Large-Scale Elastic Computing with Virtual Machines

by

Paul D. Marshall

B.A., Augustana College, 2006

M.S., University of Colorado at Boulder, 2010

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2013

This thesis entitled:
Large-Scale Elastic Computing with Virtual Machines
written by Paul D. Marshall
has been approved for the Department of Computer Science

Henry M. Tufo

Dr. Qin Lv

Dr. Shivakant Mishra

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Marshall, Paul D. (Ph.D., Computer Science)

Large-Scale Elastic Computing with Virtual Machines

Thesis directed by Professor Henry M. Tufo

Computational resources experience dynamic load because demand is not constant. As a result, resource providers (RPs) must estimate the appropriate amount of resources to purchase in order to best meet variable demand, possibly resulting in under-utilized resources during periods of low demand and over-utilized resources during periods of high demand. With the relatively recent introduction of infrastructure-as-a-service (IaaS) clouds, which lease virtual machines (VMs) on-demand, RPs can both deploy private clouds (offering users a new type of resource lease) and outsource appropriate workload processing to external clouds when needed. To match resource deployments with demand, RPs can create elastic environments that span private and public clouds, expanding as demand increases and shrinking as demand decreases. However, elastic environments do not provide the necessary mechanisms and techniques required to extend cluster resources automatically and efficiently for scientific workflows. Instead, they must be managed manually, which is inefficient and limits scalability, or use product-specific solutions that are not open or extensible. Furthermore, IaaS toolkits typically provide high-cost, on-demand leases that are not required by all workflow paradigms.

This dissertation presents a flexible cloud architecture and its implementation, consisting of preemptible and preset leases and an elastic environment that is capable of outsourcing cluster demand to IaaS clouds. The architecture allows RPs to adapt efficiently and cost effectively to variable demand for common scientific workflow patterns. Preemptible and preset leases are a new low-cost lease for IaaS clouds that are amenable for volunteer computing or high-throughput computing workloads. For implementation, these leases are included in the open source Nimbus IaaS toolkit and deploy preemptible VMs on idle resources, allowing RPs to increase utilization of under-utilized IaaS clouds. To adjust to variable demand, the elastic environment uses resource provisioning policies that provision and relinquish IaaS instances, outsourcing to external clouds when demand is high. The resource provisioning policies balance conflicting objectives between users and administrators, such as minimizing job queued time and the cost of the deployment. For evaluation, a complete end-to-end elastic environment is developed and used to process a large bioinformatics workload across multiple clouds.

Dedication

To my wife, Megan.

Acknowledgements

I would like to thank my advisor, Henry Tufo, and my committee for their support during this thesis. Henry provided encouragement and advice for my work at the University of Colorado and the National Center for Atmospheric Research. Kate Keahey at Argonne National Laboratory provided the initial suggestion for this work, elastically extending site resources with infrastructure clouds, as well as continued guidance and feedback on my work over the past four years.

I would also like to thank my colleagues at the University of Colorado and the National Center for Atmospheric Research for their expertise and feedback on this project. I'd specifically like to thank Guy Cobb, Jason Cope, Dmitry Duplyakin, Brad Henke, Michael Oberg, Theron Voran, and Matthew Woitaszek. The Nimbus team at Argonne National Laboratory, including Patrick Armstrong, John Bresnahan, Tim Freeman, David LaBissoniere, and Pierre Riteau offered extensive support for my work with the Nimbus toolkit and deployments on FutureGrid as well as detailed advice on many components of this project. Rob Knight, and members of his lab at the University of Colorado, specifically Greg Caporaso, Antonio Gonzalez Pena, and Daniel McDonald were helpful answering my questions about the QIIME toolkit and providing advice about this work, especially as it related to bioinformatics workflows.

Finally, I would like to thank my friends and family, especially my parents David and Elizabeth, for their support over the past six years. I am especially indebted to my undergraduate advisor, Daniel Swets *(1964 - 2011)*, who introduced me to the scientific research community and encouraged me to pursue my interests in computer science research.

Contents

Tables

Figures

Glossary

AMI (Amazon Machine Image) - Amazon-specific VM image format that is pre-configured with an operating system and applications for Amazon EC2.

AWS (Amazon Web Services) - Amazon's suite of cloud computing products, including EC2 and S3.

Batch-queue resource manager - queues user-submitted jobs for batch processing where a scheduler determines the order in which to execute jobs as well as the available resources to execute them on.

boto - Python library to interface with Amazon Web Services.

Chef - an open source systems integration framework developed by Opscode that can be used to automate common system administration tasks such as installing packages and configuring resources.

Cloud Computing - services provided over a network (e.g., the Internet) that allow users to provision infrastructure resources, deploy applications on scalable platforms, or access and run standalone applications hosted on remote servers.

Cyberinfrastructure (CI) - environments and infrastructure to support computing and data storage and analysis.

DAV (data and visualization analysis environment) - large resources designed specifically for data analysis and visualization purposes, e.g., consisting of significant amounts of RAM and graphics processing units.

EBS (Elastic Block Store) - Amazon's for-pay block storage service, allowing users to mount block volumes in an instance with persistent data storage.

EC2 (Elastic Compute Cloud) - Amazon's IaaS for-pay compute service that offers VMs on-demand from data centers across the world.

Elastic Environment - an environment capable of supporting compute and storage resources that adapt dynamically to demand, growing as demand increases and shrinking as demand decreases.

FutureGrid - a large distributed CI designed for computer science research on the use of grid and cloud computing tools and services. It consists of both HPC and IaaS resources distributed across multiple sites.

Globus Toolkit - a collection of software tools and services to enable Grid computing across distributed resources, providing services such as job management, single sign-on, and file transfer.

GlusterFS - an open source, reliable, and distributed file system capable of aggregating disk and memory resources across distributed resources into a single name space.

Grid Computing - collection of distributed resources that allow user communities to create virtual organizations and distributed workflows across the resources.

GridFTP - a Grid-enabled file transfer protocol capable of transferring datasets in parallel by opening multiple streams.

HPC (High-Performance Computing) - large high-performance compute and storage resources, such as supercomputers and clusters, with low latency interconnects used to solve large challenges with demanding resource requirements.

HTC (high-throughput computing) - computing environments that can deliver large amounts of compute capabilities over significant amounts of time.

HTCondor - a resource manager specifically designed for high-throughput computing workloads that are executed across large distributed resources.

IaaS (Infrastructure-as-a-Service) - cloud computing paradigm that provides virtual infrastructure resources, such as virtual machines, on-demand.

libcloud - Python library to interface with a variety of IaaS cloud providers, including AWS, Nimbus, and OpenStack.

MPI (Message Passing Interface) - a message-passing standard used to develop portable message-passing applications for distributed computing environments.

NFS (Network File System) - distributed file system protocol that allows clients to access a central file system server over a network.

Nimbus - a community project developing open source tools for scientific computing in the cloud, including both infrastructure-layer toolkits and platform-layer services.

OGE (Oracle Grid Engine) - a cluster resource manager (formerly known as Sun Grid Engine) capable of scheduling user jobs across distributed resources.

On-demand resource lease - a request for resources that is either fulfilled or rejected by the RP in near-interactive time.

OOI (Ocean Observatories Initiative) - an NSF funded initiative designed to build an advanced science-driven sensor and computing infrastructure to improve mankind's understanding of the ocean and its processes.

OpenStack - an open source IaaS toolkit, originally developed by Rackspace and NASA, which can be used to deploy private or public clouds.

PaaS (Platform-as-a-Service) - cloud computing paradigm that provides platform services, such as programming language execution environments and databases, which are hosted in the cloud and scaled by the cloud provider.

Preemptible and preset resource lease - a resource lease that configured by an RP administrator and is deployed on idle resources and can be preempted immediately by on-demand resource leases.

QIIME (Quantitative Insights Into Microbial Ecology) - an open source toolkit, consisting of native components as well as community tools, for comparison and analysis of microbial communities.

Resource lease - a request for a specific amount of resources, e.g., a dual-core 2.93 GHz Xeon VM with 8 GB of RAM and 20 GB of disk space. The lease may optionally include a defined time period for the resource.

RP (resource provider) - an organization that makes computing and storage resources, such as an IaaS cloud or supercomputer, available to one or more user communities.

S3 (Simple Storage Service) - Amazon's for-pay object storage service, allowing users to store an unlimited number of objects distributed across multiple regions.

SaaS (Software-as-a-Service) - cloud computing paradigm that provides individual applications, hosted in the cloud that can be accessed via a web-based interface.

Spot instances - Amazon's IaaS instances that allow users to bid on unused capacity for possibly a reduced cost. Spot instances can be terminated by Amazon at anytime when the spot price is raised, allowing Amazon to reclaim infrastructure for on-demand resource leases.

TeraGrid - a large Grid computing infrastructure comprising 11 sites and integrating distributed supercomputers and high-performance clusters with the Globus toolkit. It operated from 2004 to 2011, when it was replaced by XSEDE.

Torque - an open source resource manager for HPC supercomputers and clusters.

VM (Virtual Machine) - a virtual duplicate of a real machine, e.g. Intel's x86, that allows custom software stacks to be installed and multiple operating systems to share the same hardware.

Volunteer computing - a computing environment where the resource owners donate their systems to other purposes, such as executing tasks to process datasets without immediate deadlines.

XSEDE (eXtreme Science and Engineering Discovery Environment) - follow-on to the TeraGrid that includes 17 sites integrating distributed supercomputers and high-performance clusters with the Globus toolkit.

XtreemFS - an open source, fault-tolerant, and distributed file system capable of replicating data across multiple storage servers.

Chapter 1

Introduction

Physical resources, such as high-performance computing (HPC) clusters or data analysis and visualization environments (DAVs), provide static capacity but experience dynamic demand. Resource providers (RPs) must estimate the appropriate amount of static physical resources to purchase in order to meet variable user demand. As a result, resources may be under-utilized during periods of low demand and over-utilized during periods of high demand. Demand typically bursts when users are actively debugging their applications and running full-system simulations to generate results for a deadline, such as a paper deadline or an urgent computing deadline (e.g., tracking the path of a hurricane). Demand typically decreases between deadlines and when users focus on gathering data or writing code. Furthermore, many problems, from climate modeling to gene sequencing or financial risk analysis, require vast computational and storage resources to perform relatively simple analyses. With even the slightest increase in the level of detail these problems scale rapidly. A single simulation, submitted by a single user, has the potential to grow far beyond today's largest resources. Large-scale distributed infrastructures expand beyond the resources owned by a single organization and provide a suitable platform for dynamic multi-user communities experimenting with these challenges. At the same time, a key challenge for RPs is balancing periods of low demand and high demand in order to most effectively serve their users. Ideally, RPs would prefer to operate their resources at a relatively high level of utilization while simultaneously responding to user requests almost immediately, temporarily offloading demand to external resources, if necessary and cost effective.

Therefore, to serve users and RPs effectively, large-scale distributed infrastructures should meet the following criteria:

- **Elastic**: The infrastructure should adapt dynamically to demand, expanding as demand increases and contracting as demand decreases, minimizing wasted resources and costs.
- **Customizable**: Users should be able to customize the software stack of the infrastructure, from the operating system upward, allowing them to deploy complex software stacks with unique requirements.
- **Scalable**: The infrastructure should provide sufficient resources to address large-scale and resource-intensive challenges.
- **High utilization**: The infrastructure should be capable of maintaining high utilization and still serve users with a variety of needs, including users with immediate deadlines as well as users with volunteer computing workloads who may not have immediate deadlines.

A variety of large-scale distributed platforms and resource managers exist for dynamic multi-user communities with large-scale challenges. HTCondor is one example [117]; it was developed in the late 1980s and is now a well-established and feature-rich resource manager. HTCondor provides a generic environment for processing high-throughput computing (HTC) workloads by harvesting cycles from idle workstations or scheduling a pool of compute resources by distributing tasks to these systems. Similar to HTCondor, the "at home" paradigm of computing, as implemented by a system such as BOINC [37], provides a large-scale cycle-scavenging platform that has been successfully leveraged for specific applications, including protein folding [85] and the search for extraterrestrial intelligence [38], and has been deployed on a world-wide scale. PlanetLab [50] is a large-scale platform that is specifically focused on the deployment of network applications and services on a global scale. The Grid [63] is a generic platform for integrating and sharing distributed compute and storage resources, especially those dispersed across multiple organizations. However, the Grid has a number of constraints that have limited its adoption. In particular, it assumes that control over the remote resource is with the site and not the user and, therefore, users with complex software stacks may encounter significant difficulties when deploying and configuring their software. The Grid also lacks the necessary capabilities to expand and contract dynamically as demand fluctuates, confining workflows and applications to a specific set of standalone resources.

Recently, cloud computing has emerged as a new computing paradigm that allows users to access resources through well-defined application programming interfaces (APIs) and, thus, scale applications immediately based on demand and outsource excess demand to external resources when needed. Software-as-a-service (SaaS) clouds provide Web-based access to individual applications and Platform-as-a-service (PaaS) clouds provide a cloud-based platform for applications that use the platform's specific services. Unfortunately, users must develop custom applications for specific SaaS and PaaS clouds. However, infrastructure-as-a-service (IaaS) clouds [39] provide generic infrastructure resources, typically in the form of virtual machines (VMs), as an on-demand service, allowing users to deploy existing complex software stacks and create elastic environments that adapt dynamically to demand. Elastic environments expand as demand increases in order to respond quickly to user requests and shrink as demand decreases, limiting the compute cycles and storage wasted by the deployment. Large cloud providers also offer sufficiently scalable IaaS infrastructures for demanding applications and workflows. Finally, on-demand provisioning provided by IaaS clouds is ideal for users with deadlines [90], for example, a paper deadline or an urgent computing deadline where sufficient resources must be allocated for immediate use [43], [52].

From an RP perspective, RPs may choose to deploy elastic environments that either extend existing physical resources, such as an HPC cluster, with IaaS resources or deploy standalone elastic environments in the cloud, outsourcing demand when needed. Currently, elastic environments must be managed manually, which is inefficient and limits scalability, or use product-specific solutions that often only interface with a single cloud provider or application. RPs may also choose to deploy private IaaS clouds in order to offer users traditional IaaS on-demand leases. However, IaaS toolkits that are used to deploy private clouds typically only provide high-cost, on-demand leases where specific requests for resources are either accepted or rejected in near-interactive time.

IaaS RPs must significantly over-provision their infrastructure to ensure on-demand availability, requiring them to pay a high price for operating a resource with low utilization, or reject a large proportion of user requests, which effectively eliminates the on-demand nature of the IaaS cloud. At the same time, not all users require truly on-demand access to resources, such as those designed for recoverable systems where interruptions in service are expected.

This Ph.D. dissertation presents a flexible cloud architecture and implementation that improves underlying IaaS cloud infrastructure utilization and adapts resource deployments for platform-layer user environments to match variable demand. In particular, this dissertation presents a new, low-cost, preemptible and preset resource lease for IaaS clouds that deploys preemptible VMs on idle IaaS resources. Such preemptible and preset leases allow RPs to increase utilization of under-utilized IaaS clouds by offering a new resource lease amenable to users who don't require immediate access to resources, such as those with volunteer computing workloads that can leverage publicly-owned resources [37] or HTC workloads [88]. The open source Nimbus IaaS toolkit [18] is extended to support preemptible and preset leases. This dissertation also presents a large-scale elastic environment that adjusts to demand using flexible resource provisioning policies to balance user and administrator requirements. The elastic model extends existing services and resources with private and public IaaS instances, outsourcing excessive demand to IaaS clouds. The environment adds and removes instances based on two major factors: 1) the current demand, for instance, the number of jobs in a work queue, and 2) the requirements specified by the resource administrator, such as whether he or she wishes to minimize cost or responsiveness of the deployment. For implementation, the environment leverages existing auto-scale services, such as Phantom [80], to integrate with a wide-variety of infrastructure clouds.

Other scheduling algorithms and policies exist to balance demand, utilization, and cost in static environments, including supercomputers, dynamic HTC or volunteer computing environments, and large-scale distributed and shared Grid environments. IaaS clouds, however, introduce three new properties that dictate the need to investigate and develop new algorithms, policies, and models to balance demand and cost in IaaS clouds. First, many IaaS cloud providers charge for use, often with actual currency (e.g., U.S. dollars). Second, IaaS clouds provision resources on-demand, either immediately granting access to the resource or rejecting the request. Though IaaS cloud environments may contain a variable number of instances, the size of these environments may be dictated by the user. This differs from dynamic volunteer computing environments where membership of workers is influenced by outside factors, such as the availability of idle desktop workstations. And third, the underlying IaaS infrastructure needs to be significantly over-provisioned in order to guarantee on-demand access to resources the majority of the time as opposed to traditional clusters or supercomputers, which are typically provisioned to ensure relatively high utilization.

A primary goal of this work is to assist the scientific community by developing a reactive and scalable implementation of the flexible cloud architecture that is open source and easy-to-use. An open source implementation eliminates the costs associated with acquiring the software, allowing anyone to deploy and improve it. An easy to use solution allows scientists and researchers to leverage the implementation without investing significant time or money into software engineering or system administration tasks. Therefore, the implementation will possess the following attributes:

- **Open and extensible**: Users and system administrators must be able to adapt the implementation to suit their needs. For example, they should be able to enable preemptible VMs as needed, customize the elastic environment to match their workload patterns, and extend the software to support additional cloud providers.

- **Reactive and scalable**: The system should react quickly, efficiently, and appropriately to changes in demand in order to maximize job throughput and minimize costs. Additionally, the large-scale elastic environment needs to be scalable in order to meet the challenges faced by demanding scientific problems.

- **Easy to use**: The elastic environment needs to be transparent, or nearly transparent, to users and require minimal effort for the system administrator. The elastic environment should contain a set of parameters to tune the elastic deployment in order to match the resource deployment with site-specific workload patterns and requirements.

Other open source tools and research projects provide a subset of this elastic environment functionality. As an example, OpenNebula [115] includes a set of cloud drivers to manage VMs on local cloud infrastructure as well as public clouds. However, OpenNebula's cloud drivers are tightly integrated into OpenNebula and, therefore, both local and remote resource deployments must be managed directly by OpenNebula. The cloud drivers also do not integrate with existing scientific workload managers. No open source, extensible, and sufficiently scalable elastic environment managers exist to extend local HPC clusters with IaaS instances based on demand or deploy large-scale elastic environments in private or public clouds. Additionally, no open source IaaS toolkits include preemptible and preset leases, allowing private IaaS deployments to increase utilization by efficiently supporting workloads designed for recoverable systems. Finally, resource provisioning policies are needed to balance user- and administrator-defined requirements in elastic environments that use IaaS clouds. This dissertation presents scalable elastic computing services and policies capable of deploying, contextualizing, and managing large-scale elastic environments.

1.1 Thesis Statement and Intellectual Contributions

Thesis Statement:

Cloud computing resources can provide an elastic and high-performance environment for large scientific challenges where resource deployments closely match user and workload requirements.

This Ph.D. dissertation presents the development and evaluation of the flexible cloud architecture and its implementation. This dissertation addresses several areas of work not addressed in the existing scientific and cloud computing communities. The flexible cloud architecture provides a comprehensive system that adapts appropriately and dynamically to demand at both the cloud infrastructure and user environment layers. At the cloud infrastructure layer, the flexible cloud architecture provides preemptible and preset resource leases that allow RPs to offer cycles that would have otherwise been idle to other processes, such as volunteer computing tasks, which typically do not have deadlines or depend on other tasks in the set. At the user environment layer, the flexible cloud architecture provides an elastic environment that can span local physical resources, private clouds, and public clouds, outsourcing demand as needed. The elastic environment also uses resource provisioning policies that respond appropriately to demand by provisioning or relinquishing cloud instances. The policies consider both user and administrator requirements and attempt to minimize costs.

Several intellectual and engineering contributions are derived from this research. The intellectual contributions include:

- Design of a flexible cloud architecture that adapts efficiently to variable demand, for both the user environment and the underlying infrastructure, and supports common scientific workflow patterns and characteristics.

- Formulation of resource provisioning policies for elastic environments that balance user- and administrator-defined requirements, minimizing costs.

- Extend an existing scheduling algorithm for Grid environments to support IaaS environments that uses a genetic algorithm to balance conflicting objectives.

- Evaluation and analysis of the elastic cloud model, including both compute and data aspects. The model consists of a standalone cluster, a private cloud with limited scalability, and an "infinitely" scalable for-pay public cloud provider.

The engineering contributions include:

- Development and evaluation of preemptible and preset leases for the open source Nimbus IaaS toolkit, allowing users to leverage a new type of resource lease for workloads without immediate deadlines, such as volunteer or HTC workloads.

- Development and evaluation of a scalable multi-cloud elastic environment that adapts to variable demand, outsourcing work when needed using infrastructure clouds.

- Development of an elastic cloud simulator, which is used in this dissertation to develop and analyze the resource provisioning policies and elastic cloud model.
- Evaluation of the elastic environment with a bioinformatics use case, demonstrating the end-to-end capabilities of the elastic environment for workloads with significant data requirements.

1.2 Organization

This remainder of this dissertation is organized as follows. Chapter 2 presents an overview of existing technologies and platforms for large-scale distributed computing environments, including tightly-coupled distributed operating systems and more loosely-coupled large-scale distributed infrastructures. Chapter 3 highlights the main challenges of flexible computing environments with infrastructure clouds and discusses related work. Chapter 4 presents the flexible cloud architecture that allows both the user environment and underlying infrastructure to adapt efficiently to variable demand. Chapter 5 presents the preemptible and preset leases for IaaS clouds, allowing RPs to offer a new type of lease to users, as well as the evaluation of these leases. Chapter 6 describes the scalable and multi-cloud elastic environment and its implementation and evaluation. Chapter 7 presents the resource provisioning policies and elastic cloud model as well as the simulation-based evaluation. Chapter 8 demonstrates the end-to-end capabilities of the multi-cloud elastic environment with a bioinformatics use case. Chapter 9 concludes this dissertation with a summary of major contributions and a discussion of future work, including the general applicability of this work to new domains.

Chapter 2

Background

Large-scale infrastructures must be able to handle dynamic, and sometimes significant, demand in multi-user environments as well as provide customizable user environments due to diverse user needs. Some applications require specific services or libraries while other applications may require different underlying operating systems. Systems capable of addressing these two challenges must allow individual users to customize their own software environments and also provide significant compute and storage resources when needed. Existing static resource environments are unable to adapt to variable demand and typically only provide users with a shared software environment, forcing all users to conform to a single software stack. Forcing users into such environments is unrealistic for complicated workflows with diverse applications and requirements. Additionally, a single static resource may have trouble meeting user deadlines when experiencing heavy load and leveraging distributed resources provides extra compute and storage resources to help meet demand [90]. However, if these distributed environments integrate a set of heterogeneous and physical resources (as is typically the case in large-scale Grid environments), the challenges of deploying a complex software stack are only magnified.

This chapter reviews existing technologies and platforms for large-scale distributed computing environments. In particular, the chapter provides an overview of virtualization, a fundamental technology used by IaaS clouds that provides customizable user environments, before discussing tightly-coupled distributed operating system approaches and loosely-coupled large-scale distributed environments, such as PlanetLab [50] and HTCondor [117]. The chapter ends with a discussion of

Grid computing environments and an overview of current cloud computing paradigms, including software-as-a-service (SaaS), platform-as-a-service (PaaS), and infrastructure-as-a-service (IaaS).

2.1 Virtualization

Virtualization, and in particular the use of system virtual machines, presents users with a generic environment where they can tailor any level of the software stack to meet the needs of their applications. System-level virtualization solutions implement an additional layer of software between the hardware and operating system known as a Virtual Machine Monitor (VMM) or hypervisor. The hypervisor presents a virtual implementation of a real machine to the operating systems running on top of it. System-level virtualization was clearly demonstrated in the IBM VM/370 time-sharing system [53] in the 1970s. The VM/370 was heavily influenced by the modular design of Massachusetts Institute of Technology's (MIT) Compatible Time-Sharing System (CTSS). In particular, the VM/370 system borrowed components from CTSS's compatibility and protection mechanisms, which allowed multiple programs to be run on a single system without modification. The VM/370 system contains three major components: the Control Program (CP), the Conversational Monitor System (CMS), and the Remote Spooling and Communications Subsystem (RSCS). The CP functions as a hypervisor, allocating and managing the hardware amongst a group of VMs, where each VM runs independently from the other VMs. The CMS is a single user operating system, running as a VM that provides an environment for a user to interact with the machine, for example, allowing users to enter commands or create and manage files. The RSCS provides network support for VMs running on a VM/370 system with the appropriate network equipment, allowing VMs to send or receive files to other systems.

Developed in the 1990s, SimOS [107] was designed to model computer hardware for researchers and computer system designers, allowing them to model complete computer systems in varying levels of detail for testing and verification purposes. SimOS was able to simulate the underlying hardware, including the CPU, memory management unit, and I/O devices, in various levels of detail. It used binary translation to dynamically convert a particular code into new code, simulating its execution under different hardware configurations. Disco [45] and Cellular Disco [68] moved beyond the simulation emphasis of SimOS and allowed multiple operating systems to run simultaneously on the same system, increasing utilization of the resource. Disco and Cellular Disco supported unmodified commodity operating systems, such as Windows NT and several Unix variants, in a virtual environment by emulating instructions, the memory management unit, and the trap architecture of the processor. Cellular Disco added support for hardware fault containment as well as more advanced resource management mechanisms and policies. The concepts developed in Disco and Cellular Disco were commercialized by VMware, Inc. in 1998.

In 2001 the University of Cambridge introduced Xen [42] as an open source x86 virtualization platform, allowing anyone to deploy virtual environments for their x86-based applications. The original implementation of Xen used a paravirtualization approach as opposed to binary translation in order to support virtualization on architectures that don't support it natively. Xen's paravirtualization approach requires modifying the guest operating system to run in ring 1 instead of ring 0 on x86 processors; this prevents the guest operating system from executing privileged instructions, such as installing a new page table. Instead, all privileged instructions must be executed by the Xen hypervisor running in ring 0. Xen uses shared memory and asynchronous buffer-descriptor rings to transfer I/O to and from guest operating systems.

Recently, virtualization-specific enhancements have been added to the x86 architecture, including Intel-VT [16] and AMD-V [4]. Intel and AMD have also added additional virtualization support in Nehalem and Barcelona, including Extended Page Tables [100] for more efficient guest OS to physical address mapping. Even though virtualization provides numerous benefits to users, such as a customizable software stack and isolation from other VMs, it still requires users and administrators to have direct access to physical resources to setup, manage, and allocate VMs. Virtualization solutions also require manual management of instance deployments, requiring administrators to monitor usage and explicitly add or remove instances from their virtual environment.

2.2 Distributed Operating Systems

Virtualization solutions may oversubscribe a single machine with multiple operating systems running simultaneously, distributed operating systems typically do just the opposite. Distributed operating systems combine multiple systems together to form the appearance of a single system. The main advantage of such a system is that it hides the complexities of the distributed environment from the user and yet the user still receives many of the advantages of a distributed environment, including the ability to leverage additional resources that dynamically join or leave the system at any time. While there have been numerous research projects that have successfully built and deployed distributed operating systems, none of them have gained substantial traction beyond the research phase for large-scale scientific deployments.

An early distributed system developed at Digital Equipment Corporation (DEC), called VAX-clusters [83], connected a group of VAX computers with a message-oriented interconnect so that they appeared as a single system. Standard processors and a general-purpose operating system were used to build VAXclusters. VAXcluster computers are all located within a single security domain, can access shared services, including disk and print services, and run a copy of the VAX/VMS distributed operating system. VAXclusters provide a low-overhead communication architecture and a high speed message-oriented interconnect. Nodes may join or leave a VAXcluster at any time, but a node can only be a member of one VAXcluster at a time. Cluster managers prevent nodes from partitioning the cluster into two or more clusters through a quorum-voting scheme. In this scheme, individual nodes contribute a vote and a cluster manager tallies the votes. If the total is equal to or greater than the quorum then the cluster continues operations. If the total falls below the quorum, activity is suspended. A distributed lock manager is used to control access to shared resources, which may only serve a single cluster at a given time.

Plan 9 [104] is a distributed operating system developed at Bell Laboratories, heavily influenced by the design of Unix. Plan 9 presents a more loosely coupled environment from VAXclusters. Systems are not required to fully exist within the same security domain, thus, home users may connect remotely to utilize services provided by Plan 9. Services in Plan 9 are grouped into a variety of categories, including CPU servers that concentrate compute power, file servers that provide storage, and terminals that give users a dedicated interface to the system. The CPU servers and terminals run the same kernel so applications execute on the local terminal or on a centrally located CPU server. Plan 9 resources, including disks, terminals, or processes exist in a single namespace and appear to the user as file systems, allowing uniform access. Plan 9 utilizes a file caching mechanism, based on file version numbers, to amortize the effects of relatively slow interconnects between remote terminals and the central file servers. In recent years, Plan 9 has seen renewed development with a collaboration between Sandia National Lab (SNL) and IBM, specifically to provide a distributed programming and execution environment on a large supercomputer, the IBM Blue Gene, with hundreds of thousands of cores [97].

Amoeba [99] is a distributed operating system with similarities to Plan 9. The Amoeba architecture contains a processor pool for extra compute power, diskless workstations for user interaction, and specialized file and database servers. These components are interconnected and all systems run the same kernel, allowing processes to run anywhere in the distributed environment. Amoeba is an object-based system where individual components are represented as objects on which operations are performed. User processes interact with objects through Remote Procedure Calls (RPC) to request operations. The actual user environment in Amoeba is an emulation of Unix. This way, users can run many familiar Unix editors, compilers, and tools in Amoeba. However, Amoeba is not Unix and does not provide 100% compatibility. Amoeba can also be deployed in a wide-area network environment. In this case, Amoeba deployments are divided into domains where each domain is a collection of local area networks and broadcast messages are restricted to individual domains.

There are many other distributed operating systems, including the V distributed system [49], Sprite [103], and CHORUS [108] that share similar characteristics with those discussed here. While all of these distributed operating systems abstract away many of the underlying complexities of distributed environments, they all face the same fundamental challenges that have thus far prevented them from evolving into production-level, large-scale environments capable of tackling massive challenges, such as those in the scientific community. Because distributed operating systems bring multiple systems together as a single system, the underlying communication protocols can cause performance problems as the system scales. Furthermore, due to the research-oriented nature of many distributed operating systems, few production-level scientific applications have been ported to run in a distributed operating system environment. Perhaps a notable exception is the SGI UV [27], which is a commercial system capable of scaling a single OS over 2048 cores and up to 64 TB of RAM using a non-uniform memory access (NUMA) interconnect. However, because the SGI UV uses proprietary SGI technologies and environments, it has seen limited adoption within the broader scientific community.

2.3 Large-Scale Distributed Systems

An alternative to the tight interconnectedness provided by distributed operating systems is a more loosely coupled approach to integrate large distributed resources. These loosely coupled environments share similar characteristics: first, they manage a distributed set of resources often on a large scale. Second, they dynamically integrate and organize the resources. Lastly, they present an interface to the user for interacting with the environment. Unlike distributed operating systems, these systems make no attempt to present a single environment to the user. Users are expected to develop their applications and workflows specifically for a set of distributed resources using message-passing systems, such as a message passing interface (MPI) application [70].

PlanetLab [50] is an exceptionally successful network overlay testbed for hosting and testing large-scale deployments of distributed network services. Users deploy services in slices across the overlay network. A slice is a horizontal cut of resources that includes processing, memory, storage, and networking resources across individual PlanetLab nodes. Virtual machines provide isolation between slices. PlanetLab nodes host multiple virtual machines, each running their own services. PlanetLab's virtual machines are based on Linux Vservers, which provide the appearance of multiple independent servers running on the same node. Vservers implement isolation at the system call interface and enforce isolation through the processes security context as well as checks on the UID/GID when a process attempts to access a privileged resource. However, users aren't given root on their slices, instead they are given "pseudo" root on the system, which provides them with the ability to access many of the services that require root. Because isolation is implemented at the system call interface, a single kernel is shared along with numerous user-level daemons, possibly allowing one malicious Vserver to exploit a kernel-level bug and allow it to gain control over all of the Vservers. The Linux CPU scheduler provides a degree of fairness between processes, however, it does not provide guarantees that are required for resource reservations. This potentially allows processes to abuse their share of the resource. PlanetLab places a cap on outgoing traffic to ensure that all Vservers share the network fairly.

To support large-scale deployments, PlanetLab must be able to discover available resources, dynamically create slices on those resources, and then launch services within those slices. PlanetLab manages its large-scale infrastructure with three solutions: a boot monitor that allows extensive remote manipulation of the machine, a boot server that downloads instructions to a newly booted machine, and a process running on the node to get and apply runtime updates of non-kernel packages. PlanetLab also maintains the expectation that nodes are dynamic, that is, they may occasionally be unavailable. Thus, services running in PlanetLab are expected to gracefully handle such dynamism. This is an expectation that PlanetLab management operations can exploit: when required, small groups of nodes can be taken offline without giving consideration to the services running on the nodes.

The most obvious limitation with PlanetLab given its primary focus on network services, is that it is not a generic platform for running any application; its architecture reflects this in numerous ways. For example, Vservers share the same kernel, creating a potential security hole, as noted earlier, and limiting applications to services provided by specific kernel versions. Vservers also provide minimal control over resource usage and isolation. The network link is a perfect example of this: PlanetLab expects that services won't abuse the network and instead of rate-limiting network traffic, excessive users are simply disabled. Many scientific applications have large I/O demands where it may be favorable to limit I/O throughput over time as opposed to disabling the resource for the particular application.

HTCondor [117] is a distributed job scheduler capable of harnessing idle workstations or clusters for compute cycles. Some of the most prominent features of HTCondor include a distributed job submission mechanism, a framework for matching jobs with resources, check pointing and migration, job suspend and resume, and support for heterogeneous platforms. Though HTCondor is a generic platform for managing jobs on large distributed infrastructures, it isn't ideal for many types of applications. HTCondor was designed to be a cycle-scavenger, thus it is typically deployed on systems that are, first and foremost, intended for other purposes, such as office or lab workstations. Many HTCondor deployments don't offer any resource availability guarantees, nor any level of performance guarantees. On these dynamic deployments it is not only possible but it is expected that individual executions will be interrupted temporarily or need to be terminated and moved to new resources as individual machines become active for their original purpose and unavailable in the resource pool. Thus, it can be especially problematic for certain applications, such as MPI applications that do not natively contain checkpoint-restart capabilities, if they are not executed on resources with dedicated availability.

The combination of user-defined policies with an intricate matchmaking process that describes virtually every combination of job requirement and resource capability, in addition to a vast set of features offered by HTCondor, creates a usability issue. Deploying and maintaining a feature-rich HTCondor pool is a time-consuming undertaking. Also, users must learn how to express their job and resource requirements through ClassAds, which can be complicated for typical scientific workflows. Finally, HTCondor lacks a defined payment model for resource usage. Resource owners can only specify priorities, e.g., a computer science department operating a HTCondor pool on their department machines can specify that jobs from users within the department have a higher priority than jobs from users outside the department.

