: Using online simulation to predict application performance in diverse grid network environments," in *Challenges of Large Applications in Distributed Environments (CLADE'04)*, Honolulu, Hawaii, June 2004.
- [166] D. Xu, P. Ruth, J. Rhee, R. Kennell, and S. Goasguen, "Autonomic adaptation of virtual distributed environments in a multi-domain infrastructure," in *15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06)*, Paris, France, June 2006.
- [167] J. Xu, S. Adabala, and J. A. B. Fortes, "Towards autonomic virtual applications in the In-VIGO system," in *2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, Seattle, WA, June 2005.
- [168] S. Yamasaki, N. Maruyama, and S. Matsuoka, "Model-based resource selection for efficient virtual cluster deployment," in *3rd International Workshop on Virtualization Technology in Distributed Computing (VTDC '07)*, Reno, NV, 2007.
- [169] W. Yu, R. Noronha, S. Liang, and D. K. Panda, "Benefits of high speed interconnects to cluster file systems: a case study with Lustre," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
- [170] Zenoss, Inc. Zenoss: Commercial Open-Source Application, Systems, and Network Monitoring. [Online]. Available: <http://www.zenoss.com/>
- [171] J. Zhang, M. Yousif, R. Carpenter, and R. J. Figueiredo, "Application resource demand phase analysis and prediction in support of dynamic resource provisioning," in *Fourth International Conference on Autonomic Computing (ICAC '07)*, 2007.
- [172] X. Zhang, K. Keahey, I. Foster, and T. Freeman, "Virtual cluster workspaces for grid applications," Argonne National Laboratory, Tech. Rep. ANL/MCS-P1246-0405, April 2005.
- [173] M. Zhao, V. Chadha, and R. J. Figueiredo, "Supporting application-tailored grid file system sessions with WSRF-based services," in *14th IEEE International Symposium on High Performance Distributed Computing*, July 2005.

- [174] M. Zhao, J. Xu, and R. J. Figueiredo, "Towards autonomic grid data management with virtualized distributed file systems," in *IEEE International Conference on Autonomic Computing (ICAC '06)*, June 2006.
- [175] M. Zhao, J. Zhang, and R. Figueiredo, "Distributed file system support for virtual machines in grid computing," in *13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, 2004.