At home computing [37] is similar in many respects to HTCondor. It attempts to utilize the capabilities of distributed workstations. However, at home deployments are often application specific, such as SETI@home [38] or Folding@home [85]. Although these deployments have been quite successful for specific applications, they require large investments in software engineering time to tailor individual applications for large distributed deployments. Additionally, these deployments are primarily suited to serial tasks because of their widely distributed nature.

2.4 Grid Computing

The Grid [63] is a large-scale distributed platform that provides a foundation for integrating and sharing distributed resources, creating an environment for hosting and executing applications. Two major examples from the scientific computing communities include the TeraGrid [48], now XSEDE [31], and the Open Science Grid (OSG) [105]. The Grid bridges organizational boundaries for individuals or groups of users at participating sites, allowing them to form a single virtual organization (VO). The Grid uses a set of libraries and middleware to implement compute and storage management, a security infrastructure, and monitoring and reporting services. The security infrastructure allows for single sign-on so that once a user is logged into the grid he or she can utilize all of the authorized resources without signing into each resource separately.

In 2001, Foster et al. [63] highlighted the Grid problem as “flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources.” The authors present an open Grid architecture that highlights and categorizes the protocols, services, application programmer interfaces (APIs), and software development kits (SDKs) that contribute toward solving the Grid problem. The Globus Toolkit [62] is an implementation of much of this proposed architecture, for example, GRAM [54] provides resource management and GridFTP [34] and RFT [35] provide data management.

Grid computing enables direct access to physical resources, and due to the limited overhead introduced by Grid middleware this makes it an ideal candidate for large and demanding scientific applications. However, because of the direct access to remote physical resources, Grid computing assumes that control of the resource is with the remote site. In practice, this has proved to be a limiting assumption for some users [89], [33] due to the heterogeneity of hardware, operating systems, and libraries used across Grid resources. For example, Agarwal et al. [33] note these difficulties for deploying high energy physics applications because of application dependencies that require specific OS versions and libraries. For security purposes Grid users are not given root on remote resources, thus it can be difficult or impossible to deploy custom software stacks required by a complex scientific workflow. Applications must also be ported to the different hardware platforms on the Grid, which can be a long and painful process depending on the complexity and requirements of the applications. Finally, because Grid environments are typically implemented directly on physical resources, provisioning of those resources is not dynamic or on demand.

As an example, we attempted to deploy the Quantitative Insights Into Microbial Ecology (QIIME) open source bioinformatics toolkit [47] on the TeraGrid. The TeraGrid's resources included both x86 systems, such as Ranger at the Texas Advanced Computing Center (TACC), and PowerPC systems, including Big Red at Indiana University (IU) and Frost at the National Center for Atmospheric Research (NCAR). QIIME integrates over 30 tools, including uclust [59], BLAST [36], and FastTree [106], among others, for analyzing and comparing complex and diverse microbial communities. These applications are developed by a variety of research groups and organizations using many different programming languages and libraries. The QIIME toolkit itself consists of Python scripts and modules, FastTree is a single C file, BLAST is an x86 binary, and another application is written in Haskell while yet another is a Java application. Deploying the complete QIIME software stack for even a single user on a variety of architectures, operated by different organizations with different user and system policies was a time consuming and frustrating task. And while the QIIME toolkit consists of over 30 applications, it also depends on a large number of common system dependencies that were not always available on the systems (e.g., via common system package managers such as Debian Apt [10]). Deploying QIIME for multiple users exacerbated these problems and required working with individual system administrators at each site to help download and install dependencies. As systems retired and new systems were deployed on the TeraGrid, the entire toolkit had to be reinstalled from scratch.

Furthermore, running QIIME's embarrassingly parallel workloads on supercomputers that are architected primarily for large tightly-coupled parallel jobs required significant modifications. Frost, a Blue Gene/L at NCAR, only reserves job partitions in groups of 64 cores, meaning that a single-core serial job still reserved 64 cores. The Blue Gene/L also only provides 256 MB of RAM per core and performs a full reboot of its nodes between jobs. Therefore, to run a large number of embarrassingly parallel QIIME tasks efficiently, we used the Blue Gene/L's high-throughput computing (HTC) mode [51] and modified its launcher to run multiple QIIME tasks between reboots [96]. While the Blue Gene/L provided substantial cycles, its memory constraints limited the size of datasets that could be processed for certain applications; it also required substantial investment in software engineering time and root access to the system to make the HTC modifications.

2.5 Cloud Computing Paradigms

In 2005 Globus Virtual Workspaces [81] demonstrated a mechanism to provide isolated user environments, which leveraged system-level virtualization, across Grid computing resources for different VOs. These isolated environments allowed VOs to customize and manage their own execution environments across Grid computing resources. Globus Virtual Workspaces eventually evolved into the open source IaaS Nimbus toolkit [18], which offers on-demand resource provisioning of VMs via API-based access. This allows users to integrate resource provisioning directly into their workflows and deploy complex software stacks in contained virtual images. In addition to IaaS, PaaS and SaaS cloud computing paradigms also provide mechanisms for developing and deploying applications in the cloud. PaaS cloud computing runs applications that use the platform's specific services, such as its programming execution environments and database services. Unlike IaaS clouds, users cannot easily deploy existing applications on PaaS clouds without customizing them for the specific platform. SaaS clouds are even more constrained than IaaS and PaaS clouds. Users must build and deploy individual applications and deploy them as standalone services that are typically accessed via a Web-based interface.

IaaS clouds, such as those provided by FutureGrid (Table 2.1), typically consist of front-end servers that monitor and control a large pool of back-end compute servers, allowing on-demand

Table 2.1: Nimbus, OpenStack, and Eucalyptus IaaS Clouds available on FutureGrid in March 2013.

IaaS Clouds				
Name	Location	IaaS Software	Virtualization	Total cores
Hotel	UChicago	Nimbus	Xen	328
Sierra	SDSC	Nimbus	Xen	144
Sierra	SDSC	OpenStack	Xen	24
Foxtrot	UFL	Nimbus	Xen	176
Alamo	TACC	Nimbus	KVM	96
India	IU	OpenStack	Xen	224
India	IU	Eucalyptus	Xen	240

provisioning of the resources with VMs. Typically, a user-defined resource lease specifies that a VM image be deployed across a specific amount of resources (e.g., based on the number of CPU cores or the amount of RAM required). If the cloud is able to fulfill the request, the VM image is transferred from an image repository to the back-end compute resources and then booted. If the cloud is unable to service the request (e.g., there are not enough available resources), it is rejected. Because of virtualization's isolation properties, users are given complete control over the software stack inside their VMs. Users can elect to deploy any operating system and configure the image to perform any function. In 2006, Amazon released the Elastic Compute Cloud (EC2) [3], a commercial IaaS cloud offering that provided Linux instances on-demand for a minimum of \$0.10 per hour (in 2013 the same instance now costs \$0.06). Amazon EC2 is perhaps the most prominent example of a commercial IaaS cloud platform with EC2 data centers around the world.

PaaS clouds provide a specific environment and are more restrictive than generic IaaS clouds. Users are limited to the programming languages and services supported by the environment. For example, a user cannot deploy a custom database within a PaaS environment, instead applications must support the PaaS database solution. A primary advantage of PaaS clouds is that the underlying services are typically designed to scale as needed. Thus, a user is not responsible for increasing database capabilities under heavy demand, instead, the PaaS provider ensures that the service is capable of handling the demand. Both Microsoft's Azure platform [30] and Google's Compute Engine [14] began as PaaS clouds, however, they have both recently added IaaS capabilities, allowing users to deploy generic Linux instances. Other examples of PaaS clouds include Engine Yard [11] and Heroku [15], both which initially supported Ruby applications. However, Heroku has expanded to include support for Java, Node.js, Scala, Clojure, and Python applications. Engine Yard has evolved to include support for JRuby, PHP, and Node.js.

SaaS clouds provide specific applications to users where the application itself is hosted and maintained in the cloud. User data associated with the applications is also hosted in the cloud. These applications are usually Web-based, commercial offerings include Google's Gmail, Google Docs, or Yahoo! Mail. Science gateways, such as the Asteroseismic Modeling Portal (AMP) [119] at the National Center for Atmospheric Research (NCAR) and Galaxy [67] at Penn State, also offer SaaS platforms, providing users with easy-to-use Web-based interfaces for running scientific applications. SaaS applications are clearly the most constrained cloud environments, limiting users to a specific application. SaaS applications also have high software engineering costs because the entire environment and deployment must be customized for a specific application. However, because SaaS applications are web-based, they also typically provide cross-platform and easy-to-use graphical interfaces for users.

2.6 Discussion

Designing a large-scale computing infrastructure that adapts elastically to demand, provides customizable user environments, and maintains high utilization while still serving the needs of diverse user communities requires building on underlying components that support elasticity and are customizable. Existing infrastructures and approaches, such as distributed operating systems and Grid computing environments, lack the necessary components required to create scalable elastic environments for users with complex software stacks and demanding workloads. However, with the introduction of IaaS clouds that use virtualization to support isolated user environments and a variety of resource provisioning leases (e.g., on-demand), it is now possible to create scalable elastic computing environments that adapt dynamically to demand and support diverse user communities with demanding workloads.

Chapter 3

Architecting a Flexible Cloud Computing Environment

IaaS clouds provide an ideal foundation for flexible computing environments that adapt efficiently to demand, both for users and resource providers (RPs). On-demand resource provisioning allows users or RPs to create elastic environments where resource deployments expand or contract based on immediate demand, using private or public IaaS cloud providers. Elastic environments can be used to extend local site resources with cloud resources or they can be deployed entirely in the cloud. As an example, a local batch-queue cluster may be extended with IaaS resources, creating an elastic environment that integrates static local cluster resources with dynamic and virtual cloud resources. Alternatively, an RP may instead choose to deploy an elastic cluster entirely in the cloud for their user communities, completely outsourcing demand when needed. Figure 3.1 is a model of a elastic environment with both static local site resources and cloud resources. From an RP's perspective, IaaS clouds can be employed to offer different forms of resource leases to users, which help to optimize resource utilization and service offerings for the RP.

To create an efficient flexible cloud environment a number of infrastructure- and platform-layer challenges must be addressed as well as the monetary aspects of such deployments. Infrastructure challenges include both low-level technical challenges, such as minimizing virtualization overhead, and higher-level availability and utilization challenges. For example, IaaS cloud providers must over-provision their infrastructure, operating a large amount of idle resources, to ensure on-demand availability. This may result in poor resource utilization, which is particularly problematic for scientific communities that typically achieve high utilization because of batch-queue resource

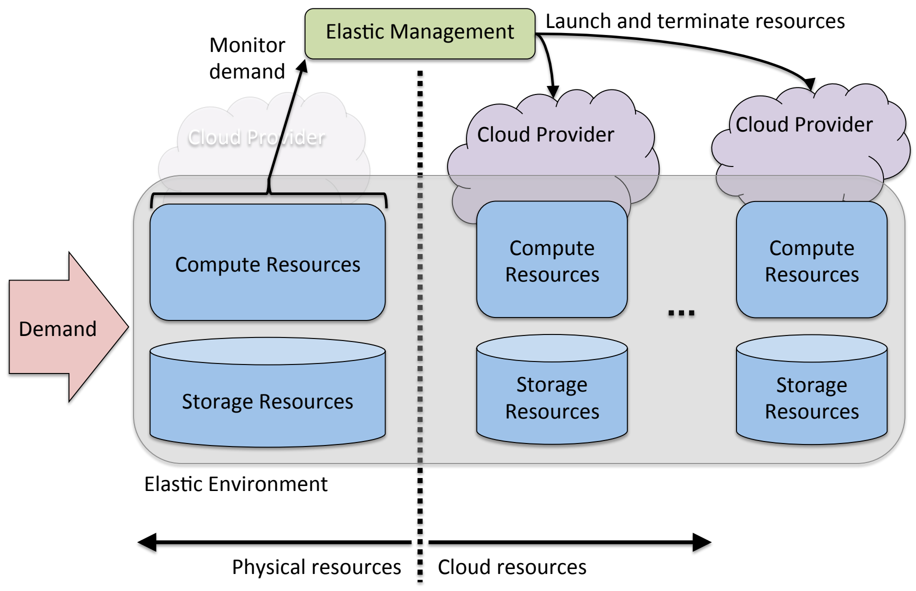

Figure 3.1: A model of an elastic environment. The elastic environment may extend physical resources with IaaS resources or be deployed entirely in the cloud.

managers. User-level elastic environments must also address platform-layer challenges, such as developing algorithms and policies to balance user and administrator requirements when provisioning elastic environments. Workflows and applications need to be adapted to support IaaS APIs and integrate into dynamic computing environments. Finally, the monetary aspects of such deployments must be addressed as well. Specifically, the environment should not only adapt based on demand, but should also consider the financial impact of for-pay IaaS deployments. This chapter provides an overview of these challenges and discusses related work. It ends with a discussion of open research questions.

3.1 Cloud Infrastructure Challenges

Cloud infrastructure challenges include both low-level performance challenges and higher-level utilization and availability challenges. The virtual nature of IaaS resources introduces an additional layer of overhead, potentially degrading performance and increasing the complexity of the software stack. However, despite these challenges, there has been much work recently to address the performance concerns of virtual IaaS resources. Solutions to address higher-level infrastructure challenges, such as offering on-demand availability while still achieving reasonable resource utilization, vary from one cloud implementation to another.

3.1.1 Performance

At a low level, infrastructure clouds typically use commodity hardware and virtualization solutions, such as Xen [42], to provide contained environments for users. Virtualization allows users to securely share the same underlying hardware. Users can install custom software stacks, including different operating systems, and configure them as needed. Virtualization, however, introduces another layer of overhead, which can be problematic for workflows with strict performance requirements. Recent advancements from hardware vendors, including AMD-V [4] and Intel VT [16], have minimized much of this overhead, especially for the CPU. Even if CPU cores aren't shared, highly active users may saturate other resources, such as the network or disk, negatively impacting other users in the cloud. Recent research, including VMM-bypass techniques [74], show promise for minimizing hypervisor overhead, however, they have yet to be widely adopted and deployed on existing clouds and potential security issues must be examined and addressed. Amazon's high-performance cluster instances provide reasonable performance, as demonstrated by their recent inclusion in the Top500, and eliminate the noisy neighbor problem by not allowing users to share the underlying node hardware. Amazon's provisioned IOPS for EBS volumes allow users to achieve a guaranteed level of IO performance required by certain applications, however, many of these offerings come at a high cost and are unavailable for smaller and cheaper instances. Finally, Amazon is one of the only major public cloud providers that currently offer many of these high-performance features, which limits deployments that require such features to Amazon's infrastructure.

Creating large-scale elastic environments for user applications and workflows also face challenges because of their distributed nature. Elastic environments that extend local resources with

IaaS resources may need to operate over the Internet, which introduces endless potential for poor performance and unreliability. If the environment extends an HPC cluster these problems are only exacerbated since cluster software and applications typically assume the opposite, that is, relatively reliable and known performance with minimal interference on shared resources. As an example, an elastic environment deployed across the Internet may have trouble using NFS to provide a shared file system. Instead, more modern file systems with better reliability for unpredictable network performance should be used, such as Gluster [13] or XtreemFS [76]. Large organizations may have the appropriate resources to establish peering agreements with cloud providers, such as the Ocean Observatory Initiative's (OOI) use of CENIC and NorthWest GigaPoP's peering agreement with Amazon [20] that provides two 10 Gbps connections into Amazon Simple Storage Service (S3) and Elastic Compute Cloud (EC2). Unfortunately, this is not likely a solution for many research groups that instead must use or develop sufficiently reliable software solutions.

Lastly, many cloud providers offer a large number of instance types that vary significantly in performance. Amazon, for example, currently offers over 15 instance choices, that cost from $0.02 per hour for a micro Linux instance to $4.60 per hour for an eight extra large Linux instance. Additionally, any number of VM images with different software configurations can be deployed on these instances that also impact performance. And finally, public cloud providers do not publish detailed technical specifications of the underlying hardware. Users are instead left with the burden of benchmarking and analyzing different instance types in order to identify the optimal price-to-performance ratio for their applications and workflows. This is a time consuming process and also a potentially expensive one since a large number of instances must be deployed and benchmarked regularly in order to have an up-to-date understanding of a particular cloud's performance. A central and well-defined "live" benchmarking platform for scientific computing would help to provide insight into the current performance of different IaaS clouds. This would allow users to determine if a cloud's current price-to-performance ratio is desirable for different instance classes.

3.1.2 Availability and Utilization

Another challenge faced by IaaS providers is increasing infrastructure utilization without sacrificing on-demand requests. To ensure on-demand availability, IaaS providers must over-provision their infrastructure, however, over-provisioning significantly reduces utilization. IaaS providers that choose not to over-provision their infrastructure risk rejecting user requests, no longer providing on-demand availability. Different IaaS implementations have addressed this issue using a variety of approaches, from offering different types of resource leases to powering off or suspending idle resources.

Amazon EC2 uses "spot" instances to address this challenge. With spot instances, users place bids for the amount that they are willing to pay for an instance, a rate typically lower than the on-demand cost of the instance. Amazon sets a spot price and terminates any bids that fall below the spot price. Amazon can therefore serve a larger community of users and increase utilization of their resources, while also reserving the ability to reclaim infrastructure when needed (and without notifying spot users). User applications and workflows that can leverage spot instances can potentially move to the cloud for lower cost than applications that require higher cost on-demand instances. Volunteer computing systems, such as BOINC [37], and workflows with sufficient checkpoint restart capabilities may be able to use spot instances effectively with little or no modifications, however, long running applications or large parallel application that do not include support for fault-tolerance may need significant enhancements in order to use spot instances. An alternative to offering a multitude of resource leases is to instead suspend or power off idle infrastructure resources, also known as "green computing." For example, Lefevre et al. [86] propose a framework for energy efficient cloud computing to reduce electrical consumption by migrating tasks to consolidate jobs, where possible, and then powering off idle nodes. Energy efficient approaches allow RPs to save on energy costs, however, resources that are completely powered off go unused and are not available to users at a lower cost. Therefore, RPs should may choose to consider green computing techniques that dynamically power off unused components of individual nodes, greatly reducing energy use but still keeping them available.

While green computing techniques allow RPs to conserve on energy costs and spot instances are a viable solution for public cloud providers that charge for use with real currency, private and community IaaS providers require solutions that do not require monetary bids. For example, IaaS toolkits should allow RPs to deploy VMs on idle resources that can be preempted by on-demand requests. Preemptible instances will allow users with appropriate workflows and applications to use over-provisioned IaaS infrastructure effectively, while the RP is able to increase utilization without sacrificing on-demand availability or powering off idle nodes.

3.2 Platform Challenges for Elastic Environments

Platform-layer services orchestrate deployments across IaaS clouds and create large-scale elastic environments that adapt to changing user demand. Therefore, platform services must address challenges related to resource provisioning, scheduling, and contextualization as well as application and workflow integration. In this section, we examine platform-layer challenges required to create useful large-scale elastic environments and the related work.

3.2.1 Dynamic Cluster Resource Provisioning

VioCluster [109] is one of the first systems to adapt a cluster to changing demand. A prototype system, VioCluster leveraged User Mode Linux to balance nodes across multiple clusters dynamically and transparently to the user. The implementation used Portable Batch System (PBS) [72] clusters and a virtual domain that consisted of virtual machines and physical machines. In VioCluster, a virtual domain is configured as a single cluster that is managed by a PBS job scheduler; one physical machine is designated as the PBS master node and the remaining nodes are designated as compute nodes.

Each physical domain of machines, e.g., departmental clusters, creates policies to borrow or lend VMs with other domains. The borrowing domain supplies the disk image to boot on the physical machines. The decision of whether or not to borrow or lend machines is performed by the domain's broker service, which implements the administrator-defined policies. The prototype deployment of VioCluster calculates demand as the sum of all of the nodes required by the jobs in the PBS queue. The system then calculates the number of machines it would attempt to borrow (or have available to lend) as demand minus the current number of machines it has available (virtual or physical). If the result is positive, the broker attempts to borrow machines, and if the result is negative, then that is the number of machines the broker can lend. The prototype system also implements a primitive reclamation technique whereby any request of a domain to reclaim a lent physical machine is immediately granted. Nodes in the virtual domain communicate with nodes in the physical domain via an overlay network. The effect of such an overlay is that all of the virtual machines (outside of the original physical domain) have a uniform and private IP address space that are shared with the physical nodes and provide access to all of the domain services (e.g., an NFS-mounted file system).

In-VIGO [32] is a more general and complex environment than VioCluster. In-VIGO combines Grid techniques with virtualization to support engineering and scientific research communities. Virtualization is used to create dynamic, on-demand pools of resources for application- or user-specific purposes. In-VIGO takes a three-layered approach to deploying and customizing these environments. First, a base layer of resources, including compute, network, and storage resources are deployed and connected together using virtualization and Grid technologies. The second layer deploys Grid-enabled applications on those resources. The third layer builds on the previous two layers by aggregating services (e.g., through a portal) and exporting an interface to the user through an XML-based User Interface Manager.

In-VIGO uses a virtual distributed file system and a virtual network to connect its VMs. An In-VIGO resource manager interfaces with queue managers (e.g., Torque or HTCondor) as well as Grid resource managers (e.g., GRAM) and underlying operating systems to provide a single API and point of access for managing the diverse set of resources. The resource manager is responsible for determining the resource requirements of a particular job, selecting the necessary resources, and then executing the job on the resources and monitoring its progress. Applications are bundled and deployed as virtual appliances. Significant effort is required to bundle and deploy applications in the In-VIGO environment. First, the application must be deployed on all of the Grid-enabled resources, then an XML configuration file must be defined for the application, and finally a set of rules must be defined that specify the application's requirements and behavior. The rules consist of Java classes that are used by the resource manager to locate the appropriate resource and execute the application.

Cluster resource provisioning techniques rely on implementation-specific approaches that are customized for individual deployments and do not expose services at the API-level. Large-scale elastic environments, however, require API-level access for resource provisioning in order for applications and policies to adapt dynamically to demand across multiple providers. Generic resource provisioning services, techniques, and policies must be developed to integrate distributed compute and storage resources for elastic environments. Additionally, application-specific adaptors should be developed to allow different applications and workflows to integrate with distributed deployments and adapt dynamically to demand.

3.2.2 Contextualization

Individual VMs in elastic environments must be configured as part of a common group, a process referred to as contextualization, so that VMs are aware of each other and can communicate in order to run parallel jobs. Contextualization, for example, may involve exchanging SSH keys and networking information so all nodes are configured to trust each other, as shown in Figure 3.2.

Keahey et al. [82] present an architecture for contextualization. The authors implement a secure system, centered on a context broker service, that contextualizes a dynamic set of resources. The contextualization solution can, for example, transform a set of standard Linux-based cloud VMs into a cluster where one node functions as the head node, another exports a shared file system to the cluster, and the remaining serve as compute nodes. Contextualization involves three parties: the appliance provider, the resource provider, and the appliance deployer. The appliance provider prepares the specific appliance and provides the VM disk image to the appliance

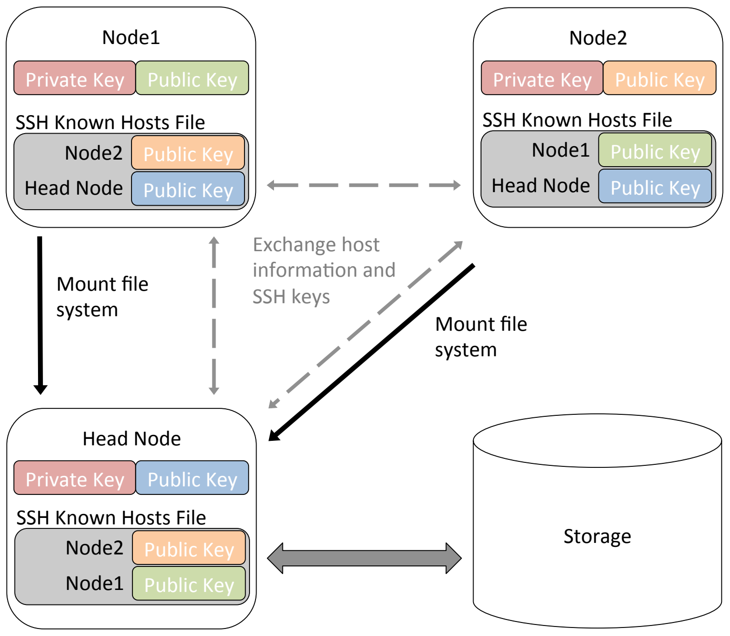

Figure 3.2: An example HPC cluster configuration, which typically requires exchanging host information and SSH keys and mounting a shared file system across all nodes.

deployer along with a contextualization template, which includes information about the components within the appliance that need to be contextualized (e.g., IP addresses, hostnames, SSH keys, etc.). The appliance deployer starts the image on the resource provider's infrastructure. A minimal set of context information provided to the appliance by the deployer include: network address and hostname, address of the context broker, a context identifier, and a set of credentials that prove trust between the appliance and context service. The context broker is a service that manages objects that contain context-specific information and facilitates the exchange of this information between all instances. The objects also contain the security and trust relationships for the context. Because the appliance will need the contextualization information shortly after boot, the information contained within the object must stabilize between all instances in the context, that is, the object must contain its complete set of information for all instances that are part of the context.

The authors provide implementations on Amazon EC2 and Nimbus. Both EC2 and Nimbus provide IP address information via DHCP. EC2 also allows users to communicate 16 KB of unstructured data to the VM via a metadata field that is associated with the cluster of virtual machines. Nimbus allows users to patch the virtual disk image with a file that contains context-specific information. After the virtual machine boots, a service running within the VM, known as a context agent, processes the file and configures the necessary components. Both of these methods only provide information in one direction: from the appliance deployer to the VM.

The context broker allows communication of context information in both directions via a context template. The template has two major components: provides and requires. The provides section of the template describes the role of the VM and the requires section of the template describes what information is needed to contextualize the VM. The VM contains application-specific contextualization scripts that use the finalized template to configure the virtual machine after the contextualization process has stabilized. The context broker is a standalone service that facilitates the configuration of a common context between independent resources; it does not assist the user launching, monitoring, or repairing these deployments.

Bresnahan et al. [44] present a tool, cloudinit.d, that allows users to launch VMs across multiple cloud providers and configure them to perform any function. With cloudinit.d, users can monitor these deployments and repair them if necessary. Users define launch plans that specify what services they want to launch. Similar to the UNIX init.d program, services can be grouped into different run levels that may depend on services in the previous levels. For example, a user may deploy a fairly typical database-backed website with cloudinit.d where the first run level boots a VM and configures it to serve as the database. Once the database VM(s) have launched and are configured, the second run level would then boot and configure the HTTP servers that connect to the database VM(s).

Wrangler [78], another tool with similar functionality to cloudinit.d and the context broker, provides a service that allows users to provision, configure, and manage clusters deployed on IaaS clouds. Users define their clusters in an XML format, specifying information such as the cloud to provision VMs on, VM image to use, and hardware type. Users specify different roles for nodes, which configure themselves appropriately based on their role. Users interact with the Wrangler coordinator service using a client that allows them to launch new clusters, list active clusters, terminate clusters, and obtain detailed information about clusters. Nodes interact with the coordinator through agents that manage node configuration and state.

System integration frameworks, specifically Chef [6], typically include many capabilities required for recontextualization. However, because Chef doesn't have a notion of "clusters" (e.g., HPC clusters) it would have to be extended to manage independent clusters of nodes. That is, Chef can be used to distribute scripts out to nodes for contextualization and gather their results but Chef can't manage distinct sets of nodes as clusters. Other standalone tools, including the Nimbus context broker, Wrangler, and cloudinit.d, don't have the ability to recontextualize an environment as it changes, distributing configuration scripts to nodes periodically, which is a key requirement for elastic environments that continually adapt to changing demand. Individual VMs in elastic environments may be added as demand increases or removed as demand decreases, and as these events occur, the remaining VMs in the elastic environment require updated information about the new set of VMs in the environment. Currently, no generic service exists to recontextualize independent clusters of virtual or physical nodes.

3.2.3 Dynamic Resource Scheduling

Once an underlying infrastructure capable of supporting elastic environments is in place, appropriate mechanisms for provisioning the resources and scheduling work on those resources needs to be developed. If there is a monetary cost associated with the infrastructures, this should be considered by the resource provisioning policies.

Currently, there are a number of existing resource management platforms that dynamically integrate and schedule resources. HTCondor [117], for example, uses a distributed job submission mechanism where any machine in the pool can submit jobs and maintain a queue of its jobs. However, a central manager is required to orchestrate the entire lifecycle of the jobs, which is responsible for tracking the queues of user-submitted jobs as well as tracking the status of all of the machines in the pool. When a job is submitted, the central manager queries for all available machines and idle jobs, it then matches jobs with machines based on job requirements and the characteristics of the available resources.

PlanetLab [50] uses a hierarchical process, based on tickets and leases to locate and schedule resources. A resource monitor runs on each node and periodically reports the status of the node to one or more agents. Depending on node availability, the agent issues tickets for the node and may decide to overbook the node. Each service in PlanetLab runs a service manager that is responsible for contacting an agent to obtain tickets. Service managers redeem tickets directly at the nodes. Moreover, each node runs a node manager that accepts tickets and compares them against the node's admission control policy to determine if the tickets can be redeemed. If the request can be fulfilled, the node manager reserves the resource, creates a VM, and returns a lease. The service manager is then responsible for starting the service in the VMs.

Dynamic resource managers, including HTCondor and PlanetLab, don't factor a specific cost for the resource (compute or data usage) into their resource allocation and scheduling strategies, instead, they schedule tasks based on the requested number of resources, the type of requested resource, and resource availability. Additionally, these resource managers often attempt to ensure some level of fairness, typically defined by the administrator, amongst the users when scheduling the resources. Separate from these dynamic resource managers, market-based systems explicitly schedule resources based on a direct cost for resource usage [87], [85], [41]. The motivation of these systems is to use an explicit cost for resource usage as an incentive for users to be honest about their resource requirements. Market-based systems seek to maximize profits for RPs by scheduling the requests that are willing to pay the most. They are not focused on selecting the optimal cost from a user perspective and lack the ability to minimize costs for generic user workloads.

3.2.4 Application and Workflow Integration

Workloads and applications also need to be extended to support dynamic resource provisioning used by elastic environments. Applications can either be modified directly to support elastic computing functionality or the functionality can be bundled in generic services. If generic services are developed, appropriate hooks must also be developed and integrated with applications or services. Bundling elastic computing functionality directly into applications has a number of advantages. First, it reduces overhead since extra components and services don't have to be deployed or administered by users. Second, policies for provisioning and relinquishing resources can be tailored specifically to the particular application and its workflow. Numerous applications leverage this approach and we highlight a few relevant examples in the following paragraphs.

OpenNebula [115] is an open source IaaS cloud computing toolkit that manages private cloud resources in addition to public cloud resources using a tightly integrated approach, enabling the creation of hybrid clouds. A set of cloud drivers in OpenNebula interface with third-party, external cloud providers. OpenNebula's cloud drivers currently support Amazon EC2 and ElasticHosts. OpenNebula only manages infrastructure cloud resources (both local and remote), it does not interface with local services or workflows, such as a local cluster resource manager.

In [98] the authors adjust the Sun Grid Engine (SGE) (now Oracle Grid Engine) [65] to deploy clusters on demand (COD). A single physical cluster is partitioned into multiple virtual clusters with VMs. Each virtual cluster is capable of meeting the needs of diverse user communities. CODs rely on Virtual Cluster Managers (VCMs) to add or remove nodes from clusters dynamically. VCMs use the current system load and a site policy to determine when to add or remove nodes from a cluster. More recently, the Oracle Grid Engine (OGE) has directly added support to provision resources on Amazon EC2 for job execution.

MIT's StarCluster [17] is a tool to create and manage batch-queue clusters on Amazon EC2. It provides tools for users to launch a cluster, dynamically add nodes to the cluster, and terminate the cluster. The Network File System (NFS) is used to share data between nodes. Two of the more interesting features of StarCluster include the ability to load balance the cluster, using OGE's load balancer, and the ability to use Amazon's spot instances as workers. In the current implementation, StarCluster only supports OGE and Amazon EC2.

In [60] and [122] the authors integrate elastic computing functionality directly into specific end-user applications, as opposed to the previous examples that were more general and focused on extending resource managers (OGE) and IaaS software (OpenNebula). In the former, the authors create a custom Amazon Machine Image (AMI) for their application, which provisions instances on Amazon EC2 directly and executes the MIT General Circulation Model. SSHFS [29] is used for data transfer. In the latter, the authors measure CPU utilization and memory usage on resources and then provision and relinquish resources based on a feedback control mechanism.

Similar to application-specific integration of elastic computing functionality, individual commercial cloud providers, such as Amazon, often provide their own tools for dynamically scaling cloud deployments based on resource utilization. Amazon Cloud Watch [2], for instance, monitors the demand on EC2 instances and automatically launches additional instances when a user-specified threshold is exceeded. These tools are cloud-specific and require that only cloud-provider supported services be used, and thus, are not general solutions for interfacing with a multitude of cloud providers or local site services and resources.

Instead of extending applications with elastic computing functionality directly, a separate service can be developed that includes such functionality. In this approach, applications must have the appropriate hooks to integrate with the service. Because the vast majority of elastic computing can be implemented in a single service, it greatly reduces software engineering time and the effort required to maintain duplicate code. Furthermore, a single service is more extensible because support for additional cloud providers is only added once and then leveraged by all applications that use the service.

Cloud Scheduler [40] is a loosely coupled approach for integrating local services with remote cloud resources. Cloud Scheduler provides a separate layer between local site services and cloud providers; it is not tied to a specific local service or a specific cloud provider. As of March 2013, Cloud Scheduler's implementation supports Nimbus clouds and Amazon EC2, and can also interface with a HTCondor queue to monitor and measure demand. Currently, Cloud Scheduler launches instances on cloud resources until its requests are denied. In the case of EC2 a hard limit is placed on the number of instances that it will attempt to launch. Cloud Scheduler divides the instances evenly among all users and rebalances the infrastructure in a greedy fashion, immediately terminating running jobs if necessary. If a resource is terminated before its job completes, the job is re-queued by HTCondor. Cloud Scheduler does not include a rich set of policies to match resource deployments with workloads, nor does it dynamically adjust cloud deployments to minimize costs.

Integrating third-party cloud resources with applications and workflows introduce security concerns as well. Individual sites need to consider the sensitive nature of their data. Sites should determine whether encryption will sufficiently protect their data or if certain datasets should not be transferred to the cloud at all. Solutions that dynamically extend local resources with cloud resources must limit the additional security risk posed to the site by adequately securing the remote instances. For example, standalone cloud instances can be secured with host-based firewalls, only allowing access to necessary services.

These existing approaches lack the necessary components to create end-to-end elastic environments for scientific workflows that integrate with a variety of cloud providers. Application-specific approaches limit elastic functionality to the individual applications and require high software engineering costs to maintain and extend. More generic approaches, including StarCluster and Cloud Scheduler, allow users with OGE and HTCondor workloads to leverage basic elastic environment functionality, such as outsourcing jobs to IaaS clouds. However, StarCluster limits users to Amazon EC2, OGE, and NFS, which may not be adequate for all scientific users. Cloud Scheduler limits users to HTCondor, which then dispatches tasks to Amazon EC2 and Nimbus clouds. These existing solutions lack advanced resource provisioning policies that adapt appropriately to demand and sufficiently reliable and parallel file systems for workloads with significant data requirements. They also do not provide a solution for recontextualization, which is required for parallel jobs.

3.2.5 Commercial Solutions

Numerous commercial providers have developed their own platform-layer services to help orchestrate and manage large deployments of IaaS instances. Amazon Auto Scale [1], for example, integrates with Amazon's CloudWatch metrics and lets users automatically scale their deployments based on defined conditions, such as resource utilization. VMware's Cloud Foundry [7] and Red Hat's OpenShift [22] provide PaaS environments for applications. Cloud Foundry supports Java, Ruby, Node.js, and Scala applications and a handful of database services, including MySQL and MongoDB. OpenShift supports Node.js, Ruby, Python, PHP, Perl, and Java applications as well as MySQL, PostgreSQL, and MongoDB databases. These PaaS environments allow users to launch scalable deployments without having to focus on or optimize infrastructure middleware services for their operations. Unlike Amazon's Auto Scaling service, these PaaS solutions do not provide a "bare" platform for any application; instead, they operate as a typical PaaS service and limit the applications that can be deployed and the services that they can use. RightScale [26] offers a variety of products, including a Multi-Cloud Management application that abstracts away the details of managing groups of instances across data centers, even those operated by different providers. RightScale also provides an auto-scaling service based on load information, similar to Amazon's auto-scaling service.

However, these commercial solutions have their own limitations. For instance, Amazon's auto-scaling service only interfaces with EC2 and does not provide a generic and open service for scientific communities that interfaces with a variety of providers. Amazon's CloudWatch metrics do not provide seamless integration with common scientific workload managers. PaaS solutions, such as Cloud Foundry and OpenShift, offer relatively open platforms for individual applications, however, these environments are often better suited for Web-based applications instead of scientific workflows with complex software stacks that often do not use the programming languages supported by large PaaS solutions. RightScale's approach allows their customers to leverage multiple IaaS providers, but they do not provide an open solution that integrates with community or scientific cloud providers. Sufficiently open solutions to create large-scale elastic environments that integrate with common scientific workload managers and private and community clouds are needed.

3.3 Financial Constraints

Monetary constraints must also be considered in any analysis as IaaS providers often charge for use. Large public IaaS providers may appear to offer “infinite” capacity, however, the scale of the deployment will certainly be limited by the budget of the user or RP deploying an elastic environment. Solutions to address the cost of cloud deployment must consider both compute and data costs, which may be significant for scientific workloads. In this section we examine cost-based grid scheduling policies as well as recent work that investigates the cost of IaaS deployments.

3.3.1 Cost-based Grid Scheduling

The majority of cost-based scheduling algorithms for Grid resources expect the environment to support advance reservations, which are then scheduled for different phases of a workflow. Yu et al. [121] develop an approach that schedules workflows on pay-for-use Grids that minimizes execution time while meeting a deadline. The approach assumes that it can accurately predict the execution time of tasks in the workflow as well as reserve available resource slots. The workflow is modeled as a Directed Acyclic Graph (DAG) and the overall deadline is divided into sub-deadlines that are assigned to various stages of the DAG. To find the optimal schedule of the entire workflow, optimized schedules are computed for each sub-deadline of the DAG by selecting the schedule with the smallest cost that still meets the sub-deadline. The system reschedules tasks if an initial deadline is violated.

Singh et al. [112] consider the impact that static resource provisioning (with advance reservations) and dynamic resource provisioning (through HTCondor Glideins) have on workflow execution time in Grid environments. The authors evaluate three different scheduling policies: FIFO, fair share, and fair share with backfill. In their treatment, they attempt to minimize the cost that each of these methods has on workflow execution, where cost is defined to be the amount of time that a job must wait before it is executed. To minimize cost, the authors query different resource providers to determine the number of resources available and the amount of time that each resource will be available. The resource that provides the earliest expected completion time for the workflow is selected. The completion time is estimated as the sum of the expected wait time and the expected execution time, which is directly related to the number of resources available for the workflow. In the advance reservation case, a reservation is made for the earliest possible time. With dynamic provisioning, jobs are submitted immediately and may begin execution earlier or later than expected depending on currently running jobs and any future jobs. In general, the authors find that dynamic provisioning provides quicker turnaround times for workloads, especially when using FIFO or a fair share scheduling. Backfill only improved completion time for fine-grained workflows.

In [111], Singh et al. develop a reservation price model for Grid computing environments that supports advance reservations. The price of the reservation depends entirely on the negative impact that the advance reservation has on queued best-effort jobs. If the advance reservation does not delay any queued best-effort jobs, the additional cost of the reservation is zero. However, if the advance reservation delays queued best-effort jobs, the cost of the reservation increases depending on the amount of delay introduced by the reservation. The intent of the reservation price model is to strike a balance between resource providers who are hesitant to allow advance reservations (due to their negative impact on resource utilization) and users who, under certain circumstances (e.g., deadline-driven), may be willing to pay more in order to assure execution of their jobs.

In [113], Singh et al. devise a cost model for Grid-based compute resources that support best-effort execution as well as advance reservation resource provisioning. RPs advertise resource slots to users for a specific cost. Resource slots consist of the number of processors and the duration that they are available. Resource providers periodically adjust the cost of the resources based on demand and availability. In their work, they assume that there is reliable and accurate information about the expected workflow execution time. The model first identifies available resources and their associated cost, based on the estimated execution time. The model then attempts to minimize both the cost of the resources as well as the workflow execution time. The concept of domination [56] is used to compare different possibilities, which consider both the resource cost and the workflow execution time. However, this may provide multiple solutions so a multi-objective genetic algorithm

[56] is used to find the best solution.

While the previously discussed Grid scheduling algorithms and models consider the cost of executing workflows on resources, they all ignore certain cloud-specific characteristics. A key difference between Grid and cloud environments is that resources are provisioned on-demand in the cloud, thus there is no need to identify available slots on a static system and schedule them. Cloud environments also typically charge for data transfer and data storage in addition to compute cycles, all of which need to be accounted for in cost-based scheduling algorithms and models.

3.3.2 The Cost of Cloud Computing

Recent research examines the cost of using IaaS clouds to process workloads, specifically those that charge money for use. In [58], Deelman et al. simulate the execution of an astronomy application in the cloud to analyze the cost of different deployments. The authors examine three scenarios using Amazon's charging model: 1) assume that local resources are available and remote cloud resources are only provisioned occasionally, 2) assume that there is local data storage and the cloud is only used for computation, and 3) the entire workflow is executed in the cloud. The conclusions of this experiment showed that provisioning the fewest cloud resources is the least expensive option and that provisioning the most resources provided the best job turnaround time. However, the authors note that there may be cases where a handful of extra resources could be provisioned for only a slight increase in cost but still achieve a modest improvement in job turnaround time. The authors found that remote I/O generated the highest cost due to Amazon's relatively high data transfer rates and storage costs were negligible compared to compute for their applications.

Assuncao et al. [55] augment a local cluster of virtual machines with virtual machines from the cloud. The authors use two schedulers, one to manage jobs on the local virtual machines and another to manage jobs on the cloud resources using a variety of job and cloud scheduling strategies, including conservative, aggressive, and selective backfill strategies. Conservative scheduling schedules jobs as they arrive and requests may begin executing earlier than scheduled if they do not delay other jobs. Aggressive scheduling only schedules jobs at the head of the queue and other jobs are allowed to be scheduled if they do not delay the job at the front of the queue. And finally, selective schedules jobs that have waited in the queue for a certain amount of time. The scheduling strategies are then evaluated by simulating the San Diego Super Computer (SDSC) Blue Horizon job traces supplied by the parallel workload archive [24] and computing the cost of performance improvement for the different strategies. The main emphasis of the simulations focus on meeting job deadlines. The authors show that a cluster with heavy load experiences higher costs and that an increasing number of deadline-constrained jobs results in higher costs in order to meet the deadlines. However, their traces and simulations do not include advanced resource provisioning policies or include data information.

Mao et al. [90] develop a cloud auto-scaling mechanism that considers both user performance requirements and budget constraints to ensure workflow tasks complete within their deadlines for minimal cost. Their approach provisions and relinquishes cloud VMs and schedules tasks across the most cost-efficient resources. Workflows consist of tasks with deadlines that can be represented as a DAG. A scaling plan is used to determine the number of instances required at any given time while a scheduling plan selects the instance type to use for a particular task. The authors leverage existing performance estimation techniques to determine the amount of time required to complete an individual task on different instance types and an earliest deadline first algorithm to schedule tasks across VMs. This work, however, does not consider common batch-queue scientific workloads or the impact of data on the cost and performance of the elastic environment.

Instead of extending resource or workflow managers, another approach is to modify individual applications to support elastic resource provisioning. In [122] Zhu et al. extend two applications with the ability to provision resources using a feedback control mechanism. CPU utilization and memory usage are monitored and used to determine when to adjust the deployment by launching or terminating resources. The primary advantage of this approach is the ability to fine-tune the resource provisioning algorithms for a specific application and its workloads. More general solutions, such as services extending batch-queues, must balance the needs of numerous users, applications, and workloads.

Many of the existing cloud-specific scheduling and provisioning algorithms consider the cost of compute resources, and most attempt to balance that cost with job turnaround time or resource utilization, but these models typically ignore data transfer and data storage costs. Moreover, these approaches often require tight constraints on assumptions about workloads and user-provided information, such as deadlines. They are not designed for large multi-user batch-queue workloads that may not have user-specified deadlines and may consist of a mixture of large parallel jobs and serial tasks or HTC workloads. Therefore, policies that leverage IaaS clouds and minimize costs and job response time for multi-user cluster workloads are needed.

3.4 Open Research Questions

Many of the underlying tools and services needed to support an open and flexible cloud computing architecture, capable of adapting appropriately to low and high demand, are currently available. IaaS services and standalone tools for contextualization and configuration provide building blocks for large-scale elastic environments that adapt to variable user demand. However, because IaaS clouds must be over-provisioned to offer on-demand resources, solutions to help IaaS providers increase resource utilization without sacrificing on-demand requests are needed. Specifically, solutions targeting the scientific community should be amenable to a variety scientific workflow paradigms and they should also be available in IaaS toolkits used to create community clouds. At the same time, an elastic environment capable of adapting to variable demand and outsourcing jobs to external clouds are needed. Such elastic environments require services to integrate with existing scientific workload managers and applications, recontextualization solutions to exchange instance information between all nodes as the environment grows and shrinks, and a reliable data movement solution that is transparent to users. Many existing cluster file systems, for example, were designed and developed for reliable networks with well-known performance characteristics, including high throughput and low latency. These file systems often experience difficulties when deployed across wide-area networks with unknown or highly variable bandwidth and latency characteristics. Additionally, new algorithms and policies need to be investigated and developed to create efficient elastic environments that balance user- and administrator-defined requirements, such as cost and job response time. These policies need to be optimized for different scientific workflows and applications since it is unlikely that single policy will be able to meet the needs of all users. The elastic environment should use these policies to adapt appropriately to demand, scale across multiple infrastructures, and be nearly transparent to users, seamlessly integrating with existing applications and workflow managers. Finally, these components must be combined together to provide a complete end-to-end flexible cloud architecture, capable of supporting diverse user communities with a wide-variety of requirements.

Chapter 4

A Flexible Cloud Architecture

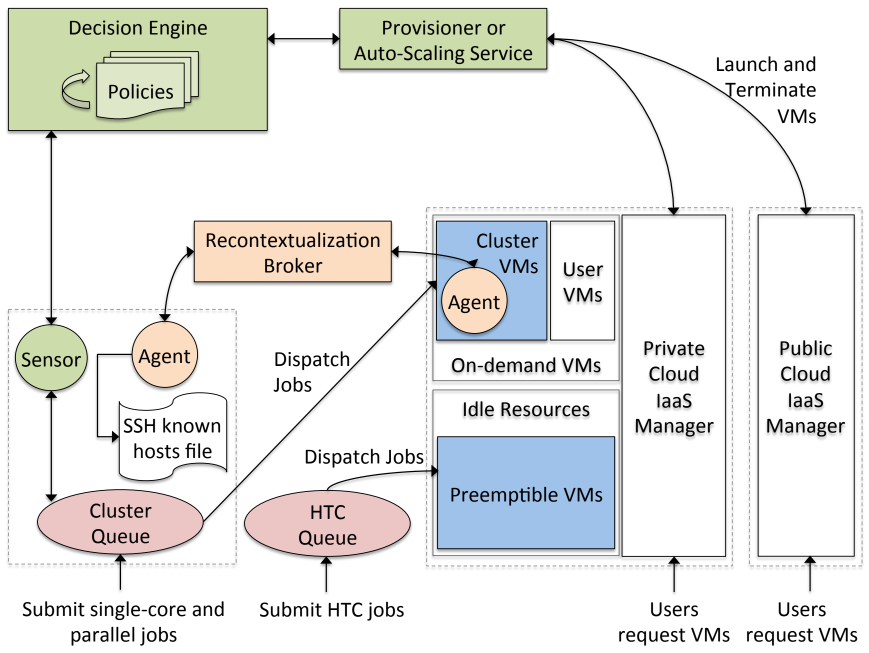

Figure 4.1: A flexible cloud architecture. The flexible cloud architecture responds appropriately to variable demand, increasing the utilization of over-provisioned IaaS clouds when demand is low and outsourcing workloads to external IaaS clouds when demand is high.

Standalone compute and storage resources experience dynamic load as demand fluctuates. These resources may be under-utilized when demand is low and over-utilized when demand is high. Creating an architecture that adapts to variable demand requires a flexible design and services that are capable of reacting to changing demand by automatically provisioning or relinquishing resources as needed, allowing RPs to operate their resources at a relatively high level of utilization while still responding to on-demand user requests in near-interactive time. Specifically, the system should meet the criteria identified in Chapter 1: elastic, customizable, scalable, and capable of maintaining high utilization. To address these requirements, we present a flexible cloud architecture, shown in Figure 4.1. The flexible cloud architecture provides a comprehensive and elastic system that adapts to variable demand, allowing RPs to improve utilization of over-provisioned IaaS resources as well as outsourcing excess demand to external clouds when needed. IaaS clouds provide VMs on-demand, allowing users to customize the entire software stack, and offer large-scale infrastructures to meet the needs of scientific workloads. Commercial IaaS cloud providers, in particular, typically operate massive data centers distributed across the globe, allowing users to deploy tens of thousands of instances when needed [9].

To increase utilization of under-utilized IaaS clouds, preemptible and preset leases, originally presented in [93], are proposed for the flexible cloud architecture. The open source Nimbus IaaS toolkit is extended to support these leases by deploying preemptible VMs on idle nodes. Preemptible VMs are generic VMs that can be configured to perform any function and may be terminated suddenly to service on-demand leases; they allow RPs to contribute cycles that would have otherwise been idle to other processes, such as HTC workloads [88] or volunteer computing systems (e.g., SETI@home [38] or Folding@home [85]). To adapt to high demand, an elastic environment is developed. First, a prototype elastic environment is created, originally presented in [92], to explore the necessary requirements and discover potential issues. Then, a large-scale design and implementation are developed. These components were originally presented in [95]. The elastic environment provisions and relinquishes cloud resources to adjust to variable demand. It can be used to extend existing site resources, such as an HPC cluster, with infrastructure clouds or it can be deployed entirely in the cloud, allowing users or RPs to outsource their entire workflows to the cloud when needed. The flexible cloud architecture, including preemptible VMs and elastic environments, is discussed in detail in the following sections. The implementation and evaluation of preemptible VMs are presented in Chapter 5 and the implementation and evaluation of the elastic environment is presented in Chapter 6. Resource provisioning policies for the elastic environment are presented and evaluated in Chapter 7. And finally, a bioinformatics use case with significant data processing requirements that leverages multiple infrastructure clouds is presented and evaluated in Chapter 8.

4.1 Preemptible VMs

Organizations may choose to deploy private IaaS clouds, using toolkits such as Nimbus [18], Eucalyptus [101], or OpenStack [23], in order to provide their users a local cloud for developing cloud-based workflows. However, to ensure on-demand availability for their user communities RPs need to keep a large portion of the resources idle. That is, they must either over-provision their cloud infrastructure and suffer low resource utilization or under-provision the infrastructure and reject a large number of on-demand requests, effectively eliminating the on-demand nature of the IaaS cloud. Low resource utilization is particularly difficult for the scientific community where batch schedulers typically ensure high utilization and much better resource amortization [120]. Thus, low utilization is a potentially significant obstacle to the adoption of IaaS clouds in the scientific community.

At the same time, not all scientific workflows require on-demand access to resources. For example, volunteer computing systems, including SETI@home [38] and Folding@home [85], leverage resources opportunistically. Such systems are designed to be failure resilient and tolerate unexpected interruptions in service. Perhaps the most salient example in the scientific community is HTC systems where the workloads do not have an immediate deadline, but instead must process substantial datasets on the order of months or years. HTCondor [117] is an example of an HTC workload manager, originally developed as a cycle-scavenger for idle desktop workstations. Therefore, we propose preemptible and preset leases for IaaS clouds to combine on-demand provisioning with the opportunistic allocation of cycles from idle IaaS nodes to other processes.

4.1.1 Preemptible and Preset Leases

Preemptible and preset leases provide infrastructure resources to users at an unspecified time for an indeterminate amount of time. The cloud administrator supplies the VM for these leases, that is, a user cannot deploy his or her own VMs without the assistance or intervention of the administrator. This differs from the on-demand and non-preemptible leases typically offered by cloud providers or spot instances provided by Amazon where users can provide their own VM and are either granted or rejected access to non-preemptible resources in near-interactive time.

Preemptible and preset leases deploy preemptible VMs on idle IaaS resources. In this context, preemptible VMs are generic VMs that can be configured to perform any function, such as contribute cycles to HTC or volunteer computing workloads. Preemptible VMs have two limitations: 1) they can be terminated suddenly at any time to free up space for the IaaS manager to service an on-demand lease, and 2) the unpredictable timing of on-demand leases (from any number of users) means that there may be a variable number of preemptible VMs running at any given time. Preemptible leases can be provided to users at a lower cost than non-preemptible, on-demand leases since the IaaS manager can reclaim resources when needed by terminating preemptible VMs. Because of these limitations, certain applications and workflows are not well suited for preemptible VMs (e.g., parallel applications, which require all processes to be available during the full execution). While large parallel applications are not ideal for volatile preemption environments, many workflow paradigms are suitable candidates for such environments. For example, HTC or volunteer computing workloads [117], which consist of jobs eventually needing to be processed but do not have an immediate deadline and do not depend on other jobs in the set, are typically designed for dynamic environments with a variable number of workers. In this case, it is acceptable to terminate HTC jobs in the middle of their execution and re-queue them for later execution as long as the workload is eventually processed.

4.1.2 Extending IaaS toolkits for Preemptible VMs

The underlying physical IaaS resources as well as the user communities, both for on-demand VMs and preemptible VMs, influence the design of the implementation and should be considered. First, applications and workflows that accommodate volatile environments should be identified; specifically, the applications and workflows should handle failures gracefully. Second, the granularity with which preemptible VMs are deployed should be determined. That is, will single-core preemptible VMs be deployed on multi-core IaaS hypervisor nodes? Or will multi-core preemptible VMs be deployed, possibly using all cores on the VMM? Single-core preemptible VMs allow the IaaS administrator fine-grained control over the deployment, however, the disadvantage is the increase in overhead required to run multiple VMs simultaneously on the same physical system. Multi-core preemptible VMs, and in particular a single preemptible VM per VMM node, reduces the virtualization overhead on the VMM node. The disadvantage of this approach is that an on-demand request for even a single-core VM may cause the preemptible VM to be terminated, leaving the remaining cores idle on the VMM node. An IaaS administrator may also choose to configure preemptible VM deployments based on other resources, such as RAM instead of (or in addition to) CPU cores.

The size of the preemptible VM deployment, relative to the size of the IaaS cloud, must also be determined. The preemption deployment may use all available nodes, however, the overhead required to terminate preemptible VMs in order to service on-demand requests should not be ignored. The overhead may be somewhat significant if preemptible VMs are shutdown cleanly (requiring up to 10s of seconds) instead of trashed immediately. Therefore, the IaaS administrator may choose to only allow preemptible VMs to utilize a fraction of idle resources, leaving some resources available to service on-demand requests immediately. Finally, the mechanism for deploying preemptible VM images to idle VMM nodes should be selected. One choice is to transfer a fresh preemptible VM image from the IaaS image repository for every preemptible VM deployment (commonly referenced as VM image propagation). However, this approach introduces additional network contention and may slow the deployment of on-demand VMs, depending on the underlying storage and network architecture of the cloud. A second choice is to propagate the VM image for preemptible VMs to the VMM nodes and cache it, only redeploying the image if it has been updated or removed from the cache on the VMM node. A third choice is to manually transfer the preemptible VM image to all nodes, reducing network contention (since it is only performed when the administrator chooses to, ideally when the network is not heavily utilized) and reduce the launch time for preemptible VMs since the image doesn't need to be copied across the network. The main disadvantage with this approach is that the administrator must manually copy the preemptible VM image to all nodes when it is updated.

4.1.3 Termination Policies for Preemptible VMs

Preemptible VMs are terminated when the IaaS service needs to reclaim resources on the cloud to deploy an on-demand VM. Terminating preemptible VMs impacts the applications running inside the VMs. Ideally, termination policies for preemptible VMs should consider the applications running inside the preemptible VMs, however, such policies would require a substantial investment in software engineering time and effort. Gathering and processing this information is highly application dependent and requires modifying the underlying IaaS toolkit to adapt to all application workloads running inside the preemptible VMs. Perhaps the most straightforward policy is to randomly terminate preemptible VMs until the on-demand VM can be launched. Unfortunately, if the preemptible VM is running a job then its work will be lost while other idle preemptible VMs continue running. Another basic policy is to terminate preemptible VMs that have been running for the least amount of time until the on-demand VM can be launched. This policy assumes that the preemptible VMs running the longest may also be executing jobs for the longest amount of time and, therefore, will lose the most work when terminated. The intuition behind this policy is that a workload consisting of long-running jobs (or a mixture of long and short jobs) will be impacted the most by terminating preemptible VMs that have been running the longest whereas a workload consisting of short-running jobs will only be moderately impacted by the termination of any preemptible VM. More advanced termination policies for preemptible VMs can use "hints" from the IaaS scheduler and form a more complete picture of cloud VM placement to terminate preemptible VMs. For example, if the termination policy is aware of VM placement on VMM nodes, it may be able to terminate the fewest number of preemptible VMs instead of blindly terminating preemptible VMs until an on-demand lease can be fulfilled.

4.2 Elastic Environments

In addition to increasing low infrastructure demand, the flexible cloud architecture can also adapt to variable user demand with an elastic environment. The elastic environment adapts to variable demand by provisioning IaaS instances as demand increases and relinquishing IaaS instances as demand decreases. In cases where demand exceeds the capacity of site resources, such as an HPC cluster or a private cloud, the elastic environment can outsource demand to external community or public IaaS clouds with minimal user intervention. The elastic environment can also be deployed by users or RPs completely in the cloud, operating as a standalone entity, allowing users to process their workflows entirely in the cloud when needed.

A number of different approaches can be used to create the proposed elastic environment. Using an application-specific approach, elastic functionality can be bundled in individual applications, including the ability to monitor demand, provision and relinquish resources, and policies to ensure efficient resource utilization. The main advantage of this approach is that provisioning policies can be tailored to the specific application, that is, the application's policies don't need to consider a wide range of inputs and different execution paradigms. However, the main disadvantage is the high software engineering cost required to extend and maintain every application with elastic computing functionality.

A second, service-based, approach is to create a central service responsible for adapting to changing demand. The central service interfaces with individual applications through lightweight application-specific sensors and policies that monitor demand. By bundling much elastic environment functionality in a central service, the software engineering cost required to support a large number of cloud providers, which are constantly changing and updating their services and APIs, can be minimized. Custom policies for different applications can still be created and used by the central service, allowing it to adapt appropriately for many different applications and execution paradigms.

4.2.1 Elastic Manager Prototype

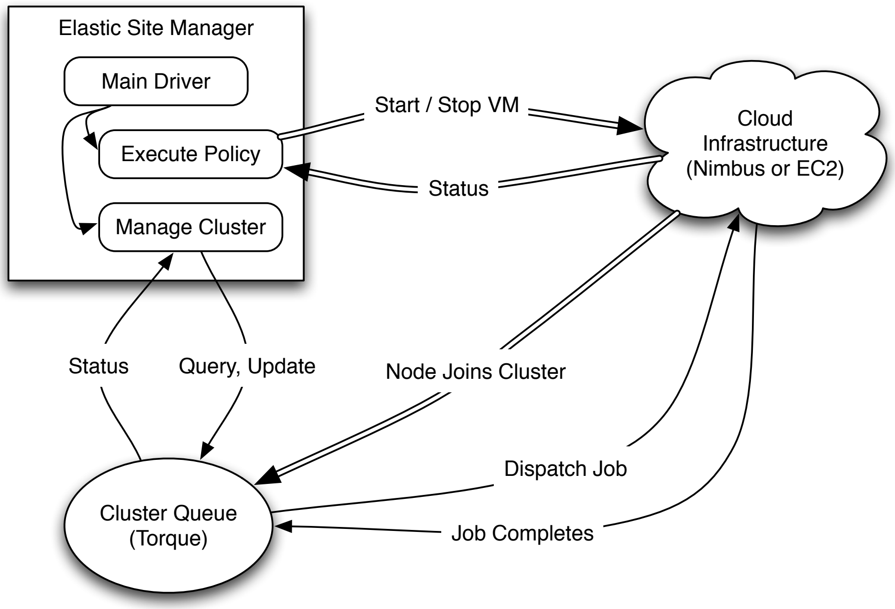

Figure 4.2: Architecture of the elastic manager prototype.

In [92] we created a prototype “elastic site” environment that dynamically extended a Torque cluster with IaaS resources. The architecture is shown in Figure 4.2. The prototype implementation used the service-based approach described above, that is, it encapsulated elastic computing functionality as a separate service. It monitored the Torque queue and responded to changes in the queue by launching or terminating instances on Amazon EC2 or Nimbus clouds, which were then added to the cluster at run time. The implementation dynamically opened and closed network ports using iptables. Cloud instances exchanged host information and SSH keys with the Torque head node through the Nimbus context broker [82]. Though the prototype provided valuable insight into the requirements and challenges of large-scale elastic environments; it had a number of limitations that prevented it from scaling appropriately, using multiple IaaS clouds simultaneously, and recontextualizing continually as the environment adapted to demand.

Due to the initial design and implementation, the prototype elastic manager ran directly on the Torque head node because it called and parsed Torque commands to perform cluster operations. The prototype called the Java-based Nimbus cloud client to launch and terminate instances. The Nimbus cloud client polls until operations complete, meaning that individual calls to launch a VM could take up to a few hundred seconds. This greatly limited the scalability of the elastic manager since it could only run, approximately, a dozen simultaneous cloud client threads launching or terminating instances. Therefore, the prototype was only able to launch a dozen instances every hundred seconds. While the prototype supported multiple cloud provider interfaces, it could only be configured to use a single cloud at a time. Furthermore, the prototype deployment used a preinstalled cluster image on each cloud and the image was preconfigured to trust the cluster head node and join the cluster at boot. This required that the image be setup completely on every cloud infrastructure supported by the cluster, a tedious and untenable process after the first time. The prototype leveraged the Nimbus context broker to exchange host information and SSH keys between newly launched cloud instances and the cluster head node. However, because the Nimbus context broker can only contextualize nodes at launch, additional instances cannot join or leave the context after the original group of nodes in a context have launched. And since the elastic environment continually adapts to demand (by launching and terminating nodes), the prototype could only launch single-node contexts so the head node and cloud instance exchanged information. The prototype also relied on Torque's ability to SCP individual files to and from the head node. Unfortunately, these limitations prevented the environment from supporting parallel jobs since the environment didn't provide a shared file system and cloud instances never exchanged host information or SSH keys with each other. Lastly, the prototype had no self-monitoring or repairing capabilities, which prevented the system from identifying failed instances and automatically replacing them. These issues are addressed in our later implementations, described in the following sections.

4.2.2 Large-Scale Elastic Environment

To address the limitations of the initial prototype, we present a scalable elastic environment capable of creating environments that span multiple clouds simultaneously and recontextualizing as nodes join or leave the environment. The environment can also deploy a reliable and shared file system that allows workers to access the same namespace. Instead of tightly integrating components, a distributed design for the elastic manager is selected that also uses a service-based approach. Our first iteration of the elastic manager leveraged components that evolved into the open source Phantom service [80], which is based on the Ocean Observatories Initiative (OOI) Elastic Processing Unit (EPU) [21]. The elastic management services provide a framework to monitor demand, react to demand, and launch and terminate instances across different IaaS clouds. The system consists of three main components: sensors that monitor demand, decision engines that respond to demand, and a provisioner or auto-scaling service to manage IaaS resources. Unlike the Java-based cloud client, the system uses IaaS Representational State Transfer (REST) interfaces and does not continually poll until operations, such as VM launches and terminations, complete; instead, it periodically queries the status of IaaS instances. The decision engine loops periodically and responds to changes in demand by electing to launch additional IaaS instances, terminate existing IaaS instances, or leave the environment unchanged. Sensors are deployed throughout the environment that monitor demand and send information to the decision engine. However, the initial elastic management components used a "pull" queue model where individual workers pull tasks from a central queue. This differs from a "push" queue model, such as those used by batch-queued clusters, where a central scheduler identifies available workers and dispatches tasks to the workers, supporting both single-core and parallel jobs. To support a push queue model, we created a custom decision engine that responds to job information, such as the number of queued jobs and the state of workers. In the push queue model, the sensors both monitor demand and execute operations on behalf of the decision engine, for example, by adding newly launched nodes to the cluster or opening ports.

Unfortunately, because our initial design was tightly coupled with the elastic management components, the entire elastic management software stack had to be configured and deployed, including numerous unneeded services. This version also used the advanced message queueing protocol (AMQP) [118] to communicate between all components, including the Torque sensor and core elastic management services. This resulted in a moderately heavyweight deployment with an unnecessary amount of required configuration and dependencies. To address this limitation, we updated the architecture so that the sensors and policy are now decoupled from the central elastic scaling services. Specifically, the architecture consists of the same basic components, including sensors to monitor demand, a decision engine that executes a policy to evaluate demand, and an auto-scale or provisioning service to scale the deployment as required. The auto-scaling service is also responsible for monitoring the deployment and replacing failed instances. The process of gathering sensor information, evaluating the policy, and responding to demand by expanding or contracting the deployment loops continuously. The sensors and policy now communicate with the central auto-scaling service using Amazon's auto-scaling API as implemented by open source auto-scaling services, such as Phantom [80]. This way, a user can deploy their sensors and policy on any system they choose (e.g., a local Torque head node) and these components can communicate with a central scaling service via a simple API supported by various libraries (e.g., boto [5]).

4.2.3 Automating Deployment and Configuration

To support multiple cloud providers seamlessly, the installation and configuration of worker nodes is automated using a system integration framework, such as Chef [6], and software package managers, such as Debian Apt [10]. Using this approach, a base image that is available on any cloud (e.g., a new Debian 5.0 installation) is configured as a worker at boot by the system integration framework and common software is installed using the software package manager. Once the software is installed, the worker is configured to join the cluster head node automatically.

Unfortunately, many scientific applications are not currently bundled using common software package managers. And worse yet, some scientific researchers are completely unaware of best practices and community standards for software development and packaging. For example, some developers only release a single source file of their application and do not version it as they continue to develop. Other developers require that variables be hard coded in source files at the time of deployment. To deploy complex scientific software stacks, we develop a generic software deployment tool, app-deploy, to automatically download, configure, compile, and install scientific applications. App-deploy allows users to specify the applications in a configuration file (e.g., the URL and build process) and then download, compile, and install the entire application stack into a contained environment. In cases where application deployments require extensive customizations, such as modifying source files at deployment time, the customization process can be confined to a single Python module.

4.2.4 Contextualization

The final component required to complete our elastic environment system is to create a shared and trusted context between all nodes in the environment, originally presented in [95]. For example, all nodes in a cluster must exchange SSH host keys and host information in order to communicate with each other and support parallel jobs. Because elastic environments continually adapt to changing demand the environment must be recontextualized periodically. To recontextualize elastic environments, we propose a recontextualization service that securely and periodically exchanges information between all nodes in the context. The service maintains an ordered list of nodes in the context, including hostnames, IP addresses, SSH keys, and a generic data field for all nodes. The service updates the list as nodes join or leave the environment. All nodes in the context periodically query the recontextualization service, which sends an order list of updates of the context to the node. The node then applies the updates in order (e.g., by adding the SSH keys of newly launched nodes to the SSH known host file). The entire large-scale elastic environment implementation, including application-adaptors, elastic management interfaces, and the recontextualization service are presented in Chapter 6.

Chapter 5

Preemptible VM Implementation and Evaluation

The focus of this chapter is to present our preemptible VM implementation and evaluate its effectiveness. To increase utilization of IaaS clouds while still offering on-demand VMs, the open source Nimbus IaaS toolkit is extended to support preemptible and preset leases that deploy preemptible VMs on idle cloud resources (as shown in Figure 5.1), further details can be found in [93]. The implementation allows RPs to address low demand by contributing IaaS resource cycles, which would have otherwise been idle, to other processes, such as running HTC or volunteer computing jobs. To evaluate the implementation, we configure the preemptible VMs as HTCondor workers to process an HTC workload. We evaluate the ability of the system to increase the utilization of the IaaS cloud without sacrificing the ability to service on-demand leases. We also consider the ability of the preemption-enabled environment to process HTC jobs with minimal workload overhead. We demonstrate that preemptible VMs are able to increase the utilization of an IaaS cloud from an average of 36.36% and maximum of 43.75% to an average utilization of 83.82% and maximum of 100% while only introducing 6.39% workload overhead.

5.1 Implementation of Preemptible Virtual Machines

The open source Nimbus cloud computing toolkit provides on-demand access to resources, allowing RPs to deploy a private or community IaaS cloud for their users. We extend Nimbus 2.6 to support preemptible and preset leases, deploying preemptible VMs on idle VMM nodes. Our implementation makes a number of assumptions: 1) the Nimbus administrator must configure

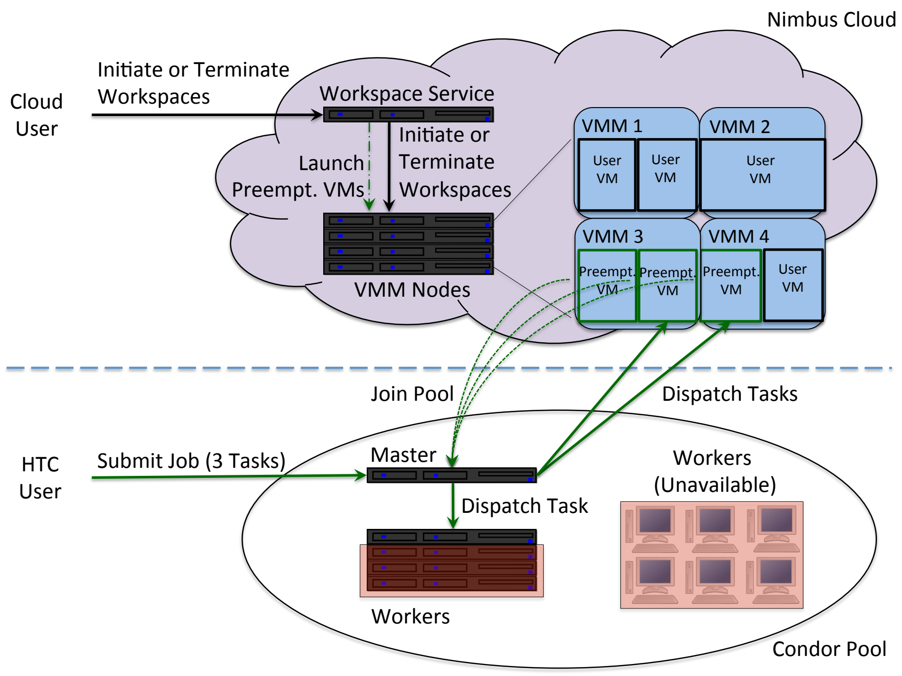

Figure 5.1: An example preemptible VM deployment that integrates idle IaaS resources, running preemptible VMs, with an HTCondor pool to process high-throughput computing tasks.

preemption, 2) one preemptible VM is deployed per VMM node, 3) unless a maximum is specified, preemption automatically attempts to deploy as many preemptible VMs as possible when enabled, and 4) preemption cleanly shuts down the preemptible VM, allowing applications running inside the preemptible VM to respond appropriately (e.g., by notifying a central scheduler to reschedule jobs running inside the VM). Moreover, our implementation supports two termination policies: selecting preemptible VMs at random and selecting the preemptible VM that has been running for the least amount of time in order to minimize work lost by unexpected and premature termination. In February 2011, our preemptible and preset leases were included in the Nimbus 2.7 release, allowing Nimbus to deploy preemptible VMs on idle VMM nodes.

5.1.1 Configuration Options

Configuration options for preemptible VMs are specified in a single file on the Nimbus service node, allowing administrators to enable and configure preemptible VMs easily. In addition to an option to enable preemptible VMs, the options include:

- Max.instances: Specifies the maximum number of preemptible VMs to launch. The default is 0 and launches as many as possible.
- Memory.MB: Specifies the amount of RAM to use for preemptible VMs. The default is 64 MB.
- VCPUs: Specifies the number of VCPUs to use for preemptible VMs. The default is 1 VCPU.
- Duration.seconds: Specifies the amount of time (in seconds) preemptible VMs can run before being terminated. The default is one week.
- Termination.policy: Specifies the termination policy. The policies currently supported include "most recent" to terminate the preemptible VMs that have been running for the least amount of time and "any" that terminates a random preemptible VM. The default is the former.
- Retry.period: Specifies the duration (in seconds) to sleep between attempts to deploy preemptible VMs on idle VMM nodes. The default is 300 seconds.
- Network: Specifies the network to use for preemptible VMs. The supported options include "public" and "private."

5.1.2 Nimbus Workspace Service Extensions

For implementation, we extended the Nimbus workspace service [18] to support preemptible and preset leases, which deploy preemptible VMs on idle VMM nodes. The workspace service manages VMM nodes and handles user requests. Preemption functionality is added to Nimbus by the creation of a separate Java class. When the Nimbus workspace service starts, the configuration file is read, and if preemptible VMs are enabled, preemptible VMs are deployed on any available VMM nodes. The process loops continuously, attempting to launch preemptible VMs on VMM nodes and sleeping for the duration specified in the configuration file between iterations.

In addition to core preemption functionality, we also modified the Nimbus scheduler to detect possible rejected on-demand requests and terminate preemptible VMs if needed. The current implementation terminates a single preemptible VM and then attempts to service the on-demand request. If the request still can't be fulfilled, it continues terminating preemptible VMs and attempting to launch user VMs until it either succeeds or all preemptible VMs are terminated, in which case the user request is rejected and the user notified. Rejecting the request up-front, without terminating preemptible VMs, would require a complete picture of VMM resources and VM placement on those resources, something not currently provided by any IaaS toolkit.

5.2 Preemptible VM Evaluation with HTC Workloads

To evaluate preemption, we consider two perspectives: First, we examine the ability of the system to increase utilization without sacrificing its ability to provision on-demand resources. Second, we examine the ability of the preemptible deployment to process HTC jobs using cycles that would have otherwise been idle. The evaluation environment consists of a preemption-enabled version of Nimbus 2.6, deployed on the Hotel cloud on FutureGrid [12]. Nimbus manages a group of 16 VMM nodes, each with single-socket 2.4 GHz 8-core Intel Xeon processors and 3 GB of RAM per core with 20 GB allocated to user VMs. This allows for a total of 128 single-core VMs. The Nimbus workspace service, responsible for servicing on-demand user requests for VMs, ran on an additional node on Hotel and also hosted the user VM image repository.

To evaluate our solution, we configure preemptible VMs to use the entire VMM node, that is, a single preemptible VM uses all 8 cores on the VMM node. To fulfill user VM requests (even single-core), the entire preemptible VM must be terminated. We chose this level of granularity for the preemptible deployment to reduce virtualization overhead on VMM nodes and avoid extra network contention caused by transferring a single preemptible VM for each core over the network. We also manually push the preemptible VM image out to all VMM nodes ahead of time, again, to reduce unnecessary network contention. Preemptible VMs are configured to join an HTCondor pool at boot and the HTCondor master runs on a separate evaluation node. The HTCondor pool only contains HTCondor workers from the preemptible VMs. Two additional nodes, the same as the VMM nodes described above, are used to generate and manage the workload. One of the nodes hosts the HTCondor master service and queues the HTC HTCondor jobs. The other node executes the Nimbus workspace service workload by requesting single-core on-demand user VMs from the Nimbus workspace service.

All evaluations use the "most recent" termination policy. This policy terminates the preemptible VMs that have been running for the least amount of time first. Preemptible VMs are shutdown cleanly when they are terminated, which allows the HTCondor workers running inside the preemptible VM to notify the master to reschedule any jobs running on the VM. If clean shutdown isn't used, HTCondor relies on timeouts to reschedule jobs; in our experience this can sometimes take up to two hours. (At the time of our evaluation, HTCondor had an experimental feature to reverse the direction of the pings used to determine the status of worker nodes, which would allow us to trash the preemptible VM and quickly reschedule its jobs. However, when we enabled the feature it did not behave as expected, therefore, we did not use it and instead chose to shutdown the preemptible VMs cleanly.)

5.2.1 Preemption Evaluation Metrics

We use the following four metrics to evaluate preemption from both the perspective of the IaaS cloud provider as well as the HTC user. The metrics are:

- *Utilization*: The percentage of user cycles consumed by CPU cores on the VMM nodes in the IaaS cloud. This includes cycles consumed by HTC jobs and on-demand user VMs. We do not include the cycles consumed by preemptible VMs that are not running HTC jobs. Therefore, it is possible for all VMM nodes to be running preemptible VMs, but not HTC jobs, and still have 0% utilization. Higher utilization is better.

- *First queued time*: The amount of time that elapses between the time a HTCondor job is first submitted to the HTCondor queue and when it begins executing for the first time.
- *Last queued time*: The amount of time that elapses between the time an HTCondor job is first submitted and the time the job begins executing before completing successfully. Because on-demand user VMs can cause the termination of preemptible VMs at any time, HTCondor jobs may be rescheduled multiple times before finally executing until successful completion. Shorter time is better since this metric represents the time it takes until the job is able to run successfully, producing results for the user.
- *User VM service response time*: The amount of time it takes the Nimbus workspace service to respond to a user request for on-demand VMs. More specifically, the amount of time that elapses between the time the workspace service first receives the request and the time it determines whether to reject the request or launch the appropriate VMs. This does not include the time required to propagate the VM images or boot the VM. If preemptible VMs are deployed, this metric does include the time to terminate them (either via clean shutdown or by trashing them).

5.2.2 Workload Traces

The workloads used for evaluation consist of real workload traces that are representative of common HTC and IaaS workloads. The workloads are adjusted to fit the size of our evaluation environment and are summarized in Table 5.1. The HTCondor workload is a workload trace from the Condor Log Analyzer at the University of Notre Dame [8]. It contains 748 jobs that each sleep for variable amounts of time. The minimum job runtime is 1 second and the maximum is 2089 seconds, with a standard deviation of 533.2. The trace submits 400 jobs immediately, followed by

Table 5.1: Preemption evaluation workloads that consist of serial jobs for the HTCondor workload and requests for 8-core VMs for the IaaS workload.

HTCondor Workload	
Number of Jobs	748
Minimum runtime	1 second
Maximum runtime	2089 seconds
IaaS Workload	
Number of available cores	128
Minimum on-demand instances	0
Maximum on-demand instances	56

348 jobs 2573 seconds later. The IaaS cloud workload trace consists of user requests for on-demand VMs. The workload is from the University of Chicago's (UC) Nimbus science cloud. This workload trace was chosen because it is generally characteristic of the traces observed on the UC Nimbus cloud, despite its lack of dynamism. The UC Nimbus cloud did not appear to be highly dynamic on relatively short time intervals (e.g., over a few hours), instead, users typically requested a specific number of instances for a long period of time (e.g., 6 VMs for 24 hours). The IaaS workload also demonstrates the expected behavior of an over-provisioned cloud, that is, it contains many idle VMM nodes that are available to service on-demand VMs. We also multiplied the requests for VMs in the workload by 8 because the UC Nimbus science cloud only contained 16 cores while our evaluation environment consists of 128 cores. Thus, a request for a single-core VM on the UC Nimbus cloud would be a request for 8 single-core VMs in our evaluation.

Both workloads submit individual and independent requests for a single core. The HTCondor jobs are "sleep" jobs that sleep for the desired amount of time. For the on-demand workload, VMs are launched and run for the appropriate amount of time. Because preemptible VMs use the entire VMM node, they are capable of executing 8 jobs concurrently across all 8 cores on the VMM node. Finally, we terminate the evaluation shortly after the overlapping HTCondor trace completes.

Despite the infinite number of possible on-demand and HTC workloads we could have considered, many of which may have artificially highlighted the usefulness of preemptible VMs to the on-demand IaaS user community or the HTC user community, we instead chose to base our evaluation off of two realistic workload traces. With these workload traces we are able to demonstrate the usefulness of preemptible VMs to both user communities given at least one possible scenario.

5.2.3 Understanding System Behavior

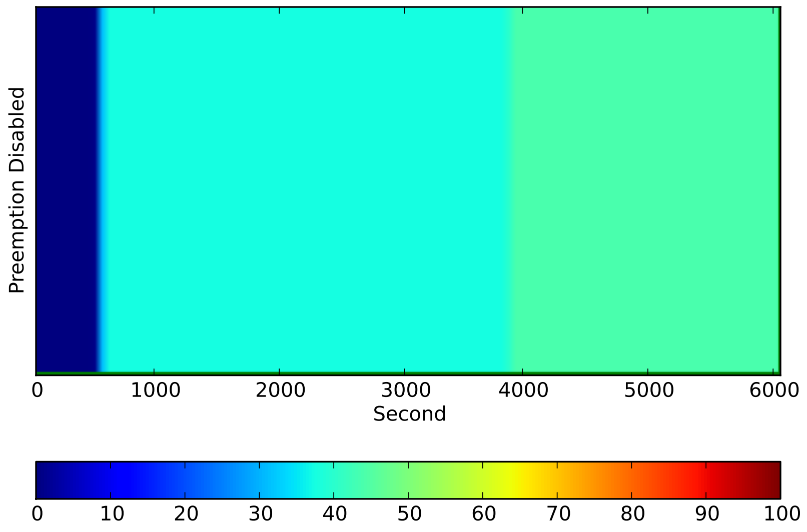

Figure 5.2: Utilization of an IaaS cloud with preemptible VMs disabled.

We compare three scenarios to understand the interactions between preemptible VMs, on-demand VMs, and the HTC workload. The first scenario only considers the on-demand IaaS user workload. In this scenario, the IaaS cloud achieves an average utilization of 36.36%, Figure 5.2, with a minimum utilization of 0% and a maximum of 43.75%. The second scenario considers the HTCondor workload. In this scenario, we execute the HTCondor workload using all 16 VMMs (128 cores) without the on-demand IaaS user workload. The entire workload completes in approximately 84 minutes (5042 seconds) and is shown in Figure 5.4. Finally, in the third scenario, the HTCondor workload is combined with the on-demand IaaS user workload. By combining the two workloads, the HTCondor workload takes an additional 11 minutes and 45 seconds to complete, completing

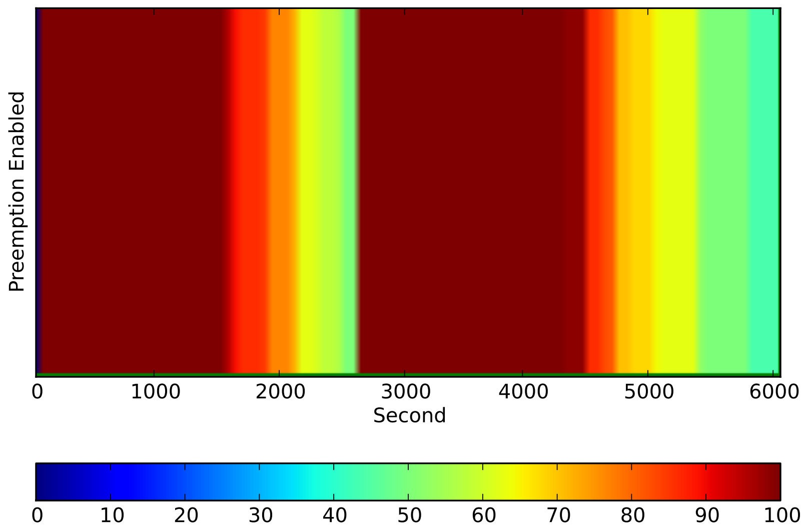

Figure 5.3: Utilization of an IaaS cloud with preemptible VMs enabled.

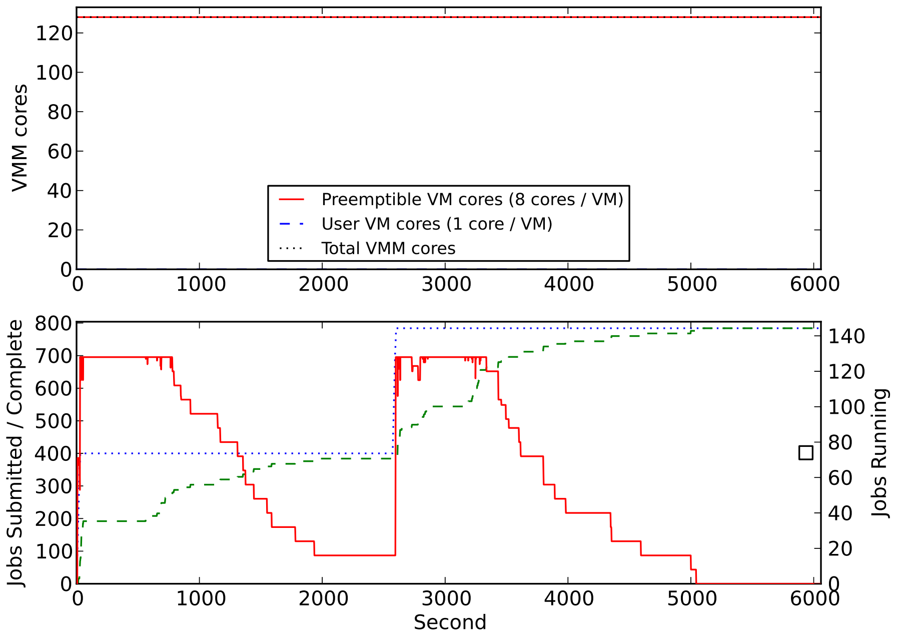

Figure 5.4: Trace of the HTCondor workload executing on all VMM nodes without the IaaS workload.

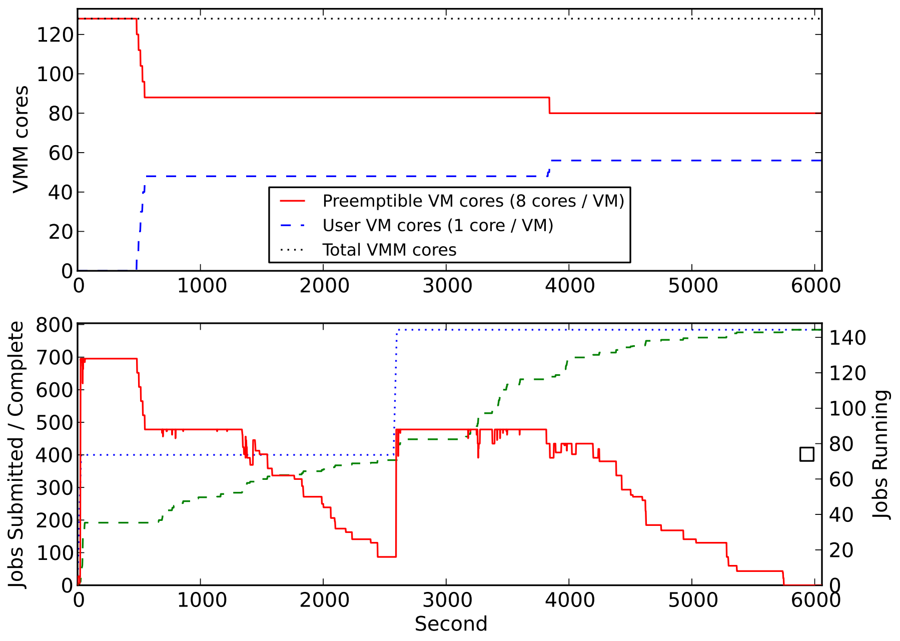

Figure 5.5: Trace of the IaaS workload running on-demand VMs along with the HTCondor workload executing on available preemptible VMs.

in approximately 96 minutes (5747 seconds), as shown in Figure 5.5. At the same time, the utilization of the IaaS cloud, Figure 5.3, increases to an average utilization of 83.82% with a minimum utilization of 0% and a maximum of 100%. The utilization drops just before 6000 seconds into the evaluation because HTCondor jobs begin to complete, and therefore, the IaaS cloud is only running on-demand user VMs.

The utilization of the infrastructure increases significantly without sacrificing the ability to launch on-demand VMs because the IaaS cloud is able to process HTCondor jobs and service user requests for on-demand VMs at the same time. However, the increase in utilization is dependent on the amount of work in the HTC workload, longer and more jobs will translate into higher utilization. However, such jobs may not translate into high efficiency. Long-running HTC jobs are more likely to be interrupted by on-demand VMs, requiring the jobs to be requeued, and possibly, lose all work unless they support checkpointing. Increasing utilization of the IaaS cloud without

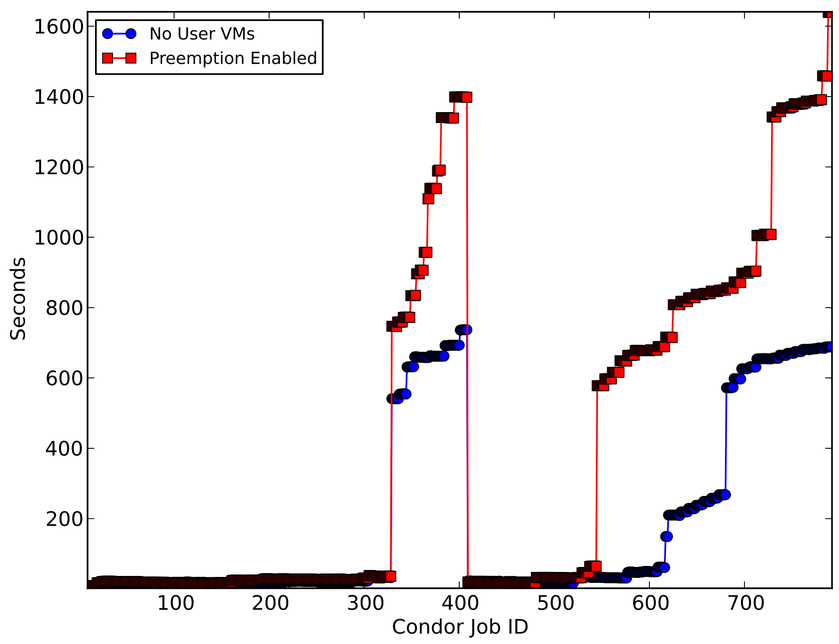

Figure 5.6: Time from when HTCondor jobs are submitted until they first begin executing.

sacrificing the ability to service on-demand requests certainly benefits the cloud provider. Figure 5.5, also demonstrates that preemption is advantageous to HTC workloads, as it is only delays the workload 11 minutes and 45 seconds when combined with on-demand VMs. However, it is presumed that the cost of utilizing preemptible VMs would be lower than dedicated on-demand VMs since the cloud provider can reclaim preemptible VMs at any time and without warning.

5.2.4 Understanding System Performance

To understand how preemption impacts on-demand users and HTC users, we again consider the three scenarios. The first scenario is only the on-demand IaaS workload, the second scenario is only the HTCondor workload running on the 128-core VMM infrastructure, and the third scenario overlays the first two scenarios.

As we would expect, the HTCondor first queued time, that is, the time from when the job is first submitted until it first begins executing, is smallest when HTCondor has exclusive access to

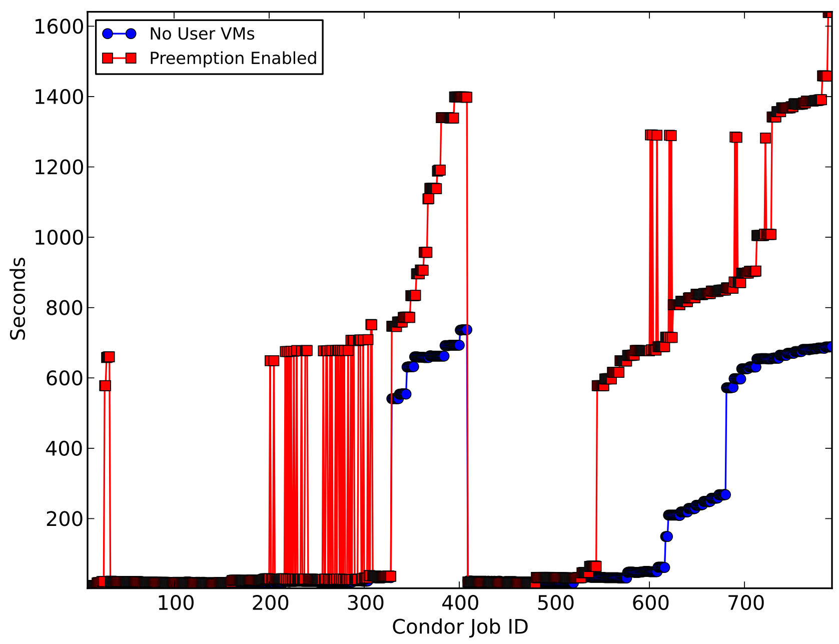

Figure 5.7: Time from when HTCondor jobs are submitted until they begin executing for the last time before completing successfully (e.g., due to rescheduling because VMs were terminated).

the resources, see Figure 5.6. Introducing on-demand, IaaS user VMs increases the HTCondor first queued time for some jobs because there are fewer preemptible VMs processing HTCondor jobs. On-demand VMs also increase the amount of time HTCondor jobs are delayed until finally executing before successful completion, as shown by the 48 spikes in Figure 5.7. These 48 jobs actually first begin executing much earlier, as shown by the lack of spikes in Figure 5.6, however, the jobs are delayed by the arrival of on-demand VMs, which cause preemptible VMs to be terminated and HTCondor jobs to be preempted. Of the 48 preempted jobs, the average delay before executing for the final time is 627 seconds, with a standard deviation of 76.78. The minimum delay is 273 seconds and the maximum is 714 seconds. These 48 jobs spent a total of 22,716 CPU seconds processing the HTCondor workload before they were preempted. The entire HTCondor workload requires a total of 355,245 CPU seconds, thus, for these experimental traces, the use of a preemption-enabled IaaS cloud resulted in 6.39% overhead for the HTCondor workload.

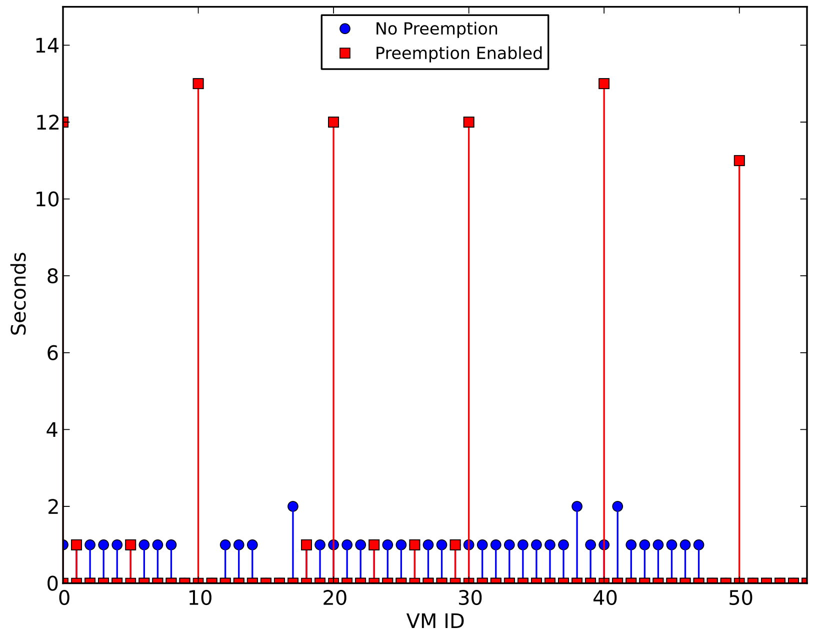

Figure 5.8: Time to process an on-demand request for a VM.

Preemptible VMs also impacts user requests for on-demand VMs, as is shown in Figure 5.8. Without preemption, all on-demand requests are handled in 2 seconds or less. With preemption, the amount of time required to respond to an on-demand request can be as high as 13 seconds when a preemptible VM needs to be terminated. After the preemptible VM is terminated for a single-core on-demand VM, the remaining 7 cores can be used to service on-demand VMs within 2 seconds. The large increase is due to the fact that Nimbus must cleanly shutdown preemptible VMs in order to accommodate on-demand user VMs. Because the evaluation environment consists of 8-core VMM nodes and preemptible VMs use all 8 cores, terminating one preemptible VM actually frees 8 cores for on-demand user VMs, allowing an additional 7 on-demand requests for single-core VMs to be served within seconds.

5.3 Conclusion

The usefulness of preemptible VMs clearly depends on the characteristics of the workloads and the environment configuration, as well as the termination policies employed for preemptible VMs. Our evaluation, which uses representative HTC and IaaS workload traces, demonstrates that a shared infrastructure between IaaS clouds and HTC workload managers can be highly beneficial to both cloud providers and HTC users. Preemptible VMs helps increase utilization of the cloud infrastructure, and therefore decrease overall cost, while also contributing cycles that would otherwise be idle to processing HTC jobs without impacting on-demand IaaS leases. In particular, our representative HTC and IaaS traces show that it is possible to increase utilization of the IaaS cloud from an average utilization of 36.36% to 83.82% while only introducing 6.39% HTC workload overhead and delaying a portion of IaaS on-demand requests by approximately 10 seconds. Preemptible and preset leases also provide the scientific community with a new type of resource lease, allowing them to potentially move their workloads to the cloud for a reduced cost.

Chapter 6

Elastic Environment Implementation and Evaluation

The flexible cloud architecture adapts to changing demand by provisioning IaaS resources as demand increases and relinquishing IaaS resources as demand decreases, creating an elastic environment. In cases where demand is exceptionally high, the elastic environment can outsource demand to community or public IaaS clouds. Much of the elastic environment functionality is contained in three components: a sensor service to monitor demand, a decision engine that executes a policy to respond to demand, and an auto-scaling service capable of launching and terminating VMs on IaaS clouds. This chapter describes our work to create, deploy, and evaluate an elastic environment, originally presented in [95].

6.1 Implementation

The implementation extends existing elastic management services with support for batch-queue clusters and automatically scales the environment based on immediate demand. System and software integration frameworks are used to automate deployment of the elastic environment services, workers, and applications. We also develop app-deploy, a generic tool for deploying complex software stacks that can deploy a wide variety of applications with a single command into a contained environment. The dynamic elastic environments are periodically recontextualized with a REST-based service that exchanges host information, SSH keys, and any other required data. Various file system solutions can be deployed using the recontextualization broker and system integration frameworks in order to meet the demands of different applications and workflows. Finally, the elastic environment is evaluated using NSF FutureGrid and Amazon EC2, demonstrating its scalability and its ability to leverage multiple clouds simultaneously. Figure 6.2 is an example deployment of an elastic environment, consisting of a sensor to monitor the demand of an HPC cluster, a policy to respond to sensor information, and an auto-scaling service to launch and terminate instances on IaaS clouds.

6.1.1 Elastic Management

The initial implementation of elastic management capabilities leveraged components that evolved into the open source Phantom [80] service, which provides a framework to monitor demand, respond to demand, and interface with IaaS clouds. The components use the Advanced Message Queuing Protocol (AMQP) [118] for communication. The three main components are: sensors, a decision engine (to execute a policy), and a provisioner or auto-scaling service to interface with IaaS clouds. However, originally only AMQP-based sensors were supported, which use a "pull" queue model where available workers connect to the queue and request work. Unfortunately, many existing scientific applications and workflows do not currently use AMQP as a message protocol for communication. To address this, we develop a custom sensor that integrates with a widely used cluster resource manager, Torque [72], allowing any Torque-based workflow to leverage elastic environments seamlessly. We also develop a custom decision engine to respond to Torque sensors, which reports information about the jobs in the queue and the status of workers.

The Torque sensor is written in Python and it monitors the Torque queue and sends information to the decision engine. The sensor gathers job information, including the total number of queued jobs and the total cores requested by queued jobs, as well as node information, including a list of Torque worker hostnames and the state of workers (free, busy, offline, down, etc.). The Torque sensor executes commands on behalf of the decision engine, which does not need to run on the same system as the Torque head node. For example, when a new node is assigned an IP at boot, the information must be sent to the Torque server node so it can be added to the cluster. Similarly, when the decision engine elects to terminate workers, it instructs the sensor to mark nodes as offline and remove them from the cluster, etc.

The custom decision engine responds to Torque sensor information by choosing to either launch additional workers, terminate workers, or leave the environment unchanged. Each time the decision engine loops (typically every few seconds), the decision engine performs a number of actions. First, it determines whether or not to launch additional instances based on the number of queued jobs and available workers. For example, if the decision engine implements a relatively basic on-demand policy it would calculate the number of instances to launch as the total number of queued jobs minus the total number of available workers. If the result is a positive integer, the decision engine would notify the provisioner to launch that many instances. Second, after evaluating whether or not to launch instances, the decision engine then instructs the Torque sensor to add any recently launched instances to Torque, mark any idle workers as offline in preparation to terminate them, and terminate any workers it had previously marked as offline. And third, the decision engine also replaces stalled instances. Instances are considered stalled if they have been pending for 10 minutes or more, a value that is configurable by the administrator. However, if even one instance that was pending moves into the running state, the decision engine waits another 10 minutes before attempting to replace stalled instances.

6.1.2 Support for Auto-Scale Services

We also decouple the sensors and policies from individual elastic management components, extending them to support central cloud auto-scaling services via Amazon's auto-scaling API as implemented by open source services such as Phantom [80]. Amazon's auto-scaling API provides an explicit mechanism for creating auto-scaling groups, which define the desired capacity of the environment, that is, the number of instances to deploy as well as minimum and maximum values for the group. The auto-scaling service, such as Amazon Auto Scale or Phantom, continually monitors the instances and ensures that the group never falls below the minimum or exceeds the maximum. For example, a user may create a launch configuration for an auto-scale group that defines the clouds to use. The launch configuration is registered with the auto-scaling service and then the user creates an auto-scale group, defining the number of instances to deploy. The auto-scale service then deploys the instances across the different clouds and continually monitors them, replacing them if they crash. The lightweight sensors and policies are implemented in Python and use boto [5] to communicate with auto-scaling services. In particular, the sensor monitors a Torque queue and provides job and node information to the policy, which evaluates this information, along with various user-defined preferences (e.g., USD to spend per hour), and either launches instances, terminates instances, or leaves the environment unchanged. The Python sensor and policy loop continuously, querying the status of the environment and evaluating demand. Reactive environments should loop frequently while less reactive environments may not need to adapt to demand as frequently. The initial public implementation, Phorque, queries Torque and includes a decision engine that integrates with Phantom for auto-scaling; it is available on GitHub [25].

6.1.3 Automating Service, Worker, and Application Deployment

Base elastic environment instances, which include the elastic management components and Torque server, are deployed via cloudinit.d [44]. Cloudinit.d is an open source tool to orchestrate IaaS cloud deployments. It uses launch plans to specify the number of instances to launch as well as the services to run in those instances. Cloudinit.d is similar to the Linux init.d process; it organizes service deployment into run levels where services in each level must complete before proceeding to the next. In addition to launching instances and deploying services in those instances, cloudinit.d can be used to monitor, repair, and terminate the environment. This allows quick and repeatable environment deployments using a single command.

After the elastic environment launches new instances, workers need to automatically install and configure cluster software as well as user software. This is a crucial component of the elastic environment since workers may join or leave at any time. The Chef [6] systems integration framework is used to download, install, and configure software for the environment. Chef uses "recipes" to automate common system administration tasks, including software installation, starting services, and running bash scripts. Specifically, we develop a set of Chef recipes to download, compile, and install Torque, and other miscellaneous packages required by the environment.

To deploy complex scientific application stacks in a contained environment, we create a generic deployment application, app-deploy. Applications and user data are specified in a simple configuration file. App-deploy then deploys applications into a contained environment, all within a single directory, e.g., /opt/software/. Each application is deployed in its own directory within the specified deploy directory. For example, QIIME might be deployed in /opt/software/qiime-1.4.0/. App-deploy uses threads to download and install applications in parallel. Dependencies can be specified in the configuration file and app-deploy will wait until all dependencies have deployed before attempting to install the applications that depend on them. App-deploy also generates an environment configuration file, activate.sh, that defines information relevant to the environment, such as environment variables. The user can simply "source" the activate.sh script to activate the environment, setting their PATH environment variable correctly for all applications.

The app-deploy configuration file is divided into three sections: global, data, and applications. The global section defines information that includes the final deploy directory for all applications, the log level, and the number of threads. Applications and data are specified by their location (e.g., a URL or repository), the version to deploy, any extra options for GNU autoconf configure or make, any required environment variables to set, dependencies that should be installed first, and finally, the build type of the application. Currently, app-deploy supports the following build types: GNU autoconf, Python distutils, make, make-install, compile an individual C or C++ file, ant, copy (e.g., to copy a precompiled binary to the deploy directory), and custom. Applications that require custom deployments can define their deployment process in a Python method or module, for example, if the application source file needs to be patched before compiling.

Instead of configuring and maintaining worker VM images on each cloud provider, automating the deployment and configuration of worker instances allows us to leverage multiple cloud providers. A base image, common across all cloud providers (e.g., a fresh install of Debian 5.0 Lenny) is selected, and then can be used as a worker on any cloud provider at runtime. However, because downloading, configuring, and installing the entire software stack on each worker is time consuming and introduces possible points of failure, the software can be cached in the VM image, greatly speeding up the deployment process.

6.1.4 Recontextualization

Elastic environments constantly adapt to changes in demand, therefore, instances in the environment need to continually exchange information as they join or leave the environment. To illustrate, cluster instances that use host-based SSH authentication need to exchange host information, including hostnames and IP addresses as well as SSH host keys, with new instances as they join the environment. To handle such events, we propose a recontextualization broker that periodically and securely exchanges host information (short and full hostnames as well as IP addresses), SSH public host keys, and generic text-based data between all instances in a context. The broker is a lightweight, Representation State Transfer (REST), recontextualization service. Unlike other contextualization solutions, such as the Nimbus context broker [82] or Wrangler [78], our recontextualization broker does not use the notion of roles to differentiate between nodes in a context; it exchanges a set of information between all nodes in the context. A recontextualization client allows users to create a new context with the recontextualization broker and a recontextualization agent runs inside instances and exchanges host information with other instances through the broker. All components are written in Python and use REST over HTTPS for communication; symmetric keys are used for both user and context security.

The recontextualization agent sends its information to the broker and receives information about other nodes in the context from the broker. It uses a set of administrator-created scripts (e.g., bash scripts) to define how to handle updates from the broker. For example, when new nodes join an HPC cluster, the environment, their hostname, IP address, and SSH public key need to be added to the ssh_known_hosts file. The scripts are grouped into four categories: initialization scripts that are used by the agent when it first starts, add node scripts that are executed when the agent processes updates for a node that has joined the context, delete node scripts for when the agent processes updates for a node that has left the context, and restart scripts.

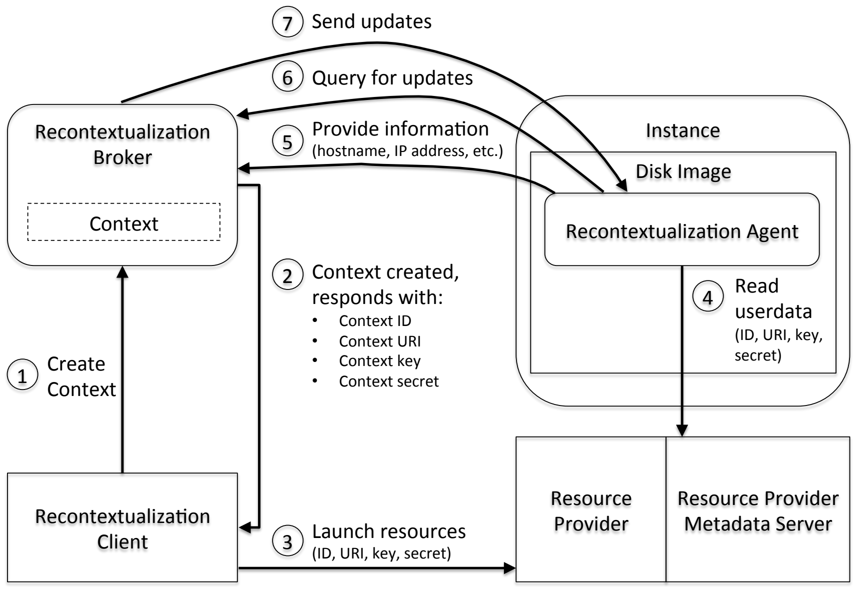

Figure 6.1: The recontextualization process begins when the recontextualization client requests that the broker create a new context. The context information is then passed to instances in the context via the cloud provider's metadata server. The agent reads its userdata field to obtain the context information and then sends its information to the broker in order to join the context. Steps 6 and 7 loop repeatedly as the agent queries for updates to the context.

Recontextualization starts when a user creates a context using the recontextualization client. This sends a request to the broker, which creates the context and responds with a unique context ID, a uniform resource identifier (URI), and a unique context key and secret. As an example, context 1 would have the following URI: https://hostname:port/ctx/1 and the key and secret are random strings unique to the context, allowing only those that know the strings to join the context. When the user launches a cloud instance, this information is passed to the agent inside the instance via the IaaS cloud's userdata field. When the instance boots, the agent starts and reads the instance's userdata field to get the context information; it then sends its information (hostnames, IP address, SSH public host key, and any generic text-based data) to the broker. The broker maintains an ordered list of all nodes that join or leave a context. The list begins with

ID 0, signifying no updates. Each list entry contains all of a node's information and an action specifying whether the node joined or left the context. The agent enters into a loop, referred to as the recontextualization period, after sending its information to the broker. At each loop iteration, the agent requests updates from the broker and processes them by calling its add node scripts, remove node scripts, and restart scripts. To request updates, the agent sends its current ID in the ordered list, beginning at 0, to the broker. The broker compares the ID to the latest list ID for the context and, if they do not match, the broker sends the updated portion of the list to the agent. The agent applies the updates in order by executing its add node scripts for nodes that joined the context and delete node scripts for nodes that left the context, followed by its restart scripts. When the agent executes the add node scripts or the delete node scripts, it passes the host information for the entry in the list to the scripts. After restarting any required services, the agent sleeps for a short time before looping again. The amount of time the agent sleeps determines the reactivity of the recontextualization process; highly reactive environments should loop frequently whereas relatively stable environments should loop less often. Future work will examine the possibility of using a "feedback control" mechanism to automatically adjust the recontextualization period. The entire recontextualization process is shown in Figure 6.1.

It should be noted that while the recontextualization broker is developed specifically for elastic IaaS environments, it could also be used with physical resource deployments (e.g., if cluster nodes need to exchange information periodically to reconfigure themselves). Additionally, it does not currently have a mechanism to "inject" the agent into a VM image, which would provide a fully automated recontextualization process; instead, the agent must be installed in the image by an administrator and configured to start at boot.

6.1.5 Support for Shared Data Access

Because the elastic environment must support a common scientific workflow patterns, a reliable shared file system is needed. Individual cluster resource managers, such as Torque, may include capabilities to transfer input data to worker instances before jobs begin executing and

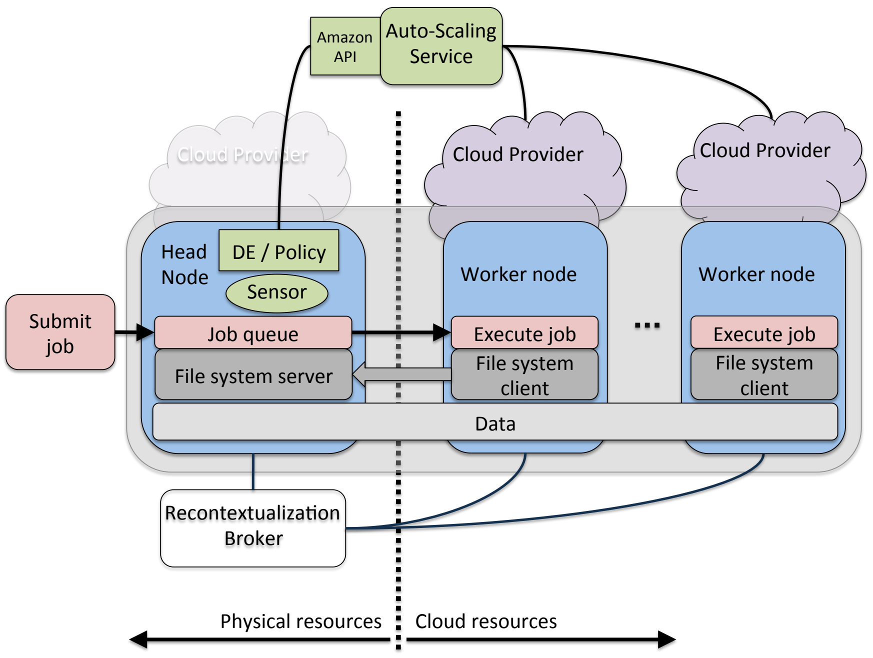

Figure 6.2: An example elastic environment deployment with support for shared data access.

output data once jobs complete. However, this prevents workers from using a shared file system, which many existing scientific workflows assume. Therefore, our current implementation can deploy shared file systems, such as NFS, using the recontextualization broker and system integration frameworks. These solutions enable administrators to configure the deployment for their specific needs, allowing any number of file system solutions to be deployed. To date, the elastic environment implementation natively supports NFS [110], XtreemFS [76], and Gluster [13] and it includes a set of scripts to download, configure, and install these file systems on both servers and clients. For the initial implementation, the file system server is only deployed on the cluster head node, and it is not replicated across multiple nodes. The file system server exports a portion of the underlying head node file system, providing a single mount point and shared file system for all clients in the environment, as shown in Figure 6.2. However, in previous deployments, we found that NFS often fails in wide-area environments, such as multi-cloud environments that operate across the Internet and therefore recommend that NFS only be used for deployments within a single cloud. Gluster and XtreemFS have proven to be relatively reliable distributed and scalable file systems for both single- and multi-cloud deployments. Users can access the Gluster file system via NFS and SMB protocols as well as Gluster's native GlusterFS protocol.

6.2 Evaluation

We evaluate the ability of our elastic environment to react to changes in demand, recontextualize as the environment adapts, and scale appropriately. The performance of the underlying cloud provider hardware, which is tied directly to the performance of the underlying node, network, and storage performance, as well as the software configuration of the deployment, is not examined. Other studies have evaluated the performance of virtualization technologies and IaaS cloud provider solutions [64], [102], [71], [66], [75]. Users with strict performance requirements should select the appropriate cloud hardware, such as Amazon EC2's cluster compute instances, which ranked #42 on the November 2010 Top500 [19].

The evaluation environment consists of deployments using NSF FutureGrid and Amazon EC2. On FutureGrid, the Hotel cloud (fg-hotel), located at the University of Chicago (UC), and Sierra (fg-sierra), located at the San Diego Supercomputer Center (SDSC) are used. Both clouds run Xen [42] for virtualization. The recontextualization broker and cluster head node are deployed in separate VMs on Hotel for all tests. The VM for the recontextualization broker consists of eight 2.93 GHz Xeon cores and 16 GB of RAM; the cluster head node VM has two 2.93 GHz Xeon cores and 2 GB of RAM. The cluster uses Torque 2.5.9 and Maui 3.3.1 [77], and also runs the elastic management components, including a sensor, decision engine, and provisioner. Deployment and configuration of the head node is completely automated using cloudinit.d. Worker nodes are deployed on both Hotel and Sierra and consists of two 2.93 GHz Xeon cores and two GB of RAM. Evaluations involving EC2 use 64-bit EC2 east micro instances as workers, primarily because micro instances are inexpensive, at only two cents per hour. Micro instances use Elastic Block Storage (EBS) for disk and contain up to two EC2 compute units with 613 MB of RAM. (An EC2 compute unit is defined as the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon.)

Deployment of worker nodes is completely automated. As part of the contextualization phase, worker nodes mount /scratch from the head node using NFS.

For all evaluations, the sensor is configured to query Torque every 60 seconds to gather job and worker information. The decision engine executes every 5 seconds, querying IaaS clouds for changes in instance state and executing the policy to adjust based on demand. Recontextualization agents send their information to the broker when they first start and then query the broker for updates every 120 seconds. We believe that a 120-second recontextualization period is appropriate for scientific batch-queue workloads, which typically contain jobs that run for hours, or possibly days since the environment won't need to be recontextualized more frequently than every few minutes, at a minimum.

6.2.1 Metrics

We define the metric *reactivity time* to be the time from when the first job is submitted until the time the last job begins running for a group of jobs submitted together. For MPI jobs, enough cores must be available to run all tasks for all jobs. That is, if a single 128-task MPI job is submitted to a cluster that deploys single-core IaaS nodes, 128 nodes must launch and join the cluster before the job will run. We define the metric *recontextualization time* to be the time from when a new node attempts to join a context until the time all nodes in the context receive and apply the updates for that node. For example, if 256 nodes are running and the context is stable (all 256 nodes are aware of each other and have exchanged host information), and then a new node attempts to join the context by sending its information to the broker, the recontextualization time is the amount of time from when the new node sends its information until the time when all 256 nodes have executed their scripts to add the new node to the context. We also examine the ability of the elastic environment to deploy instances across multiple clouds simultaneously and scale quickly. For these evaluations we use a series of traces showing workload information (jobs submitted, queued, running, complete) and VM information (VMs running and cores available).

6.2.2 Workloads

The workload for the reactivity and recontextualization tests consist of a simple MPI application that sleeps for an hour. In the reactivity tests, we only measure the time it takes until the job begins executing, and then we gather the required logs and terminate the environment. For the recontextualization test, one hour is more than enough time for the initial nodes to boot and for the context to stabilize, and then, for an additional node to boot and join the context. Once the new node joins the context and all other nodes apply their updates, we gather the required logs and terminate the environment. In the case of the multi-cloud and scalability tests, we examine the ability of the environment to scale up as demand increases and scale down as demand decreases. Therefore, the workload consists of individual single-core "sleep" jobs that sleep for 30 minutes.

6.2.3 Understanding System Responsiveness

To understand system responsiveness we consider the amount of time it takes the environment to react to changes in demand, or reactivity time, and we also consider the amount of time it takes a context, or group of nodes, to recontextualize when the context is updated, referred to as the recontextualization time. The reactivity time is the amount of time it takes a newly submitted job or group of jobs to begin running. This includes the time it takes the environment to detect the change in demand, execute the policy to respond to demand, request instances from an IaaS provider (if applicable, as determined by the policy), and wait for the instances to boot and join the cluster.

To measure reactivity time, we submit MPI sleep jobs, causing the environment to launch the needed number of nodes and begin running the job. We used EC2 east micro instances (as single-core workers), and performed a series of tests, beginning with 2 node clusters and increasing to 256, shown in Figure 6.3. We ran the test three times for each cluster size and between each test we allowed the environment to terminate all worker nodes and return to an idle state with an empty job queue.

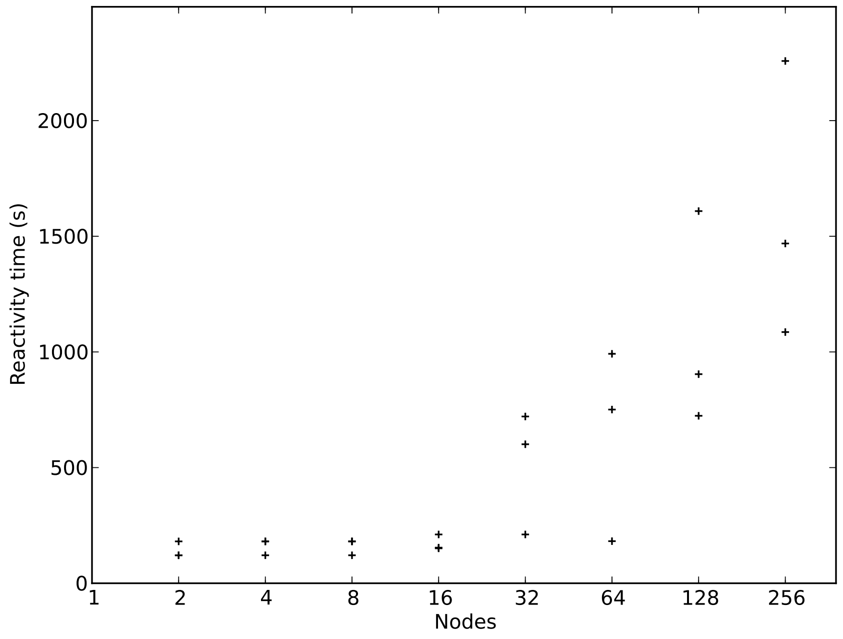

Figure 6.3: Reactivity time for 2-node clusters through 256-node clusters (3 tests for each cluster size).

Interestingly, small cluster sizes, those from 2 nodes through 16 nodes, all have relatively similar reactivity times. However, larger cluster sizes have increasing reactivity times. In particular, 32- and 64-node clusters each have one test with a reactivity time similar to smaller clusters while the other two tests have much higher reactivity times. This can be explained by the fact that the decision engine replaces stalled instances after 10 minutes. We observed that EC2 micro instances would periodically fail in our evaluation; the instances would most often fail to boot completely or access the network. As we deployed larger clusters, the chances of encountering a failed instance increased as we can see by the increasing reactivity times for larger clusters. This isn't to say that smaller clusters won't encounter failed instances occasionally, only that we did not encounter any in our evaluation.

To measure recontextualization time, we submit MPI sleep jobs that request the appropriate number of cores and allows the cluster to boot, contextualize, and start running the job. Once the job begins running, we submit another single-core sleep job, causing the elastic environment to launch an extra instance. Once the extra node booted and joined the cluster, we measured the time from when the new instance sent its information to the broker until all other nodes receive and apply the host information for the new node. We rely on Amazon's ability to synchronize the clocks of instances, which is done at the VMM layer using NTP and passed to the VMs. Again, we used EC2 micro instances, from 2 node clusters through 256 node clusters, running three tests for each cluster size.

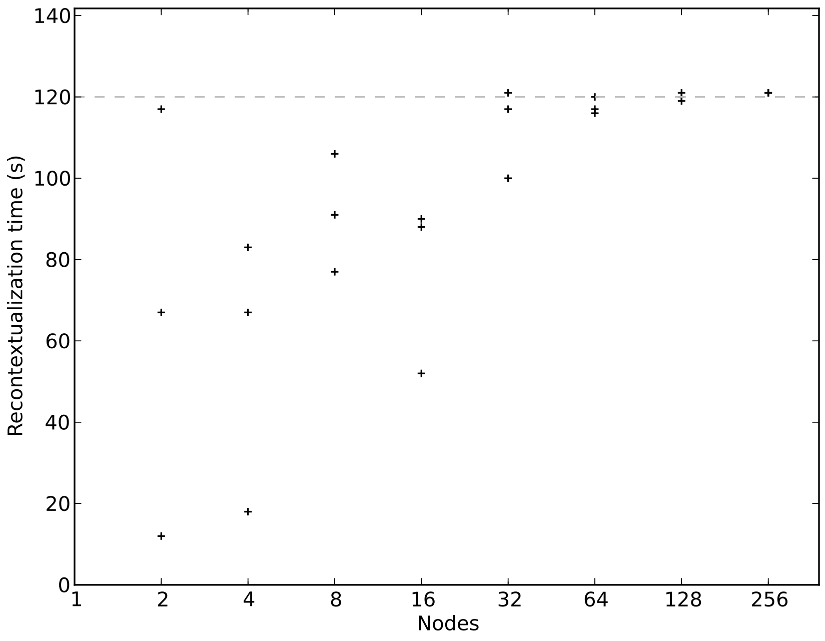

Figure 6.4: Recontextualization time for 2-node clusters through 256-node clusters (3 tests for each cluster size). Those showing fewer than 3 points are cases where values overlap.

As shown in Figure 6.4, all of the clusters are fully able to recontextualize within 1 second of the 120-second recontextualization period. Those tests that fully recontextualize before the 120 second period do so because all of the nodes in the environment check in with the broker shortly after the additional node sends its information to the broker. This is the result of the random timing when nodes boot and install their software. The 2-node cluster tests demonstrate this characteristic clearly. For one of the tests, the cluster recontextualizes quickly while for another test it takes almost two minutes for both nodes to recontextualize. As the number of nodes in the context increase, the likelihood that one or more nodes will require the full 120 seconds (after the new node joins the context) to check in with the broker increases, as shown by the three 256-node tests that fully recontextualize one second after the 120-second recontextualization period.

6.2.4 Multi-Cloud and Scalable Elastic Computing

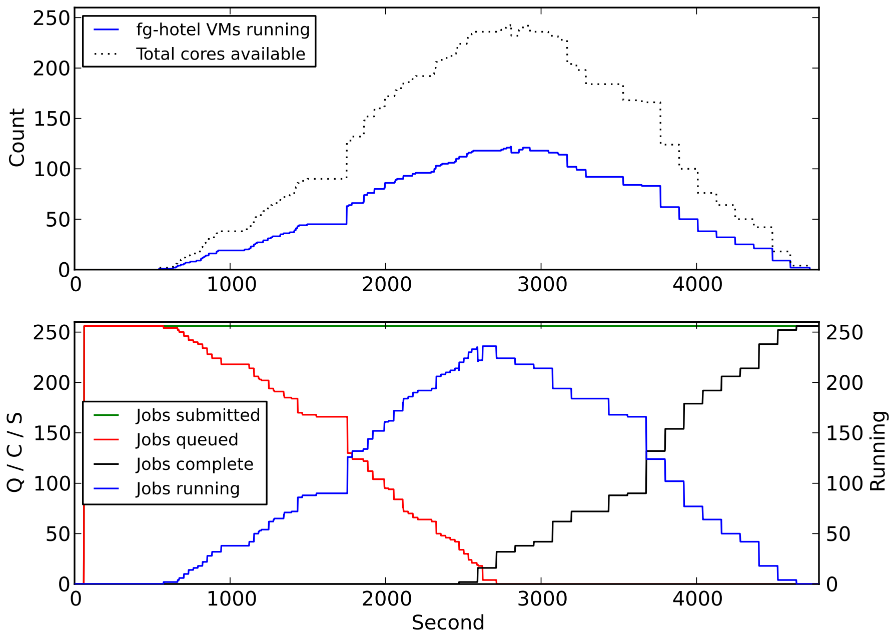

Figure 6.5: Trace of a deployment running 256 30-minute “sleep” jobs on Hotel.

To examine the ability of the elastic environment to scale as demand fluctuates and leverage multiple clouds simultaneously, we submit workloads containing 30-minute single-core “sleep” jobs that run to completion. For the first test, we only deploy workers on Hotel, submitting 256 jobs, as shown in Figure 6.5. Each Hotel worker has 2 cores and 2 GB of RAM. The elastic environment scales to over 100 VMs but it doesn’t quite reach 128 VMs (or 256 cores) within 30 minutes due to the underlying storage, network, and node performance of Hotel, which is unable to deploy 128

VMs in 30 minutes. As the first jobs complete and the instances become available, those instances are able to start processing the remaining queued jobs, stopping the environment from completely deploying the final instances. Idle instances are terminated once the queue is empty.

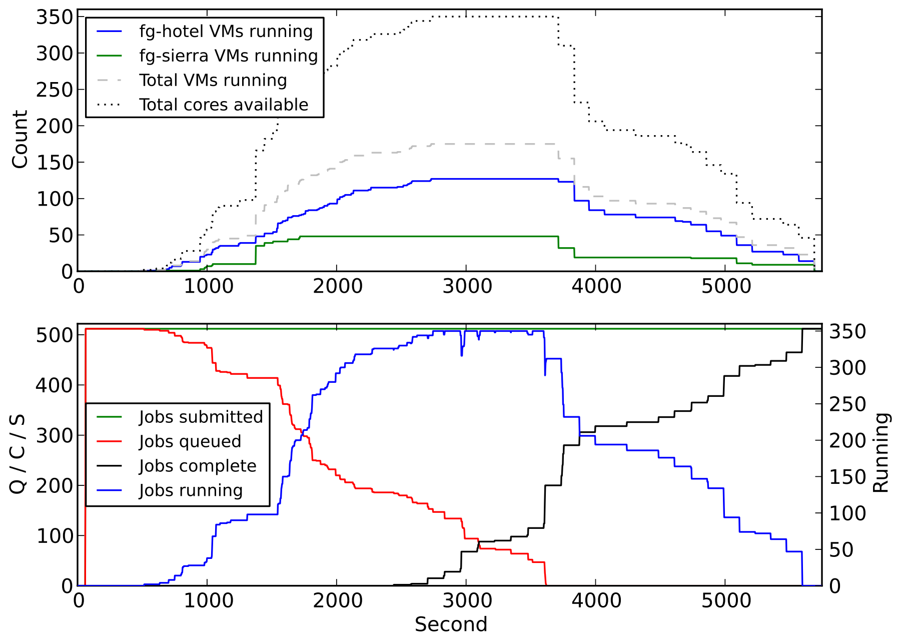

Figure 6.6: Trace of a multi-cloud deployment running 512 30 minute “sleep” jobs on Hotel and Sierra.

For the second test, workers are deployed on both Hotel and Sierra and we submit 512 jobs, shown in Figure 6.6. Again, each worker contains 2 cores and 2 GB of RAM. The workers from both Hotel and Sierra are configured to trust the head node as well as each other. In this test we observe the limited scalability of private clouds as Hotel and Sierra are unable to scale to 512 cores. Both clouds reach the maximum number of instances that they can deploy, shown by the horizontal line for VMs running on each cloud (which occurs at 2,000 seconds for Sierra and 3,000 seconds for Hotel). The scalability of clouds is limited by the amount of hardware available and also by the number of other users running on the cloud at the time. Similar to the previous test, the environment scales down as jobs complete, terminating free instances.

The final test examines the scalability of the environment using EC2 east micro instances,

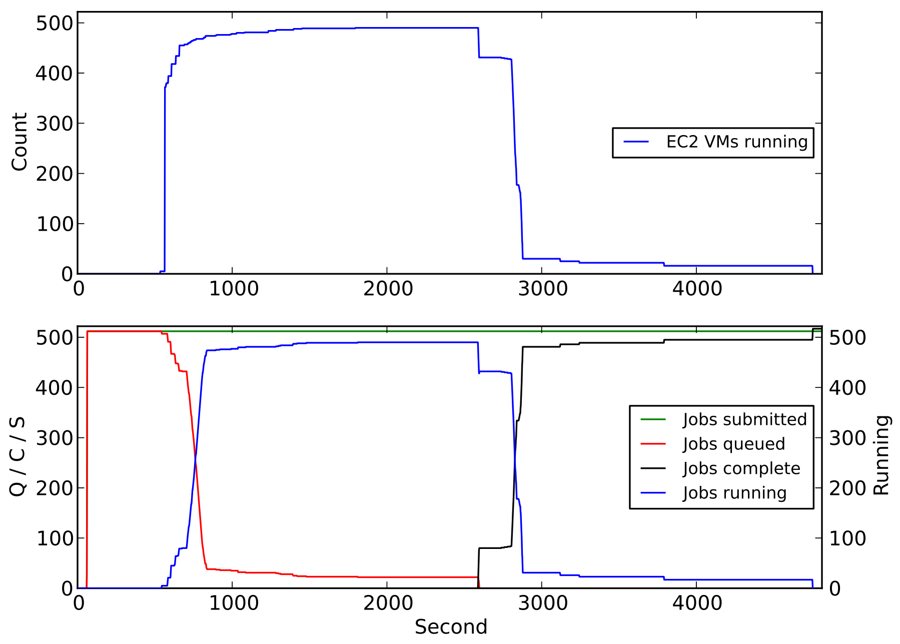

Figure 6.7: Trace of a scalability test using Amazon EC2 and 512 30 minute “sleep” jobs.

shown in Figure 6.7. We submit 512 single-core sleep jobs at one time. Though the environment scales to 476 instances in less than 15 minutes, it is not able to reach 512 instances within the first 30 minutes of the evaluation, only being able to reach a total of 490 instances. The remaining 22 instances fail to boot completely on EC2, even after the decision engine detects stalled instances and tries replacing them, multiple times. The reason for this is not entirely known. However, we did not observe any problems with the elastic management components or recontextualization broker. It is worth noting that while inexpensive single-core micro instances are used for the scalability test, the entire evaluation is run on a per-node basis. Thus, the 490-node scalability test would result in a 5,880-core cluster if 12-core worker nodes were deployed. However, a 512-node cluster on Amazon EC2, using their largest cluster instance size, cc2.8xlarge, would cost $1,228.80 per hour compared to $10.24 per hour for a 512-node cluster of micro instances at the time of this writing. Additionally, it should be noted that the key difference in the ability of EC2 to scale quickly, compared to the FutureGrid tests, is due to the performance and configuration of the underlying node, storage, and network differences between the infrastructures. For example, FutureGrid Nimbus clouds used in this evaluation contain a single VM image repository and require that the VM image be transferred to worker nodes before booting, creating contention at the VM image repository and on the network as the image is copied to all nodes. In contrast, the EC2 micro instances used EBS-backed images that can begin booting immediately once the volume is created for the instance and do not require a VM image to be fetched from an image storage service.

6.3 Conclusion

Large-scale elastic environments provide a reactive environment capable of adapting to variable demand and VMs provided by IaaS clouds allow users to install and configure complex software stacks for their workflows. Multi-cloud elastic environments, in their current implementation, are better suited for embarrassingly parallel workloads or loosely-coupled parallel jobs because they typically must use the Internet for a network between clouds, possibly resulting in unpredictable and high-latency communication performance. However, workflows that are not amenable to multi-cloud environments (e.g., applications with strict latency requirements) should be restricted to individual cloud infrastructures. Such restrictions can be enforced through node attributes or multiple queues with routing, etc. RPs can leverage elastic environments to outsource excess demand to external clouds when needed, giving them the flexibility to purchase smaller resources that meet the needs of their users a majority of the time and budget for future outsourcing costs. Fully-automated elastic environments also allow RPs or users to deploy standalone large-scale computing and analysis environments in the cloud, helping to democratize IT by allowing them to access significant resources for their research that they might not otherwise be able to access.

Chapter 7

Resource Provisioning Policies for Elastic Environments

This chapter presents and evaluates a set of resource provisioning policies to respond to variable demand in elastic environments. These policies, originally proposed in [94], balance often-conflicting requirements of users and administrators, such as minimizing the cost of deployments and reducing job queued time. Elastic environments, and resource provisioning policies, allow RPs to adjust to high demand by outsourcing demand to external clouds, such as free community clouds or for-pay public clouds. In this work we consider the scenario where a resource provider extends existing local resources with IaaS clouds on a fixed budget. We consider the following use case: a research lab at a university with a small cluster may occasionally need more capacity than they purchased in capital equipment. The lab specifies a fixed hourly budget, for example, $5 per hour, which can be used to outsource excess demand to IaaS clouds. This money may accumulate so if the lab doesn't deploy IaaS instances over a three-hour period, $15 can be used toward their next IaaS deployment.

The most straightforward resource provisioning policy is to deploy the maximum number of instances allowed by the budget and leave them running at all times. However, such a naive policy may not be the most efficient use of resources or money. Therefore, we propose basic flexible resource provisioning polices that adapt immediately to demand as well as advanced policies that balance user- and administrator-defined requirements, one of which also evaluates data transfer information between the local resource and external clouds, if provided. For comparison purposes, we also consider the cases where only the local cluster is used as well as only using free resources and the for-pay public cloud provider. Finally, we develop a discrete event simulator, the elastic cloud simulator (ECS), to evaluate our policies using scientific workload traces.

7.1 Resource Provisioning Policies

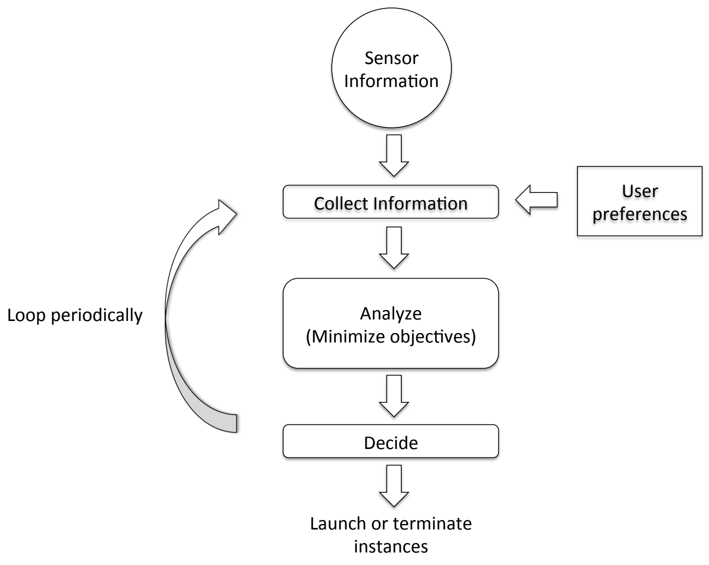

Figure 7.1: The policy execution process begins with collecting sensor information, executing the policy and minimizing objectives, electing to launch or terminate instances, and then looping after a set amount of time.

A decision engine loops periodically and executes a resource provisioning policies to evaluate demand, responding by either provisioning instances, terminating instances, or leaving the environment unchanged, as shown in Figure 7.1. In addition to collecting information about workload demand, provisioning policies may also include user or administrator provided information, such as a budget or deadline information. The basic provisioning policies consist of policies that respond immediately to demand but don't attempt to consider user or administrator requirements, maximum provisioning policies that use the maximum budget, and advanced resource provisioning policies that balance user and administrator requirements. We also compare our policies to using the local cluster, referred to as the *Local* policy. They are summarized in Table 7.1.

Table 7.1: Basic, maximum, and advance resource provisioning policies for elastic environments that adapt to variable demand.

Basic Provisioning Policies			
Policy	Configurable	Minimizes monetary costs	Includes Data
OD	No	No	No
OD++	No	No	No
ODFree	No	Yes	No
Maximum Provisioning Policies			
Policy	Configurable	Minimizes monetary costs	Includes Data
SM	No	No	No
Max	No	No	No
Public	No	No	No
Advanced Provisioning Policies			
Policy	Configurable	Minimizes monetary costs	Includes Data
ETR	No	Yes	Yes
FC	Yes	Yes	No
MCOP	Yes	Yes	No

7.1.1 Basic Provisioning Policies

We propose the following basic resource provisioning policies that respond immediately to changes in demand:

- *On-demand* (OD): The on-demand policy launches instances for all cores requested by jobs in the queued state. It stops launching instances once it has launched the requested number of instances to accommodate the request, used the allowable budget, or reached the maximum number of instances allowed by the cloud provider. On-demand begins with the least expensive cloud first before moving on to more expensive clouds, and terminates instances when there are no more queued jobs and the instances are idle.

- *On-demand++* (OD++): On-demand++ is similar to on-demand in that it launches instances for all cores of queued jobs, beginning with the least expensive cloud. The primary difference between OD and OD++ is that OD++ only terminates idle instances that will be charged by the cloud provider before the next policy evaluation iteration (and once there are no remaining queued jobs). In other words, if a cloud provider chargers instances on an hourly basis, and if the policy evaluation iteration is every 5 minutes, then OD++ will terminate idle instances that will be charged before the next policy evaluation executes (e.g., 55 minutes into the hour).

- *On-demand Free* (ODFree): On-demand free is similar to OD, except that it only uses free resources (e.g., a local cluster and a shared community cloud that does not charge for use); it does not attempt to provision resources on for-pay clouds.

Basic policies that respond immediately to demand, such as OD and OD++, are undoubtedly acceptable to users who are only concerned with reducing job turnaround time and not the cost of the deployment. Unfortunately, because the cluster administrator can't adjust these policies to meet the needs of shared cluster environments, they may not create optimal elastic environments for users.

7.1.2 Maximum Provisioning Policies

We propose the following maximum provisioning policies that use the entire budget, if provided, or ignore it completely:

- *Sustained Max* (SM): Sustained max immediately launches the maximum number of instances allowed by the cloud provider or the budget and leaves them running at all times. If multiple clouds are available, it first launches the maximum on the least expensive cloud before moving on to the next cloud until the budget is completely used.

- *No Budget* (Max): is similar to OD++ in that it provisions instances for all queued jobs and terminates idle instances before they are charged again. However, if a budget has been specified by the administrator the policy ignores the budget and provisions as many instances as needed.

- *Only For-Pay Public Providers* (Public): is similar to Max, except that it only uses for-pay public cloud providers and does not use the local cluster or free clouds, even if available.

It ignores budget information, even if provided. This policy is included primarily for comparison purposes to demonstrate the impact of only using for-pay cloud providers (e.g., Amazon EC2).

7.1.3 Advanced Provisioning Policies

Basic provisioning policies only respond immediately to demand by launching enough instances to process queued jobs while maximum policies use the entire budget to launch the maximum number of instances possible or ignore the budget altogether. In this section we present three advanced provisioning policies that attempt to balance user and administrator defined requirements, including minimizing job queued time while also considering the cost of the deployment. One policy, estimated time remaining, also uses data transfer information, if provided, in its decision process.

7.1.3.1 Estimated Time Remaining

Estimated time remaining (ETR) provisions instances for queued jobs on for-cost clouds if the estimated time required to launch the instances and transfer input data is less than the estimated time remaining for currently running jobs to complete and transfer their output data. It immediately deploys instances on all free resources (e.g., no-cost community clouds). The estimated time remaining is calculated as the difference between requested walltime and the time a job has already been running. For example, if a running job has a walltime of 2 hours, but it has already been running for 90 minutes and it is determined that it will take longer than 30 minutes to launch instances and transfer data for the next queued job, then ETR will wait for the current job to complete instead of provisioning additional instances. If the job doesn't include estimated data sizes, then the policy estimates the time required to launch instances for the job.

7.1.3.2 Average Queued Time Policy

The *average queued time policy* (FC) allows an administrator to define the minimum and maximum number of jobs that the policy should respond to as well as a desired response, r, which is considered a reasonable average weighted queued time (AWQT) for the current set of queued jobs, Q. FC launches instances for the first n queued jobs at each policy evaluation iteration where AWQT is described by the following formula:

$$AWQT = \frac{\sum_{j}^{Q} j.num\_cores * j.queued\_time}{\sum_{j}^{Q} j.num\_cores}$$

Additionally, the administrator may also optionally define a threshold value, θ, for the desired response. If the measured AWQT is less than $r - \theta$, at the policy evaluation iteration, the policy subtracts one from the number of jobs it responds to for the next policy evaluation iteration until the desired minimum is reached. If the measured AWQT is greater than $r + \theta$, then the policy adds one to the number of jobs it will respond to for the next policy evaluation iteration until the maximum is reached. For example, if an administrator determines that r=2 hours is an appropriate desired response rate with a threshold of 45 minutes, then when the policy measures AWQT to be less than 1 hour and 15 minutes it will subtract 1 from the number of jobs it responds to at each policy evaluation iteration, and when it measures greater than 2 hours and 45 minutes it will add 1. If it measures AWQT between those two values then it will respond to the same number of jobs as the previous policy evaluation iteration. After determining the number of jobs to respond to, $\hat{n}$, the policy then selects the maximum number of cloud providers it will use by evaluating:

$$NC = max(\lfloor \frac{AWQT}{r} \rfloor, 1)$$

After calculating NC and $\hat{n}$, the policy then proceeds to launch instances for each job in $\hat{n}$ across the least expensive NC clouds until instances have been launched for all jobs, the budget has been spent, or the limit on the cloud provider's number of allowable instances has been reached.

However, the policy will only launch the appropriate number of instances as determined by the requested core counts for the $\hat{n}$ queued jobs. For example, if the policy determines that it can launch 17 single-core instances based on the cost of cloud providers and available allocation credits, but there are only two 16-core jobs, then the policy will only launch 16 instances, as the 17th instance can not be used. The final action of FC is to terminate instances that will be charged before the next policy evaluation iteration, which is the same termination process as in OD++.

7.1.3.3 Multi-Cloud Optimization Policy

A major concern for users is reducing the amount of time their jobs are queued, and at the same time, a major concern for elastic environment administrators is to minimize the cost of the deployment while still minimizing job queued time. Minimizing these two conflicting objectives is a multi-objective optimization problem [57]. The *multi-cloud optimization policy* (MCOP) attempts to balance the requirements of both users and administrators. The work in [114] uses a genetic algorithm (GA) to schedule jobs across static Grid slots and served as the motivation for MCOP, which provisions and relinquishes resources on IaaS clouds and uses a GA to balance cost and job queued time.

Finding a solution requires searching as many configurations as possible, an expensive task depending on the number of queued jobs, the number of cloud providers available, and the amount of allocation credits. Since searching all possible configurations may be intractable given the limited time available at an individual policy evaluation iteration. MCOP employs a GA to search as many configurations as possible [56]. Due to the time constraints, we do not allow the GA to run until it converges, instead, we only allow the GA to execute a set number of iterations to ensure the policy decides within a reasonable amount of time. The number of iterations can be configured by an administrator depending on their needs. We realize this is not ideal, however, we believe that allowing the GA to explore a sufficient number of possible configuration will result in a better solution than simply selecting one at random. In addition to the GA, the administrator also defines two weights representing their preference for minimizing cost or job queued time.

MCOP is similar to FC in that it only considers configurations that are relevant to the core counts of queued jobs. The GA setup is as follows. Alleles in the chromosome represent individual queued jobs where a 1 signifies that a job will be considered and a 0 does not. The length of the chromosome is the maximum number of queued jobs for the independent policy evaluation iteration. We initially generate a population of 30 individuals for each cloud, that is, each MCOP considers various combinations of jobs for each available cloud. The GA is initialized with common values, generally known to perform well [69], specifically, the default GA iterates 20 times with a mutate probability of 0.031 and a crossover probability of 0.8, although these values can be changed as needed for different workloads and configurations.

MCOP's execution is similar to the standard execution of a GA. After randomly generating the initial populations, the GA then iterates 20 times for each cloud provider's populations, performing crossover and mutation based on the probability. The GA also considers the extreme cases, that is, no jobs (all zeros) and all jobs (all ones) for each cloud provider. The fitness of an individual is calculated as the weighted preference (using the administrator-defined weights) of the estimated cost and job turnaround time for the configuration; those with the lowest cost mate to produce offspring.

After all iterations complete, the final populations for each cloud provider are compared against the final populations of all other cloud providers. The cost of the configuration and job queued time are estimated by building a FIFO schedule of the jobs across the environment. Depending on the number of available cloud providers, the administrator may wish to only compare a subset of the final populations. Based on these comparisons, we generate the Pareto optimal set of solutions, using domination [56] to select the optimal solutions. Recall that domination states that one configuration dominates another if both of the following conditions are true:

(1) The cost of the first configuration is *less than or equal* to the cost of the second configuration **and** the total job queued time for the first configuration is *less than or equal* to the total job queued time for the second configuration.

(2) The cost of the first configuration is *less than* the cost of the second configuration **or** the total queued job time is *less than* the cost of the second configuration.

All configurations that are not dominated by any other configurations are added to the Pareto optimal set. To select the final solution from this set, the administrator-defined weights are multiplied by normalized values for the cost and job queued time for each configuration. The solution with the smallest value for the sum of these two items is selected. If more than one configuration is a minimum, MCOP selects the least expensive one. If two or more minimums also have the same cost, the solution is randomly chosen between them. To terminate instances, MCOP follows the same process of FC and OD++, terminating instances that will be charged before the next policy evaluation iteration.

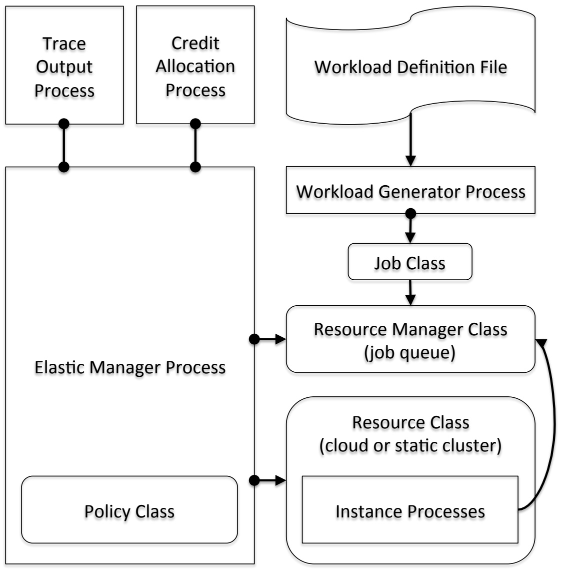
Trace Output Process
Credit Allocation Process
Workload Definition File
Workload Generator Process
Job Class
Resource Manager Class (job queue)
Elastic Manager Process
Resource Class (cloud or static cluster)
Policy Class
Instance Processes

Figure 7.2: Elastic Cloud Simulator architecture.

7.2 Elastic Cloud Simulator

Evaluating resource provisioning policies for IaaS clouds can require a substantial, and possibly prohibitive, investment in time and money. Cost is especially a concern when evaluating

Table 7.2: Summary of Amazon EC2 east launch and termination times for a 10 GB Debian 5.0 image measured over the course of a day, transferred from Amazon S3.

EC2 East Launch Times (small instance)		
Probability	Mean	σ
63%	50.86s	1.91
25%	42.34s	2.56
12%	60.69s	2.14
EC2 East Termination Times (small instance)		
Probability	Mean	σ
100%	12.92s	0.50

commercial cloud providers, such as Amazon EC2, which charge with real currency. Therefore, to evaluate our resource provisioning policies we develop a discrete event simulator, the elastic cloud simulator (ECS), which simulates all of the necessary components of the elastic environment. This includes workload generation, workload submission, launching cloud instances, processing the workload, terminating cloud instances, and accounting for allocation credits, shown in Figure 7.2. Furthermore, ECS simulates static physical resource deployments, private cloud infrastructures with limited scalability, and commercial cloud providers that provide the appearance of infinite scalability but charge for usage.

7.2.1 Measuring Cloud Variability

For simulation, we consider two main sources of variability in IaaS clouds: instance launch and termination times. To measure instance launch and termination times, we launched and then terminated 60 Debian 5.0 Linux instances (with a 10 GB VM image stored in Amazon S3) on EC2 east over the course of a day. After launching the instance, we began pinging it until we received a successful response. We calculate the launch time as the difference between the time we sent the launch request and the time of the first successful ping. To calculate the termination time, we terminate the instance and then ping it until the ping fails. We calculate the termination time as the difference between the time we sent the termination request until the time the first ping fails.

Launch and termination times are summarized in Table 7.2. Termination times appeared relatively consistent with an average time of 12.92 seconds and a standard deviation of 0.50. However, launch times experienced much more variability. Launch times appeared to group around three values; the majority of launches, 63 percent, averaged 50.86 seconds with a standard deviation of 1.91, 25 percent averaged 42.34 seconds with a standard deviation of 2.56 seconds, and 12 percent average 60.69 seconds with a standard deviation of 2.14. ECS uses these values to randomly select launch and termination times for IaaS instances.

7.2.2 Elastic Cloud Simulator Implementation

ECS is written in Python and uses SimPy [28] for simulation; the architecture is shown in Figure 19. It can either use a workload definition file specifying job arrival and run times or automatically generate and format a workload based on the Feitelson workload model [61], which provides a comprehensive HPC workload model that is representative of cluster workloads. ECS processes jobs in first-in-first-out (FIFO) order, scheduling jobs on simulated resources in the order that they are submitted. ECS can simulate static physical resource deployments, such as a batch-queue cluster, as well as dynamic IaaS clouds. Parallel jobs are only scheduled on individual resource infrastructures; they are not allowed to span multiple infrastructures due to the latency requirements of many parallel jobs and the fact that different cloud providers are often only connected via the Internet. ECS consists of a workload generator process, an elastic manager process, any number of instance processes (each representing individual worker instances), a credit allocation process that keeps track of available credits, and a trace output process that provides information about the environment. Each process loops continually for the duration of the simulation. ECS policies are implemented as individual Python modules and are completely interchangeable.

7.3 Evaluation of Resource Provisioning Policies

For evaluation, we compare the policies described in Section 7.1 and summarized in Table 7.1. The evaluation considers the use case described earlier, that is, a small research lab extends

Table 7.3: Policy evaluation workloads, including a workload generated from the Feitelson workload model, a bioinformatics trace from a Top500 supercomputer, Janus, and a bioinformatics trace with data input and output information. The Feitelson workload contains single core jobs as well as parallel jobs up to 64 cores. The Janus bioinformatics workload consists of bioinformatics jobs run on the Top500 Janus supercomputer in March 2011. The bioinformatics trace with data information, based on denoising and read mapping portions of the QIIME workflow, contains both 32-core and 64-core jobs and includes data transfer information for individual jobs.

Feitelson Model Workload	
Number of Jobs	10,000
Minimum runtime	0.0003 seconds
Maximum runtime	12 hours
Mean runtime	37 minutes
Janus Bioinformatics Workload	
Number of Jobs	51233
Minimum runtime	0.0 seconds
Maximum runtime	80.0 hours
Mean runtime	2.5 hours
Bioinformatics Data Workload	
Number of Jobs	256
Minimum runtime (64-core jobs)	5 hours
Maximum runtime (32-core jobs)	12.5 hours

a small local physical cluster with IaaS clouds on a fixed hourly budget. For MCOP, we consider two variants. In the first case, MCOP specifies a 20% preference for cost and an 80% preference for reducing queued time (MCOP-2080) and in the second case an 80% preference for cost is specified and a 20% preference for reducing queued time (MCOP-8020). The evaluation focuses primarily on the ability of the elastic cluster to reduce the runtime of workloads, minimize job queued time, and keep monetary costs to a minimum, over the two extremes, specifically, using a local cluster or only using a public cloud provider, such as Amazon EC2.

Three traces are used for evaluation: 1) a trace generated from the Feitelson workload model [61], 2) a trace of bioinformatics jobs run on a Top500 supercomputer, Janus, at the University of Colorado at Boulder from March 2011, and 3) a trace generated from runtime and data information of denoising and read mapping bioinformatics jobs. The Feitelson model was used in order to represent a common job trace for a multi-user cluster and the Janus bioinformatics trace was chosen as a real job trace from a production supercomputer. The third trace was created because existing scientific workload traces from clusters and supercomputers do not typically include the amount of data transferred on a per-job basis (if at all).

The Feitelson workload contains serial jobs as well as parallel jobs up to 64 cores. The workload submits 10,000 jobs over a 4-hour period with jobs that have a minimum runtime of 0.0003 seconds, a maximum runtime of 12 hours, and a mean of 37 minutes. The Janus bioinformatics workload contains over 51,000 jobs submitted over a month with a minimum runtime of 0 seconds and a maximum of 80 hours. The bioinformatics data workload trace was created from a set of bioinformatics jobs, which performed denoising of 454 datasets and read mapping portions of QIIME's workflow. The trace contains 256 jobs, consisting of 32-core and 64-core jobs, submitted over a 6 day period. The 32-core jobs run for 12.5 hours and transfer 125 GB out of the head node and 55 GB into the head node. The 64-core jobs run for 5 hours and transfer 48 GB of data out of the head node and 3 GB in. The workloads are summarized in Table 7.3.

As metrics, *makespan* is defined to be the time from when the first job in the workload is submitted until the time the last job completes. *Cost* is defined to be the amount of money spent (in USD) over the entire evaluation for the workload, this includes the cost of instances as well as data transfers (where applicable). *Resource CPU time* is defined to be the total number of CPU hours consumed by all instances on all resources. *Response time* is defined to be the amount of time a job is queued before it begins running, that is, the time from when it is submitted until it begins executing. Finally, *average weighted response time* (AWRT) is defined to be the average response time of all jobs in a workload, weighted by the requested number of cores. Response time is defined as the job completion time minus job submit time. Specifically:

$$AWRT = \frac{\sum_{j}^{Q} j.num\_cores * j.response\_time}{\sum_{j}^{Q} j.num\_cores}$$

AWRT demonstrates the impact the policy has on all users where increasing AWRT suggests that jobs are remaining queued for longer times.

7.3.1 Evaluation Environment Configuration

The environment simulates a small static physical cluster, a private community cloud with limited scalability, and a commercial cloud provider (e.g., Amazon EC2) that appears to offer "infinite" scalability but charges by the hour for each instance. The local cluster contains 256 static single-core instances and resources cannot be added or removed from the cluster; launch and termination times are not simulated for this resource since it represents an "always on" cluster. The private cloud contains up to 512 single-core instances. As the private cloud represents a community cloud (e.g., Magellan or FutureGrid), we do not use a monetary cost for the instances. The commercial cloud provider, on the other hand, does have an hourly monetary cost of \$0.26 per instance/hour; partial hour charges are rounded up. Data transfers are \$0.12 per GB out of the cloud. Data transferred into the cloud does not incur any cost. This pricing model is chosen to be the same as Amazon EC2. The commercial cloud does not have limited scalability; it is able to respond to as many requests as needed. Both simulated IaaS clouds randomly generate instance boot and shutdown times based on the times gathered from Amazon EC2 in Table 7.2. Because parallel jobs are often latency sensitive, they are only executed across individual infrastructures, that is, they are not allowed to execute on both the local resource and an IaaS cloud or across both IaaS clouds. The fixed hourly budget is set to be \$10 per hour for this evaluation and any unspent money accumulates. For example, if the budget is not spent over 3 hours then \$30 will be available for future purchases. The policy executes every 5 minutes to examine the job queue and status of the environment; it responds by launching or terminating IaaS instances.

7.3.2 Understanding Environment Impact on a Workload Model

For the Feitelson workload, we execute 10 iterations and simulate over 555 hours. The total makespan is shown in Figure 7.3, that is, the time from when the first job is submitted until the time the last job completes. The total cost is shown in Figure 7.6 and the comparison of makespan vs. cost is shown in Figure 7.8. AWRT is shown in Figure 7.5 and the comparison of AWRT and

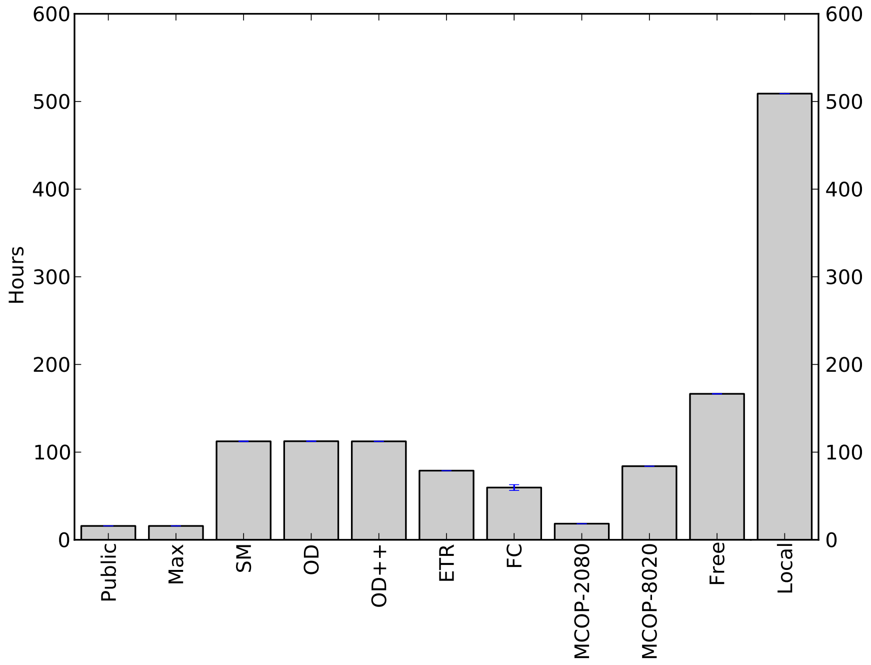

Figure 7.3: Makespan for the Feitelson workload.

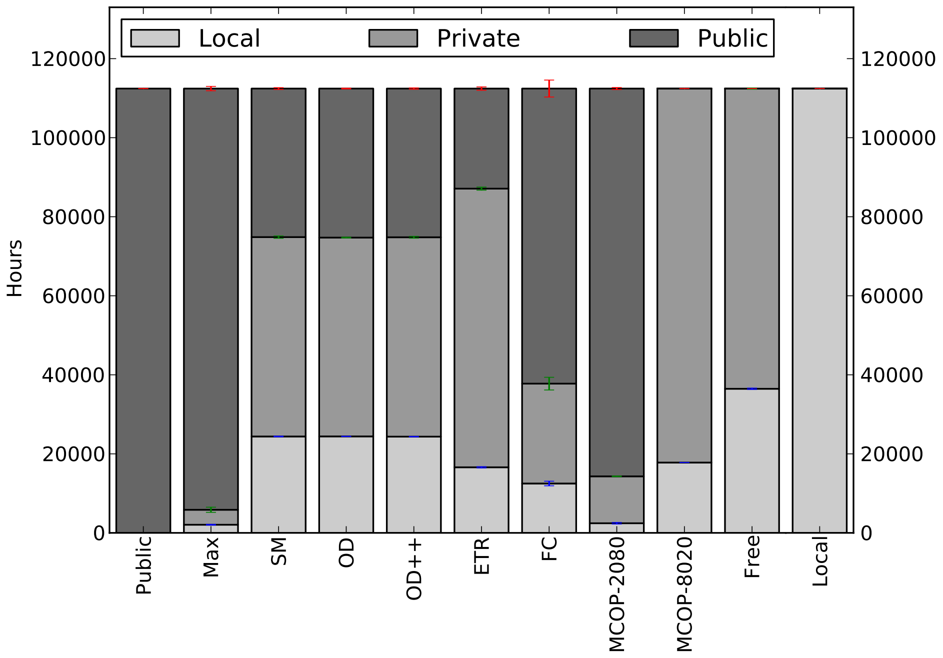

Figure 7.4: CPU time by resource for the Feitelson workload.

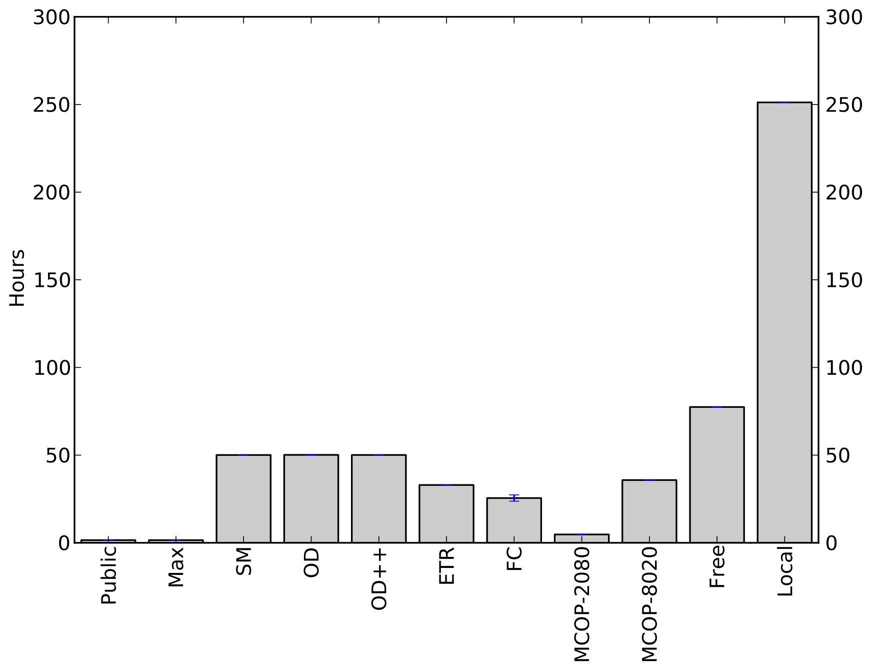

Figure 7.5: Average weighted response time for the Feitelson workload.

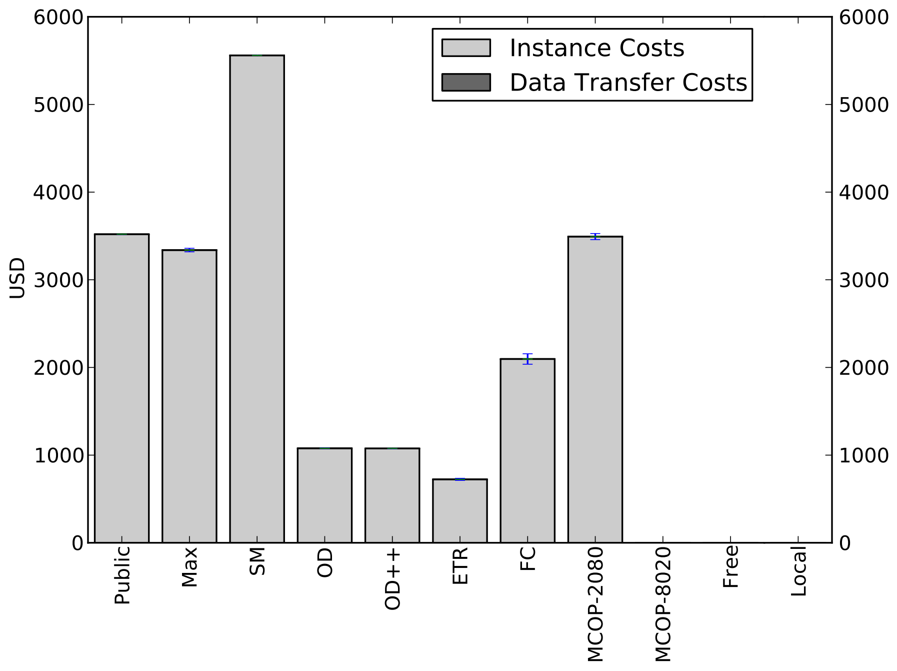

Figure 7.6: Cost (USD) for the Feitelson workload.

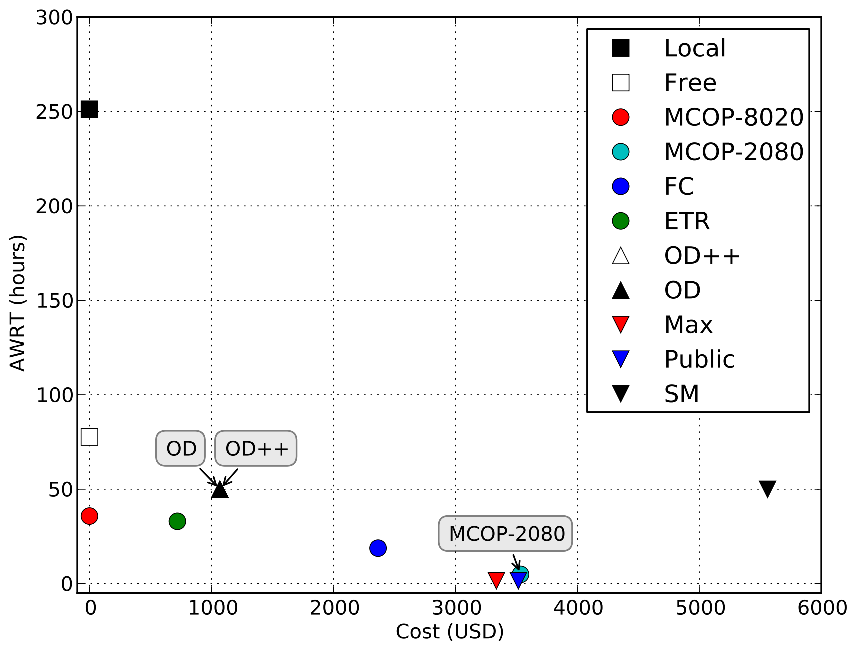

Figure 7.7: Average weighted response time vs. cost (USD) for the Feitelson workload.

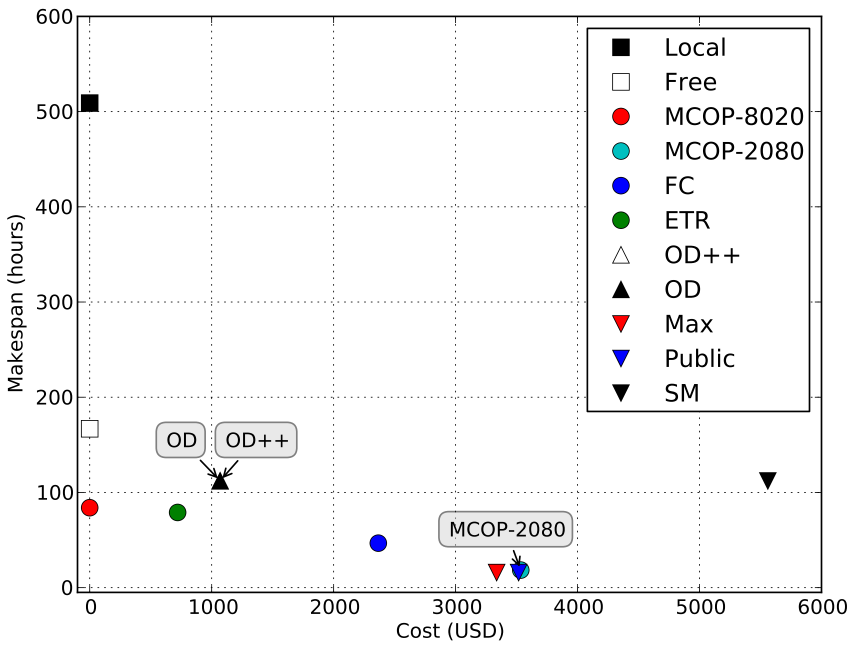

Figure 7.8: Makespan vs. cost (USD) for the Feitelson workload.

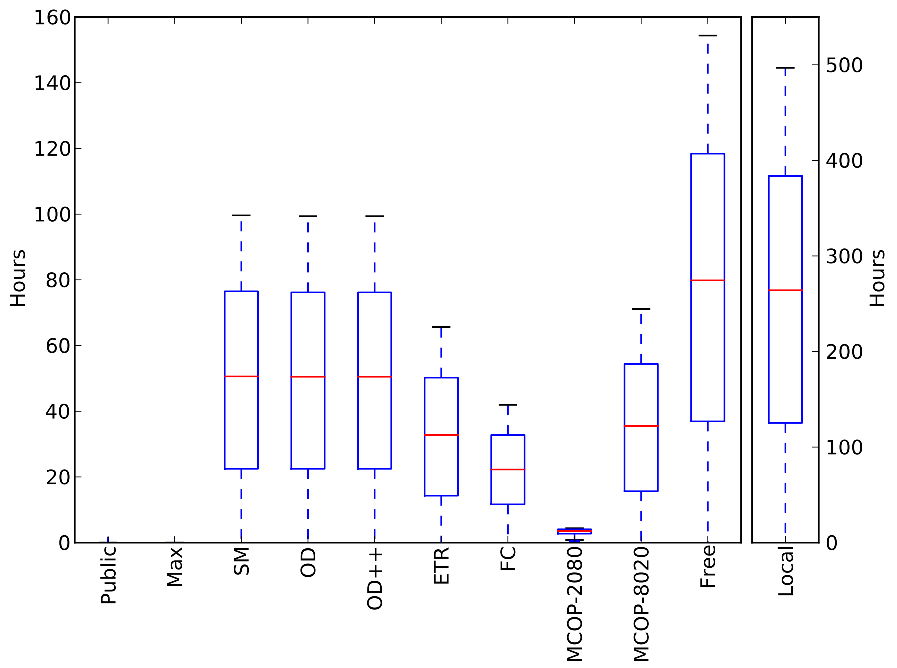

Figure 7.9: Job response time for a single iteration of the Feitelson workload.

cost is shown in 7.7. The total CPU time by resource is shown in Figure 7.4. Outsourcing to IaaS clouds shows a clear benefit over using local resources, reducing the makespan from over 500 hours to less than 16 hours using the Max, Public, and MCOP-2080 policies, which is approximately a 96% reduction in makespan. The Max and Public policies also achieve impressive job response time (Figure 7.9), with a median response times of approximately 15 seconds and a max response times of 5 minutes. However, Max, Public, and MCOP-2080 have a high cost (Figure 7.6): over $3,300. The reason those policies have a high cost is due to their heavy use of the public cloud (Figure 7.4). However, the trade-off is that those policies also appear to have the smallest average weighted response times (Figure 7.5). While using the public cloud incurs a cost, even outsourcing to the private cloud and only using free resources reduces the makespan to approximately 167 hours and achieves a median response time of 79 hours. MCOP-2080 and MCOP-8020 clearly demonstrate the trade-offs of MCOP's cost preference. MCOP-2080, which only has a 20% preference for minimizing costs, achieves low makespan and job response times. However, it experiences comparable cost to the Max and Public policies, over $3,300. MCOP-8020, on the other hand, specifies an 80% preference for minimizing costs and does not spend any money, which also results in relatively high job response time. While MCOP-8020 doesn't incur any cost because it only uses free resources, it uses the resources more effectively than the Free policy (which uses an on-demand approach) as well as OD and OD++, as it achieves lower makespans and job response times for no cost.

ETR appears to strike a reasonable balance between a minimal makespan, low monetary cost, and job response times. ETR achieves low makespan and job response times and a lower cost than all policies except MCOP-8020, Free, and Local as they incur no costs. FC achieves even better response time and makespan than ETR. However, it does so at more than twice the cost. The SM policy is the worst policy. It has a similar makespan to both OD and OD++ but it also has high monetary cost (over $5,500) compared to OD and OD++ (approximately $1070). SM also has a higher cost than Max and Public because instances run for the entire duration of the evaluation even if there are no jobs. We expected OD++ to have a noticeably lower cost or response time than OD because instances are left running until they are due to be charged again, allowing them to process additional jobs if needed. However, for this workload, that was not the case as both OD and OD++ have similar makespans, costs, and job response times.

7.3.3 Understanding Environment Impact on a Bioinformatics Workload

For the Janus bioinformatics workload, we ran 10 iterations and simulate over 818 hours. The workload makespan is shown in Figure 7.10. Overall, the policies, except Local, have similar makespans due to the fact that the workload trace is from jobs submitted over a month, thus, the last batch of jobs submitted at the end of the month largely determine overall makespan. The Local policy doesn't have enough resources available to process the jobs as quickly as the other policies. CPU time by resource is shown in Figure 7.11 where all policies except Public are able to process most of the jobs on the local cluster or private cloud, which minimizes the monetary cost of the deployment (Figure 7.13). Only the Public and SM policies have high monetary costs since Public only uses the public cloud and SM maintains running instances on the public cloud

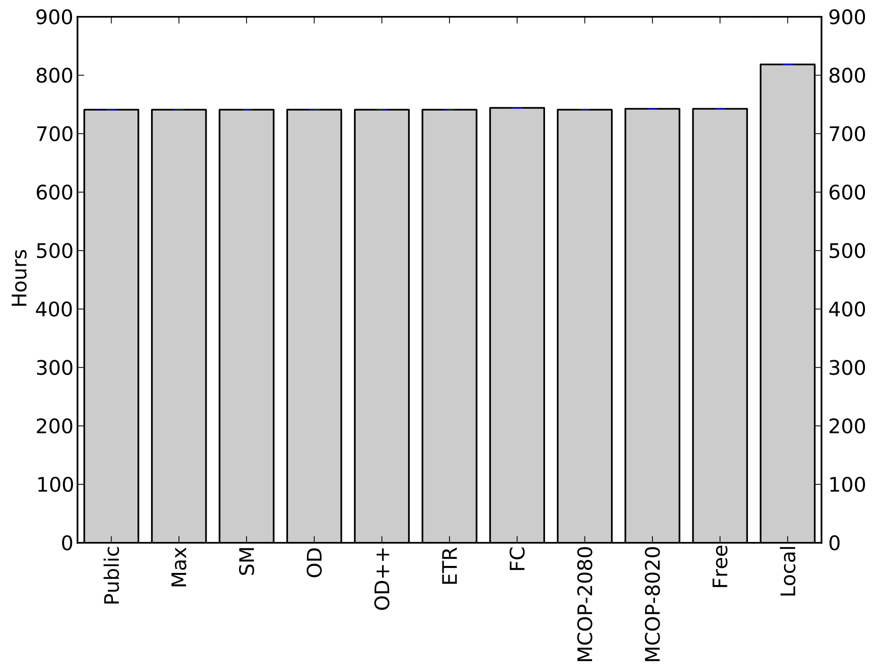

Figure 7.10: Makespan for the Janus bioinformatics workload.

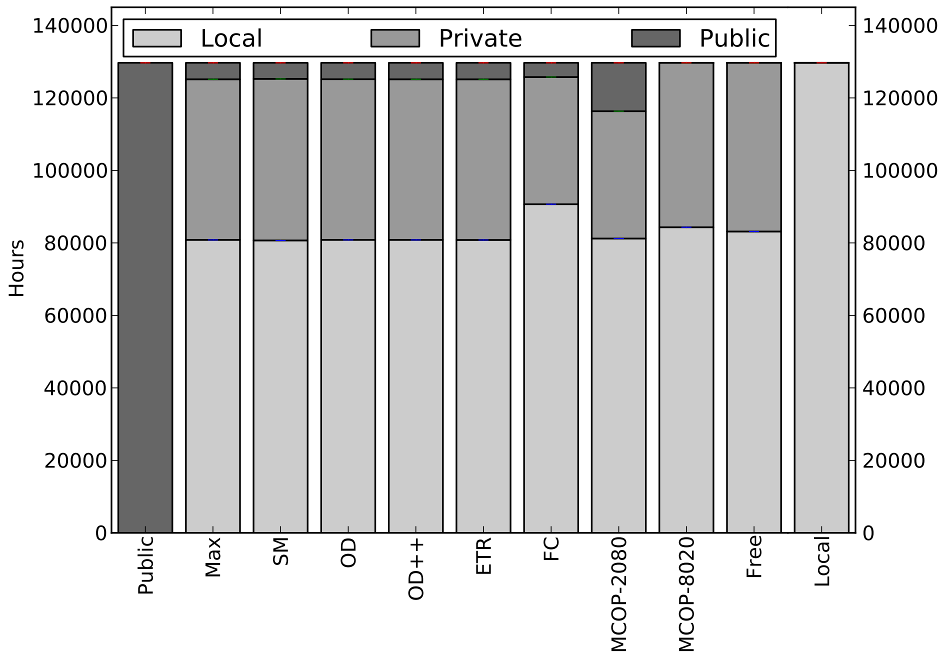

Figure 7.11: CPU time by resource for the Janus bioinformatics workload.

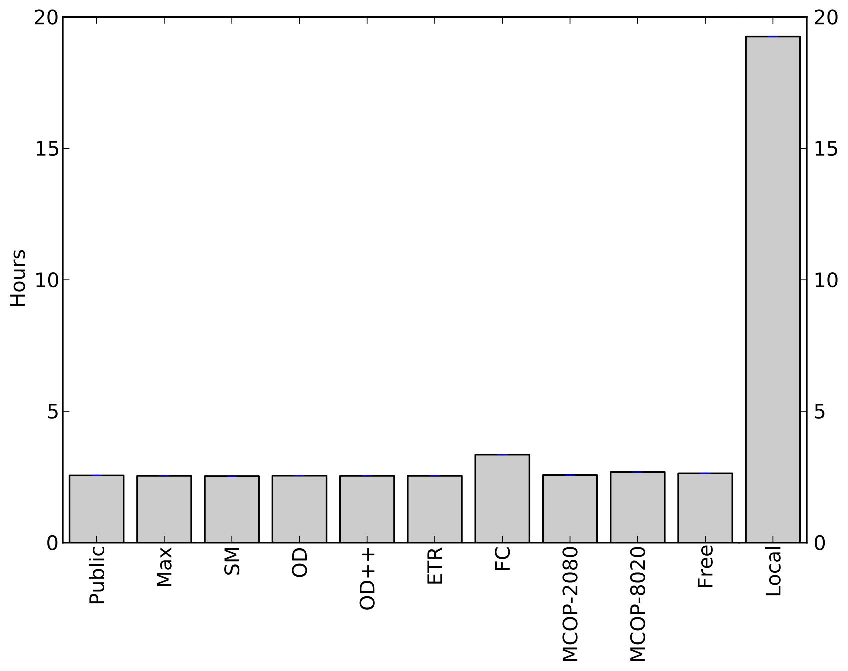

Figure 7.12: Average weighted response time for the Janus bioinformatics workload.

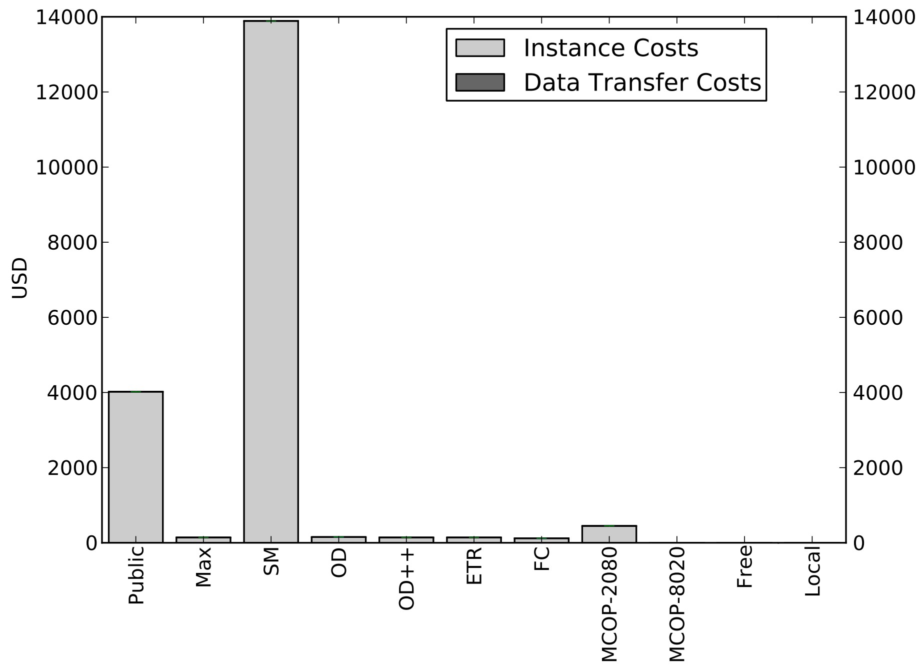

Figure 7.13: Cost (USD) for the Janus bioinformatics workload.

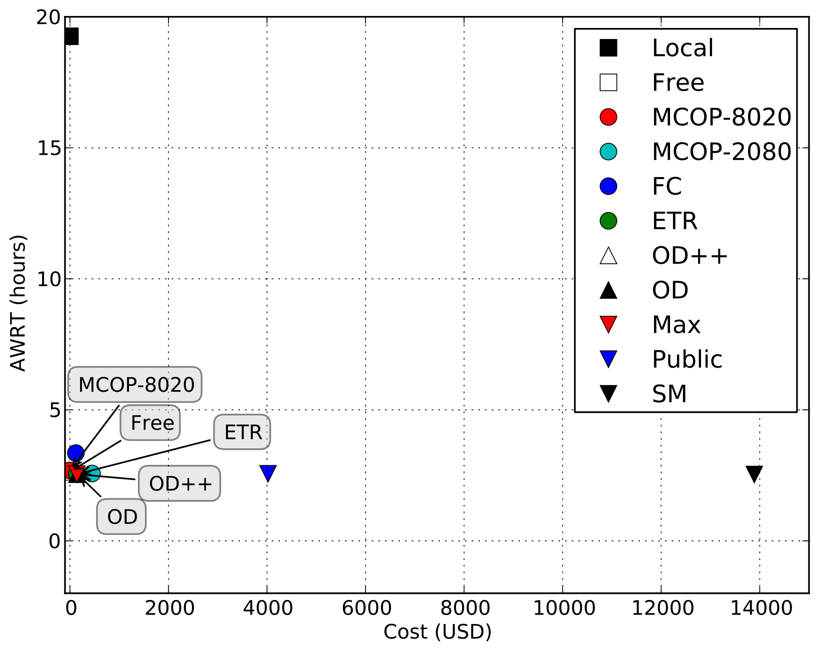

Figure 7.14: Average weighted response time vs. cost (USD) for the Janus bioinformatics workload.

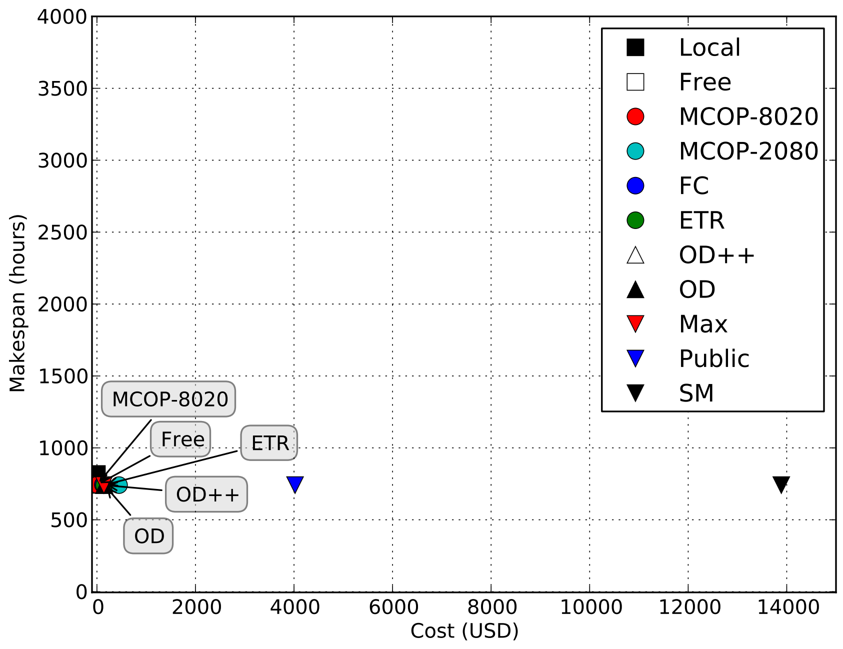

Figure 7.15: Makespan vs. cost (USD) for the Janus bioinformatics workload.

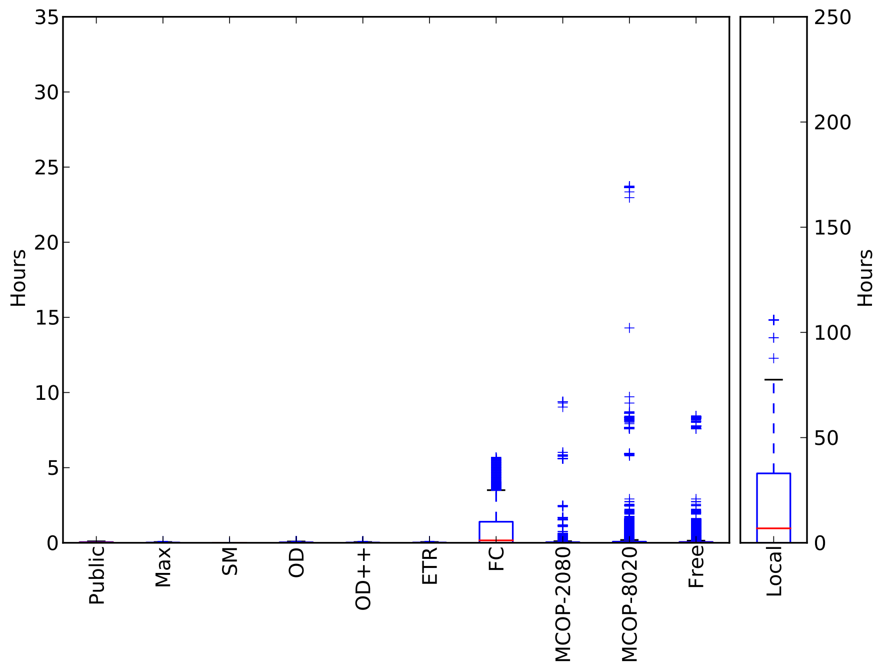

Figure 7.16: Job response time for a single iteration of the Janus bioinformatics workload.

even if they are idle. The other policies are all sufficiently dynamic enough to only use the for-pay public cloud when needed. AWRT is shown in Figure 7.12 and again, the limited resources available to the Local policy results in a much higher AWRT at approximately 19 hours compared to 2.5 to 3 hours for the other policies. FC has a slightly higher AWRT than the other policies because it adapts slower to changes in demand, only provisioning additional resources as AWQT increases. The trade-off between AWRT and cost and makespan and cost is shown in Figures 7.14 and 7.15, respectively. Finally, job response time for a single iteration of the workload is shown in Figure 7.16. Interestingly, FC, MCOP-2080, MCOP-8020, and Free all have noticeably higher job response time for some of the jobs compared to ETR, OD++, OD, SM, Max, and Public despite the fact that their AWRTs are similar. The reason for this is primarily due to the fact that those with shorter job response times are all policies designed to respond immediately to demand across all available resources whereas FC, MCOP-2080, and MCOP-8020 do not necessarily respond to demand immediately and Free is not able to use the public cloud, thus limiting it's ability outsource to a public cloud provider when needed, resulting in higher job response times. Understandably, the Local policy has the highest job response times because it doesn't have sufficient resources to meet the demand.

7.3.4 Understanding the Impact of Data

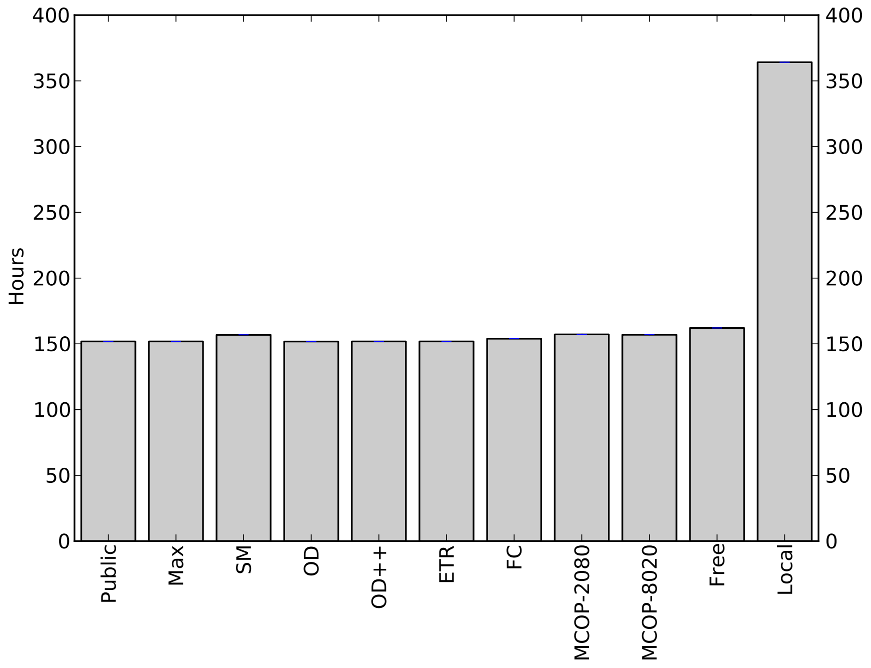

Figure 7.17: Makespan for the bioinformatics workload.

To understand the impact of the bioinformatics data workload, we execute 10 iterations and simulate over 416 hours. Similar to the Feitelson workload, MCOP-8020, Free, and Local do not incur any cost and only use free resources. However, for this workload, MCOP-8020 does not achieve nearly the same improvement in makespan (Figure 7.17) or job response time (Figure 7.23), as it is similar to the Free policy. This workload again demonstrates the advantage of outsourcing to IaaS clouds, as the makespan is reduced from over 350 hours for the Local policy to approximately 150 hours for the other policies and the average weighted response time is reduced from over 140 hours to less than 20 hours (Figure 7.19). The trade-offs of makespan vs. cost is shown in Figure 7.22 and

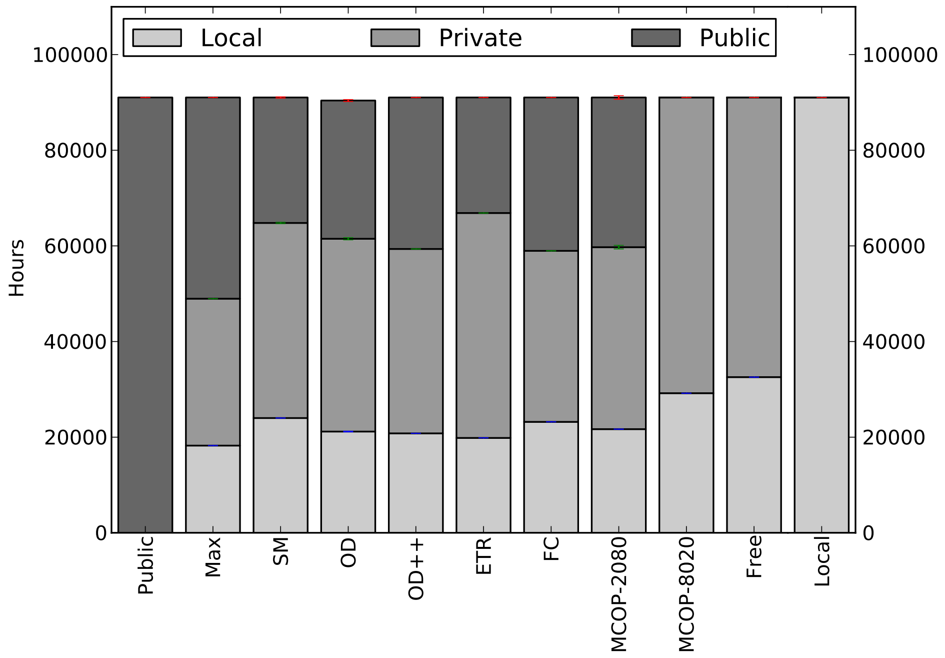

Figure 7.18: CPU time by resource for the bioinformatics workload.

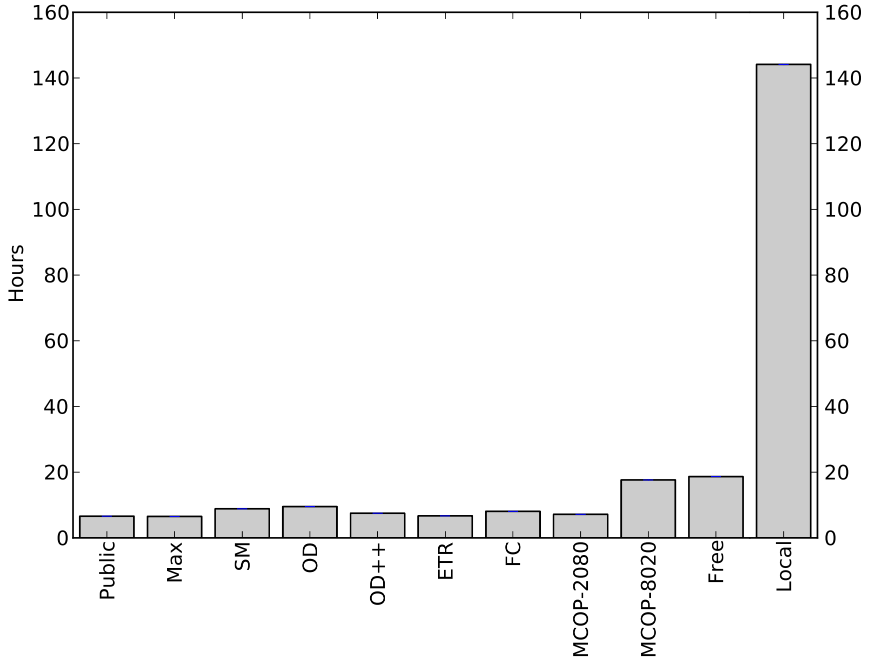

Figure 7.19: Average weighted response time for the bioinformatics workload.

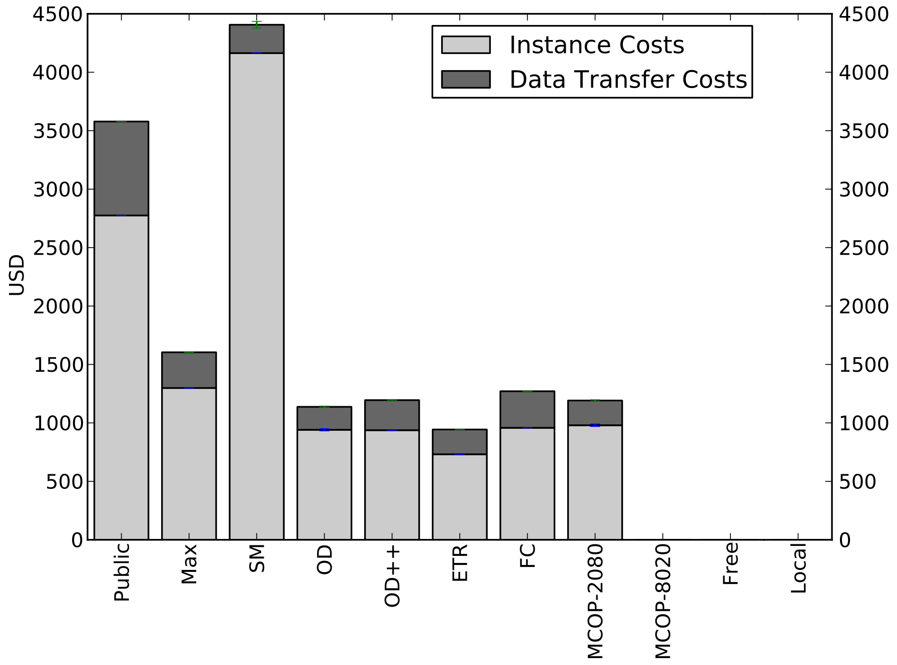

Figure 7.20: Cost (USD) for the bioinformatics workload.

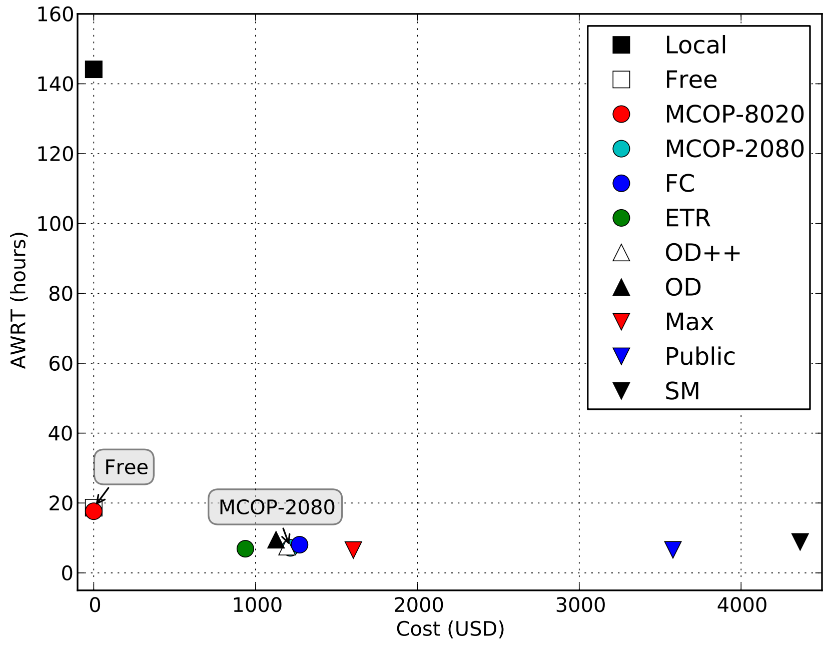

Figure 7.21: Average weighted response time vs. cost for the bioinformatics workload.

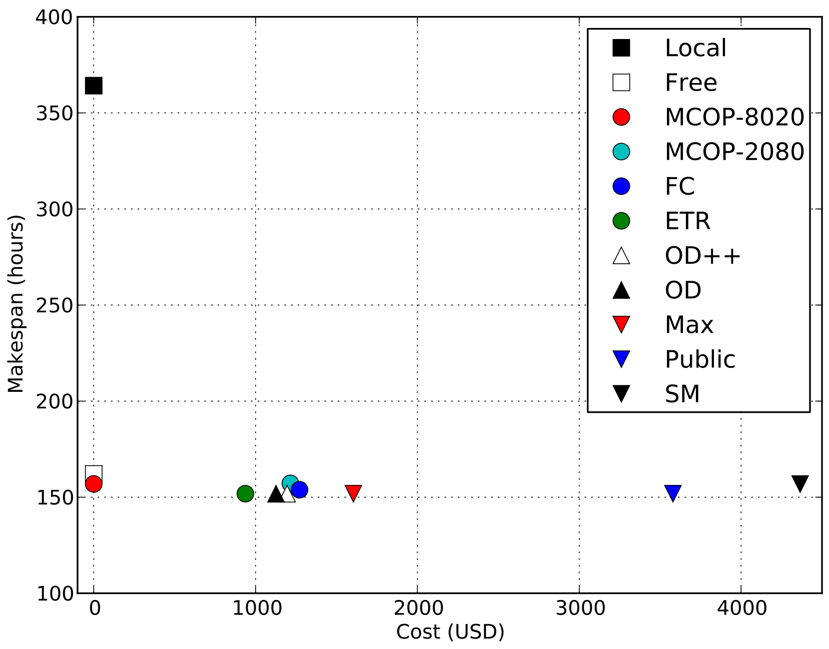

Figure 7.22: Makespan vs. cost for the bioinformatics workload.

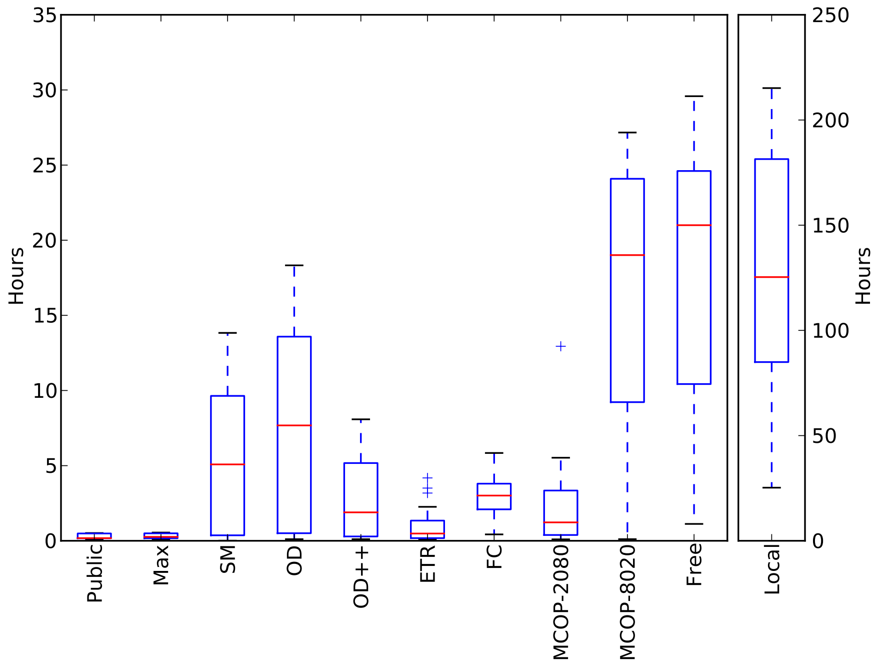

Figure 7.23: Job response time for a single iteration of the bioinformatics workload.

AWRT vs. cost is shown in Figure 7.21. ETR, the only policy that considers data transfers in its decision process, and MCOP-8020 appear to strike the best balance between a minimal makespan (Figure 7.17), low cost (Figure 7.20), and reasonable job response times (Figure 7.23). For example, ETR reduces the makespan to 150 hours from 350 hours for the local policy, representing a 57% reduction. And again, while the Max and Public policies achieve the lowest job response times with an approximate median of 0.26 hours and a maximum of 0.54 hours, they do so at high cost, over $3,500 for Public and over $1,200 for Max compared to ETR at $950. The Public policy's use of only public cloud instances (Figure 7.18) also translates to higher data transfer costs (Figure 7.20). As expected, MCOP-8020 and MCOP-2080 again demonstrate the trade-offs of choosing a preference to minimize cost. Interestingly, for this workload, OD++ achieves lower job response times than OD, however, at a slightly higher cost.

If multiple cloud providers are available and data information is known, ETR achieves the best combination of relatively low job response times, low makespans, and low costs. The Max and Public policies are clearly the best policies to minimize job response time and makespan, however, they do so at a cost. FC achieves relatively low job response time, though it is higher than ETR and the policy incurs more cost than ETR. OD and OD++ have lower costs than Max and reasonable job response times but they both have higher cost and job response time than ETR. MCOP offers administrator's the ability to specify their preference for reducing cost or job response time. SM appears to offer no value over the other more dynamic policies and has a high cost since instances run for the duration of the evaluation, whether or not they are processing jobs.

7.4 Conclusion

This chapter presented a set of resource provisioning policies for elastic environments that allow such environments to outsource demand to external cloud providers by provisioning and relinquishing cloud resources. Resource provisioning policies are a central component in the flexible cloud architecture since they determine the size of the elastic environment as well as the clouds that instances are deployed in, which directly impact job queued time and the cost of the deployment.

The question of which policy is the "best" depends on numerous factors, including the money available to the administrator, user requirements, and workload characteristics. However, it is clear that elastic environments provide many advantages over using only a standalone local cluster, allowing workload runtimes and job queued times to be reduced significantly by outsourcing demand to external clouds. Even in cases where cost is a concern, it is possible to leverage free private clouds to reduce the workload makespan and job response times. Preemptively provisioning the maximum number of instances, based on budget and available resources, doesn't offer any significant advantages over more flexible resource provisioning policies. Furthermore, policies that consider more factors than just the number of queued jobs, including the estimated time remaining and the time to transfer data or attempt to balance conflicting objectives, offer better results than basic on-demand policies, often reducing both job queued time and the cost of the deployment.

Chapter 8

Bioinformatics Use Case

To demonstrate the complete end-to-end capabilities of the elastic environment, a bioinformatics toolkit, Quantitative Insights Into Microbial Ecology (QIIME) [47], is deployed and used to process bioinformatics datasets. QIIME is a software package that supports the analysis and comparison of complex communities of microorganisms, such as bacteria based on DNA sequence data. Datasets of 10s to 100s of GB in size are generated by high-throughput DNA sequencing technologies, such as pyrosequencing [91]. These datasets contain genomic subsequences from tens of thousands of microbial organisms (e.g., bacteria) in a given sample. The main goal of this use case is to demonstrate the ability of the environment to integrate seamlessly with the bioinformatics workflow. Scientific researchers must be able to use elastic environments in the same manner that they use their existing standalone clusters, for example, logging into a cluster and running command line applications to execute the QIIME pipeline.

8.1 Bioinformatics Workflow

QIIME integrates many tools that have been previously developed into a single workflow. Additionally, it implements many techniques natively to support the analysis of microbial communities. Studies of microbial communities are becoming more frequent and researchers are continually improving their tools and techniques for analysis, allowing them to explore increasingly complex questions. A typical QIIME analysis will apply several different applications in the workflow, including OTU picking with tools such as uclust [59] or BLAST [36], multiple sequence alignment with

PyNAST [46], and phylogenetic tree construction with RAxML [116] or FastTree [106]. These steps support "downstream" statistical analyses of microbial diversity as well as generation of visualizations and publication-quality graphics. QIIME integrates over 30 software packages and libraries, developed by research groups and organizations across the world. This is reflected clearly in the large number of technologies and programming languages used. For example, QIIME's underlying software stack consists of a Haskell application, Python scripts, and x86 binaries, among others. Researchers typically execute the QIIME workflow by running a series of command line scripts. Numerous stages of the workflow include parallel support, allowing researchers to seamlessly use multiple cores on their systems or submit jobs to a cluster queue. QIIME currently supports both Torque or Oracle Grid Engine resource managers.

In this use case, we consider two parallel stages of QIIME's workflow. The first stage, denoising 454 datasets that are generated from 454 pyrosequencing technologies, is a computationally expensive procedure for correcting errors in DNA sequencing reads and involves both parallel and serial steps. The second parallel stage is the mapping of DNA sequence reads to a database of known sequences. It is used to assign sequenced genome fragments from the DNA sequencing instrument to known functional genes. Denoising mixes serial and parallel steps, and therefore submits individual single-core jobs in a chain, while read mapping submits all of its jobs at once.

8.2 Policies for Bioinformatics Workflows

Bioinformatics researchers can run the QIIME workflow as-is using basic provisioning policies, such as an on-demand policy that launches cloud instances for queued jobs and achieve reasonable results. However, creating custom policies that match the submission patterns of different parts of the workflow may significantly improve resource provisioning, allowing jobs to begin running sooner and reducing time to result depending on workflow job submission. We present a new policy, N-preemptive, customized for the denoising stage of the workflow that attempts to preemptively provision the needed number of instances and reduce the amount of time it takes for denoising to begin executing. For comparison purposes we also include the on-demand policy we proposed previously in Chapter 7. The policies are defined as follows:

- *On-demand*: launches the appropriate number of instances for all cores requested by queued jobs. Idle instances are terminated immediately. For example, if a user submits a 32-core job, the policy would request 32 total cores from the cloud provider. Once the jobs complete and the instances become idle, the instances are terminated.
- *N-preemptive*: launches N total cloud cores for every core requested by queued jobs. Similar to on-demand, idle instances are terminated immediately. For example, if the administrator sets N to be 8, then the policy would launch a total of 8 cores for each core requested by queued jobs.

The N-preemptive policy is intended for workflows similar to the denoising workflow, that is, those that submit jobs in a dependency chain. The N-preemptive policy launches multiple instances at once, allowing the subsequent jobs in the chain to possibly begin running earlier. If N is set to be 1, the policy is equivalent to the on-demand policy.

8.3 Deployment Environment

For the bioinformatics deployment, the elastic environment operates entirely in the cloud and is configured to extend a batch-queue cluster with IaaS worker nodes. The environment provides an interactive login node for users, which runs the cluster server software and hosts user applications. App-deploy is used to install the complete QIIME toolkit on all of the nodes throughout the environment. The lightweight sensor and policy are hosted on the login node and use the Phantom auto-scaling service, running on a separate node, to deploy and manage worker nodes. The recontextualization broker is also deployed on a separate node and facilitates the exchange of host information between all nodes in the environment. As worker nodes launch, they join the context and configure themselves to join the cluster. This process includes adding SSH host keys for all other nodes in the environment to their ssh_known_hosts file, connecting to the cluster server, and installing Gluster in order to seamlessly access the central Gluster file system being exported from the cluster head node. Researchers can then login to the elastic environment and execute the QIIME workflow in the exact same manner as they use it on traditional clusters. Parallel jobs are submitted to the batch queue and execute on IaaS cluster workers, which are fully integrated into the cluster and have access to read and write from the shared cluster file system.

8.4 Evaluation

Experimental setup is as follows, the elastic environment is deployed across multiple NSF FutureGrid clouds and is used to process bioinformatics datasets that contain millions of sequences. The environment consists of the Hotel cloud at the University of Chicago and Sierra at the San Diego Supercomputing Center. The head node is a dual-core 2.93 GHz Xeon with two GB of RAM running on Hotel; workers are dual-core 2.93 GHz Xeons with six GB of RAM running on either Hotel or Sierra. The head node manages the Torque 3.0.6 server components and exports the shared Gluster 3.3.1 file system. The denoising workload contains over 1.7 million sequences and the input file is 194 MB. The read mapping workload contains slightly less than 10 million sequences that are represented in an approximately 2 GB (input) file.

As a metric, *workload time* is defined to be the time from when the first job is submitted until the last job completes. *Time until all jobs running* is defined as the time until all jobs submitted together in one stage of the workflow are running. *Cumulative instance deployment time* is defined to be the cumulative amount of time that it takes to deploy all instances. And finally, *cumulative idle time until all jobs are running* is defined to be the cumulative time instances are idle until all jobs are running. For example, for the denoising workload, if the user specifies 64 jobs, cumulative idle time until all jobs are running consists of the total amount of time any individual jobs are running on instances until all 64 jobs are running, allowing the parallel portion of denoising to proceed. To demonstrate the end-to-end capabilities of the environment, a trace for QIIME's read mapping analysis is also included. The trace shows the number of jobs queued, running, and complete as well as instances launched and running, and the amount of data transferred in and out of the head node.

8.4.1 Understanding Policy Impact on Scientific Workflows

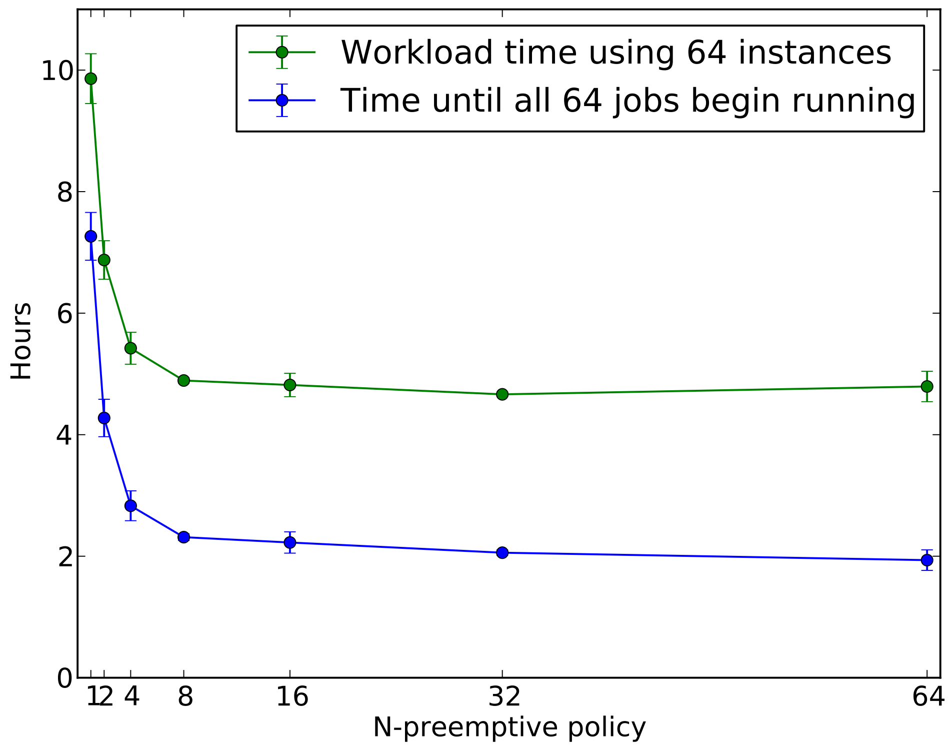

Figure 8.1: Runtime for the denoising workload using 64 instances to run 64 denoising tasks. Values of 1 to 64 are used for the N-preemptive provisioning policy and 3 iterations are run for each experiment.

For the denoising workload and the N-preemptive provisioning policy, all runs specify 64 tasks and only use workers on Hotel. QIIME submits 64 jobs in a dependency chain, beginning with a single job, and all jobs must begin running for denoising to start. Values of N from 1 to 64 are considered, meaning that when the first denoising job is submitted N instances will be launched immediately. Setting N to 1 is equivalent to the on-demand policy where only a single instance is launched for a single job. For this portion of the evaluation, we are particularly interested in the ability of the N-preemptive policy to minimize the time until all 64 jobs are running, minimize overall runtime, and minimize wasted time over simply using a basic on-demand policy (i.e., N=1). Due to the significant amount of time required for a single run, only 3 experiments are run for each value of N and the denoising workload.

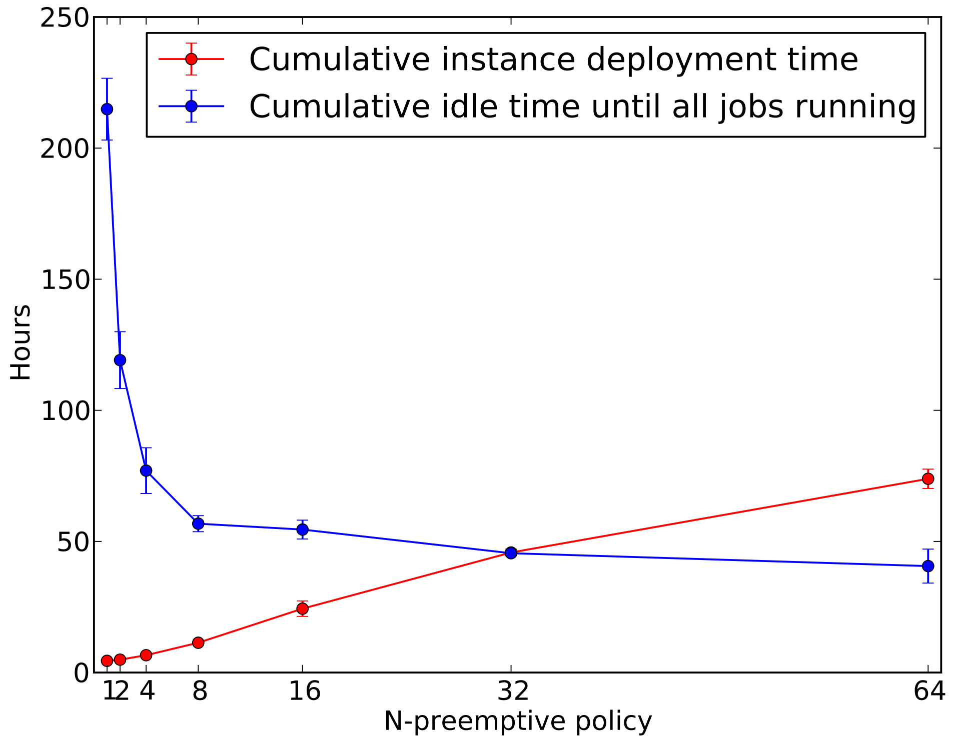

Figure 8.2: Job idle time and instance wasted time for the denoising workload using 64 instances to run 64 denoising tasks. Values of 1 to 64 are used for the N-preemptive provisioning policy and 3 iterations are run for each experiment.

Figure 8.1 shows the overall workload time and the time until all 64 jobs are running. Preemptively provisioning more than a single node when the first job is submitted significantly reduces the overall runtime for the denoising workflow. The on-demand policy ($N = 1$) takes over 7 hours until all 64 parallel denoising jobs are running whereas N-preemptive for $N \geq 8$ is approximately 2 hours. While on-demand provisioning may be well-suited for many scientific workloads, the N-preemptive policy clearly demonstrates that a provisioning policy customized for specific submission patterns can greatly reduce overall runtime and wasted time. Interestingly, provisioning all 64 instances at once does not offer substantial improvement over provisioning 8 instances or more at a time. The is primarily due to the specific cloud architecture and implementation of Hotel. That is, the more instances that are deployed at the same time, the longer it takes to deploy them because network contention is encountered as the images are transferred from the image repository to cloud

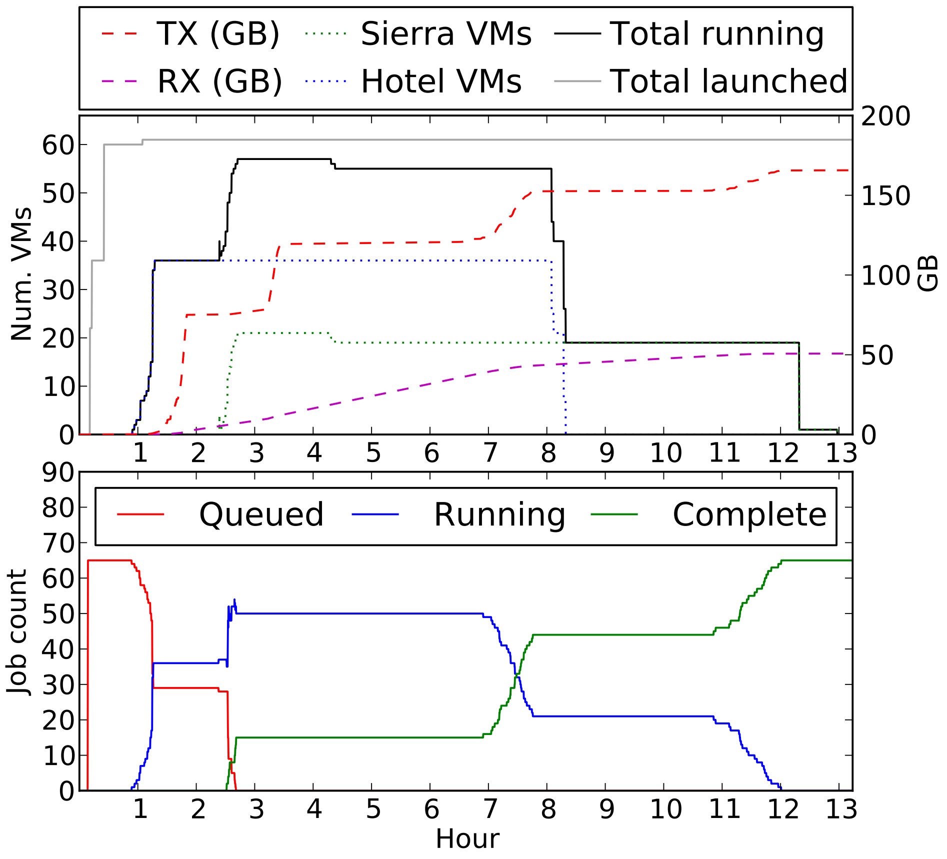

Figure 8.3: Trace of the read mapping workload using the on-demand policy (N=1), showing the number of queued, running, and complete jobs as well as VM and data information. Data information includes data transferred out of the head node to workers (TX) and data received on the head node from workers (RX).

hypervisor nodes, as shown in Figure 8.2. While increasing N reduces wasted idle time, it increases instance deployment time (again due to network contention). However, it is worth noting that increasing instance deployment time for an increasing number of instances may not be true of all cloud providers. Therefore, it is also important to consider the characteristics of individual clouds when creating or adapting a policy to match a specific workflow pattern. Individual traces for increasing values of N from 1 to 64 showing job and VM information are included in the appendix.

8.4.2 Understanding End-to-End Capabilities in Multi-Cloud Environments

To demonstrate the end-to-end capabilities of the environment, the read mapping workload, which has significant data requirements, is processed using multiple IaaS clouds. Since this portion of QIIME's workflow submits all parallel jobs at the same time only the on-demand policy is considered. Figure 8.3 shows a trace of the workload, using both Hotel and Sierra, that includes jobs submitted, running, complete, and VMs launched and running, as well as data transferred into (RX) and out of (TX) the head node. The entire workload transfers over 200 GB of data between the head node and workers, with 150 GB transferred out of the head node and approximately 50 GB in. Unfortunately, the multi-cloud environment is not able to scale to 64 instances. This is possibly due to heavy demand on the community clouds from other users. However, the environment is able to launch 36 instances on Hotel before launching an additional 21 instances on Sierra. Once the head node is configured with the Torque sensor and policy, the entire environment operates automatically and without any user intervention. Cloud workers are launched and terminated automatically as jobs are submitted to the queue. Once workers are running, jobs are automatically dispatched to the workers and data transfers seamlessly. In this elastic environment, a user can simply execute QIIME commands on the head node as they would on any other cluster, and the elastic environment adapts automatically as it should.

8.5 Conclusion

The elastic environment seamlessly outsources scientific workloads to multiple IaaS clouds and allows researchers to interact with the environment in the same manner that they use traditional clusters. The complex bioinformatics software stack, QIIME, which consists of over 30 dependencies, is deployed with a single command using app-deploy. The experimental evaluation shows that provisioning policies customized for specific workflows can be highly beneficial, in particular, the N-preemptive provisioning policy reduces the runtime of one stage of the QIIME workflow by approximately a factor of two compared to our on-demand approach. The experimental evaluation also shows the ability of the elastic environment to support large-scale deployments that seamlessly transfer hundreds of gigabytes of data between multiple cloud providers. However, the performance of these elastic environments depend largely on the performance of the underlying hardware and technologies. Consideration should be given to the limitations of the performance of the underlying hardware as well as the requirements of the scientific workflow. Scientific applications with strict latency requirements or sensitivities to OS jitter may not be well-suited for widely-distributed elastic environments given the current state of underlying technologies. However, many other scientific workloads and applications, such as loosely coupled parallel jobs or embarrassingly parallel workloads similar to these bioinformatics workloads, may be easily outsourced to IaaS clouds. Finally, the diversity of scientific applications and workflows that can effectively leverage these elastic environments will likely continue to increase as technologies evolve and mature, offering better performance for lower costs.

Chapter 9

Conclusion and Future Work

This dissertation addresses the need for flexible cloud environments that adapt to variable demand. Preemptible and preset leases, which deploy preemptible VMs, increase the utilization of under-utilized IaaS clouds and an elastic environment is developed to adapt to variable demand using IaaS clouds, outsourcing excess demand to external clouds. The open source Nimbus IaaS toolkit is extended to support preemptible and preset leases, deploying preemptible VMs on idle VMM nodes, which allow tasks to use cycles that would have otherwise been idle. The elastic environment extends site services, such as HPC resource managers, to use cloud auto-scaling services and adjust to variable demand. The environment adjusts based on two major factors: 1) the current demand, such as the number of queued jobs, and 2) the requirements specified by the resource administrator, such as minimizing job queued time or the cost of the deployment. By adapting to variable demand, the flexible cloud architecture allows RPs to purchase a smaller resource that meets the needs of their user community a majority of the time while budgeting for future outsourcing costs. RPs should give consideration to workload requirements when outsourcing demand. Certain workloads, such as HTC or volunteer computing jobs without deadlines, are more amenable to preemptible VMs because they can be preempted and rescheduled while parallel jobs with deadlines require high-cost, on-demand instances to ensure parallel or deadline-driven applications can run to completion before instances are terminated [90].

9.1 Key Contributions

To address variable demand effectively, a flexible cloud architecture and implementation is developed and evaluated. The flexible cloud architecture consists of preemptible and preset leases to help increase utilization without sacrificing on-demand leases and an elastic environment to outsource excess demand when needed. To support preemptible and preset leases, an open source IaaS toolkit, Nimbus, was extended. These preemptible and preset leases provide a new type of resource lease, typically offered at less cost than on-demand leases, for scientific users seeking to use IaaS clouds. A large-scale elastic environment was also created and a prototype implementation developed. The elastic environment scales up the deployment as demand increases and terminates instances as demand decreases. It can either be deployed entirely in the cloud or be used to extend existing site resources, such as a local Torque cluster. To balance user and administrator requirements, provisioning policies were developed and evaluated. The provisioning policies elect to launch or terminate cloud instances based on demand as well as user and administrator requirements, such as minimizing costs. Finally, the end-to-end capabilities of the elastic environment prototype is demonstrated by processing a bioinformatics workload with data requirements, outsourcing jobs to multiple IaaS clouds. The key intellectual contributions of this research include:

- Design of a flexible cloud architecture that adapts efficiently to variable demand for both the user environment and the underlying infrastructure. The architecture supports common scientific workload patterns and characteristics, including HTC or volunteer computing workloads and parallel computing workloads.
- Formulation of resource provisioning policies for elastic environments that balance user- and administrator-defined requirements, minimizing costs. The policies seek to adapt effectively to demand for both specific workloads and generic workload models in order to meet the desired requirements.
- Extend an existing scheduling algorithm for Grid environments to support IaaS environ-

ments. The policy uses a genetic algorithm to balance conflicting objectives.

- Analysis of the elastic cloud model, including both compute and data aspects. The model consists of a standalone cluster, a private cloud with limited scalability, and an "infinitely" scalable for-pay public cloud provider and examines the trade-offs of deployments that use these different infrastructures.

- Evaluation and analysis of the elastic environment using bioinformatics workloads with significant data requirements, providing additional insight into the requirements of large-scale deployments for scientific workflows.

The key engineering contributions of this research include:

- Development of preemptible and preset leases for the open source Nimbus IaaS toolkit, allowing users to leverage a new type of resource lease for workloads without immediate deadlines, such as HTC or volunteer computing workloads.

- Development of a scalable multi-cloud elastic environment that adapts to variable demand, outsourcing work with infrastructure clouds when needed.

- Development of an elastic cloud simulator to evaluate the elastic cloud model, which consists of local resources, private clouds, and public clouds for different scientific workloads.

9.2 Future Work

This thesis presented a working elastic cloud framework that transparently (to the user) extends a common open source cluster software stack to produce an elastic cluster computing environment. The resulting environment satisfies the design objectives and is thoroughly tested by running a series of simulations, common use case studies, and demanding QIIME bioinformatics workflows. There remains, however, a wide range of research and development that would improve and extend this environment. In particular, the tools and techniques presented in this dissertation can be customized and tuned for individual workloads and applications in order to improve performance and user experiences. For example, the on-demand policy may be suitable for some workflow patterns, such as those that submit jobs in batches periodically. However, it is also possible for the on-demand policy to waste significant time or resources by not responding appropriately to different workflow patterns. This was illustrated with the bioinformatics denoising workload, consisting of serial and parallel jobs, where jobs were submitted in a dependency chain. Using the on-demand policy to provision a single VM for a single queued job required over 10 hours to process a dataset while provisioning 8 or more VMs at the same time reduced the workload time by approximately a factor of two. Workflow patterns that submit jobs continuously instead of periodic batches may benefit from the gradual adjustments made by the average queued time policy instead of a basic on-demand policy, which may result in wasted time or resources by constantly launching and terminating instances.

Currently, the policy for the elastic environment must be chosen manually, however, developing a framework to automatically select an appropriate policy based on environment and workload information will allow the elastic environment to operate effectively without user intervention. Such a framework requires sensors to gather information about the environment, including its utilization and job response time, as well as its own set of policies to identify needed adjustments, such as switching to a different elastic environment policy or leveraging a different cloud providers as their offerings change. Furthermore, users with highly custom workflow patterns should be able to define and provide their own policies that are best suited for their workflow and application patterns, minimizing wasted time, resources, or money.

The elastic environment implementation also relies on existing distributed file systems, such as Gluster or XtreemFS, for data movement. Developing more advanced data movement solutions that integrate with user workflows will also be beneficial. In particular, these solutions should account for both network performance as well as the cost of transferring data into and out of clouds. These solutions should handle failures gracefully and accommodate deployments consisting of a variable number of instances. The elastic environment can also leverage existing batch-queue techniques, including the use of multiple queues for batch-queue resource managers, in order to improve performance for certain applications and workflows. For example, jobs could be routed to multiple queues on a batch-queue system where jobs in different queues execute on different resources. A single queue could be made available for users, but jobs could then be automatically routed to different hidden queues depending on their characteristics. This would allow tightly-coupled jobs with low latency requirements to be routed to a queue that only executes jobs on a local resource with a fast interconnect while parallel jobs capable of tolerating high latency or embarrassingly parallel jobs could be routed to a queue monitored by an elastic environment sensor and policy, allowing them to execute across multi-cloud infrastructure resource deployments.

A framework to perform "live" benchmarking of cloud resources is needed in order to better understand the performance of different clouds and their offerings, which continuously evolve. Such a benchmark suite will allow users to compute an up-to-date price-to-performance ratio when evaluating cloud providers and avoid some of the pitfalls associated with cloud provider over-provisioning, specifically the case where a large number of VMs share a single physical node. Another area is the development of tools and methods for image definition, creation, and resource contextualization. Improved tools and methods are required to assist users with the complicated process of configuring and deploying images for a wide variety of IaaS clouds. New workflow tools that orchestrate the execution of workflow applications across elastic IaaS environments will allow users to easily map different workflow components to appropriate resources, improving utilization and performance of their deployments depending on cloud characteristics and workload requirements.

While we demonstrate the usefulness of the tools and techniques presented in this dissertation with a bioinformatics use case, many of these tools and techniques can be applied directly to other domains without modification. Scientific applications and workflows that leverage batch-queue clusters can use this work directly, allowing users to outsource their workloads to the cloud transparently. Parallel applications that tolerate high latency communication and embarrassingly parallel applications can run across multi-cloud deployments while tightly-coupled parallel applications can be restricted to resources with low latency interconnects using multiple queues and queue routing. In particular, specific applications including those used by the Asteroseismic Modeling

Portal running the Aarhus Stellar Evolution Code and a parallel genetic algorithm [119] as well as earthquake simulation codes and geological applications [79], among others. With support for additional resource managers, such as Hadoop, large-scale data analysis applications and workflows will be able to use the environment. Furthermore, new sensors and policies can be developed to integrate web server software or database services with the elastic environment. While the policies presented in this dissertation respond to both bioinformatics batch-queue workloads and generic batch-queue workload models, they can also be tailored for other workload paradigms, such as monitoring and responding to variable CPU or memory load on web server or database processes. For example, a website administrator may choose to use a variant of the multi-cloud optimization policy to balance request response time for a Web site, instead of job queued time, with the cost of the deployment.

Bibliography

[1] Amazon Auto Scaling, Amazon, Inc. http://aws.amazon.com/autoscaling/.

[2] Amazon Cloud Watch, Amazon, Inc. http://aws.amazon.com/cloudwatch/.

[3] Amazon Elastic Compute Cloud (EC2), Amazon Inc. http://aws.amazon.com/ec2/.

[4] AMD-V. http://www.amd.com/virtualization.

[5] boto: A Python Interface to Amazon Web Services.

[6] Chef. Opscode. http://wiki.opscode.com/display/chef/Home.

[7] Cloud Foundry. http://www.cloudfoundry.com.

[8] Condor Log Analyzer. http://condorlog.cse.nd.edu.

[9] Cycle Computing, 50,000-core Cluster. http://www.cyclecomputing.com/news/news/250-cycle-spins-up-50000-core-cluster-in-amazon-cloud.

[10] Debian Apt. http://wiki.debian.org/Apt.

[11] Engine Yard. https://www.engineyard.com.

[12] Future Grid. http://futuregrid.org/.

[13] Gluster. http://www.gluster.org.

[14] Google compute engine. https://cloud.google.com/products/compute-engine.

[15] Heroku. http://www.heroku.com.

[16] Intel VT. http://ark.intel.com/Products/VirtualizationTechnology.

[17] MIT StarCluster. http://star.mit.edu/cluster/.

[18] Nimbus. http://www.nimbusproject.org.

[19] November 2010 TOP500. http://www.top500.org/list/2010/11/.

[20] OOI 10-Gigabit Peering with Amazon Web Services. http://calit2.net/newsroom/release.php?id=1709.

[21] OOI Elastic Computing Framework. https://confluence.oceanobservatories.org/display/syseng/CIAD+CEI+OV+Elastic+Computing.

[22] OpenShift, Red Hat, Inc. https://openshift.redhat.com/app/.

[23] OpenStack. http://www.openstack.org.

[24] Parallel Workload Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[25] Phorque, GitHub. https://github.com/cu-csc/phorque.

[26] RightScale, Inc. https://www.rightscale.com.

[27] SGI UV, SGI, Inc. http://www.sgi.com/products/servers/uv/.

[28] SimPy Python Simulation Package. http://simpy.sourceforge.net/.

[29] SSH Filesystem. http://fuse.sourceforge.net/sshfs.html.

[30] Windows Azure. http://www.windowsazure.com/.

[31] XSEDE. https://www.xsede.org.

[32] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, JosÈ Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: the in-vigo system. Future Generation Computer Systems, 21(6):896 – 909, 2005.

[33] A Agarwal, R Desmarais, I Gable, D Grundy, D P-Brown, R Seuster, DC Vanderster, A Charbonneau, R Enge, and R Sobie. Deploying hep applications using xen and globus virtual workspaces. In Journal of Physics: Conference Series, volume 119, page 062002. IOP Publishing, 2008.

[34] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. Parallel Computing, 28(5):749 – 771, 2002.

[35] William E Allcock, Ian Foster, and Ravi Madduri. Reliable data transport: A critical service for the grid. In Building service based grids workshop, Global Grid Forum, volume 11. Citeseer, 2004.

[36] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic Acids Research, 25(17):3389–3402, 1997.

[37] David P. Anderson. Boinc: A system for public-resource computing and storage. In Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[38] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. Commun. ACM, 45:56–61, November 2002.

[39] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, February 2009.

[40] Patrick Armstrong, Ashok Agarwal, A. Bishop, Andre Charbonneau, Ronald J. Desmarais, K. Fransham, N. Hill, Ian Gable, S. Gaudet, S. Goliath, Roger Impey, C. Leavett-Brown, J. Ouellete, M. Paterson, C. Pritchet, D. Penfold-Brown, Wayne Podaima, D. Schade, and Randall J. Sobie. Cloud scheduler: a resource manager for distributed compute clouds. CoRR, abs/1007.0050, 2010.

[41] Alvin AuYoung, Brent Chun, Alex Snoeren, and Amin Vahdat. Resource allocation in federated distributed computing infrastructures. In Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure, volume 9, 2004.

[42] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. SIGOPS Oper. Syst. Rev., 37:164–177, October 2003.

[43] Pete Beckman, Suman Nadella, Nick Trebon, and Ivan Beschastnikh. Spruce: A system for supporting urgent high-performance computing. In Patrick Gaffney and James Pool, editors, Grid-Based Problem Solving Environments, volume 239 of IFIP International Federation for Information Processing, pages 295–311. Springer Boston, 2007.

[44] John Bresnahan, Tim Freeman, David LaBissoniere, and Kate Keahey. Managing appliance launches in infrastructure clouds. In TeraGrid, Salt Lake City, UT, 2011.

[45] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. ACM Trans. Comput. Syst., 15:412–447, November 1997.

[46] J. Gregory Caporaso, Kyle Bittinger, Frederic D. Bushman, Todd Z. DeSantis, Gary L. Andersen, and Rob Knight. Pynast: a flexible tool for aligning sequences to a template alignment. Bioinformatics, 26(2):266–267, 2010.

[47] J Gregory Caporaso et al. Qiime allows analysis of high-throughput community sequencing data. Nature Methods, 7(5):335–336, 04 2010.

[48] Charlie Catlett, William E Allcock, Phil Andrews, Ruth Aydt, Ray Bair, Natasha Balac, Bryan Banister, Trish Barker, Mark Bartelt, Pete Beckman, et al. Teragrid: Analysis of organization, system architecture, and middleware enabling new types of applications, 2007.

[49] David Cheriton. The v distributed system. Commun. ACM, 31:314–333, March 1988.

[50] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. SIGCOMM Comput. Commun. Rev., 33:3–12, July 2003.

[51] J. Cope, M. Oberg, H.M. Tufo, T. Voran, and M. Woitaszek. High throughput grid computing with an ibm blue gene/l. In Cluster Computing, 2007 IEEE International Conference on, pages 357–364, 2007.

[52] Jason Cope and Henry Tufo. Adapting grid services for urgent computing environments. July 2008.

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

[54] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1459 of Lecture Notes in Computer Science, pages 62–82. Springer Berlin / Heidelberg, 1998.

[55] Marcos Dias de Assuncao, Alexandre di Costanzo, and Rajkumar Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In Proceedings of the 18th ACM international symposium on High performance distributed computing, HPDC '09, pages 141–150, New York, NY, USA, 2009. ACM.

[56] Kalyanmoy Deb. Multiobjective Optimization Using Evolutionary Algorithms. Wiley, 2001.

[57] Kalyanmoy Deb. Multi-objective optimization. In Edmund K. Burke and Graham Kendall, editors, Search Methodologies, pages 273–316. Springer US, 2005.

[58] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: The montage example. In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, pages 1 –12, 2008.

[59] Robert C. Edgar. Search and clustering orders of magnitude faster than blast. Bioinformatics, 26(19):2460–2461, 2010.

[60] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's EC2. In Cloud Computing and Its Applications, Chicago, IL, October 2008.

[61] Dror Feitelson. Packing schemes for gang scheduling. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1162 of Lecture Notes in Computer Science, pages 89–110. Springer Berlin / Heidelberg, 1996.

[62] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In Hai Jin, Daniel Reed, and Wenbin Jiang, editors, Network and Parallel Computing, volume 3779 of Lecture Notes in Computer Science, pages 2–13. Springer Berlin / Heidelberg, 2005.

[63] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. International Journal of High Performance Computing Applications, 15(3):200–222, Fall 2001.

[64] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in hpc systems. In Workshop on System-level Virtualization for HPC (HPCVirt), 2007.

[65] W. Gentzsch. Sun grid engine: towards creating a compute power grid. In Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, pages 35 –36, 5 2001.

[66] Devarshi Ghoshal, Richard Shane Canon, and Lavanya Ramakrishnan. I/o performance of virtualized cloud environments. In Proceedings of the second international workshop on Data intensive computing in the clouds, DataCloud-SC '11, pages 71–80, New York, NY, USA, 2011. ACM.

[67] Belinda Giardine, Cathy Riemer, Ross C. Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, Webb Miller, W. James Kent, and Anton Nekrutenko. Galaxy: A platform for interactive large-scale genome analysis. Genome Research, 15(10):1451–1455, 2005.

[68] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. ACM Trans. Comput. Syst., 18:229–262, August 2000.

[69] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. Systems, Man and Cybernetics, IEEE Transactions on, 16(1):122 –128, jan. 1986.

[70] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing, 22(6):789–828, 1996.

[71] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running hpc applications in public clouds. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 395–401, New York, NY, USA, 2010. ACM.

[72] Robert Henderson. Job scheduling under the portable batch system. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 949 of Lecture Notes in Computer Science, pages 279–294. Springer Berlin / Heidelberg, 1995.

[73] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In Grid Computing, 2009 10th IEEE/ACM International Conference on, pages 26–33, 2009.

[74] Wei Huang, Matthew J. Koop, Qi Gao, and Dhabaleswar K. Panda. Virtual machine aware communication libraries for high performance computing. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07, pages 9:1–9:12, New York, NY, USA, 2007. ACM.

[75] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K Panda. A case for high performance computing with virtual machines. In Proceedings of the 20th annual international conference on Supercomputing, pages 125–134. ACM, 2006.

[76] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The xtreemfs architecture—a case for object-based file systems in grids. Concurrency and Computation: Practice and Experience, 20(17):2049–2060, 2008.

[77] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 2221 of Lecture Notes in Computer Science, pages 87–102. Springer Berlin / Heidelberg, 2001.

[78] Gideon Juve and Ewa Deelman. Automating application deployment in infrastructure clouds. Cloud Computing Technology and Science, IEEE International Conference on, 0:658–665, 2011.

[79] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Data sharing options for scientific workflows on amazon ec2. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[80] Kate Keahey, Patrick Armstrong, John Bresnahan, David LaBissoniere, and Pierre Riteau. Infrastructure outsourcing in multi-cloud environment. In Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, 2012.

[81] Kate Keahey, Ian Foster, et al. Virtual workspaces: Achieving quality of service and quality of life in the grid. Scientific Programming, 13(4):265–275, 01 2005.

[82] Kate Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. eScience, IEEE International Conference on, 0:301–308, 2008.

[83] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxcluster: a closely-coupled distributed system. ACM Trans. Comput. Syst., 4:130–146, May 1986.

[84] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In Proceedings of the 4th ACM European conference on Computer systems, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[85] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. ArXiv e-prints, January 2009.

[86] Laurent Lefèvre and Anne-Cécile Orgerie. Designing and evaluating an energy efficient cloud. The Journal of Supercomputing, 51:352–373, 2010.

[87] Chunlin Li, Layuan Li, and Zhengding Lu. Utility driven dynamic resource allocation using competitive markets in computational grid. Advances in Engineering Software, 36(6):425 – 434, 2005.

[88] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. SPEEDUP journal, 11(1):36–40, 1997.

[89] Cam Macdonell and Paul Lu. Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. In Proc. of the 2007 High Performance Computing and Simulation Conf, 2007.

[90] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 49:1–49:12, New York, NY, USA, 2011. ACM.

[91] Marcel Margulies et al. Genome sequencing in microfabricated high-density picolitre reactors. Nature, 437(7057):376–380, 09 2005.

[92] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic Site: Using clouds to elastically extend site resources. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society.

[93] Paul Marshall, Kate Keahey, and Tim Freeman. Improving utilization of infrastructure clouds. In Proceedings of the 2011 11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '11. IEEE Computer Society, 2011.

[94] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 1085 –1094, may 2012.

[95] Paul Marshall, Henry Tufo, Kate Keahey, David LaBissoniere, and Matthew Woitaszek. Architecting a large-scale elastic environment: Recontextualization and adaptive cloud services for scientific computing. In Proceedings of the 2012 7th International Conference on Software Paradigm Trends, ICSOFT, 2012.

[96] Paul Marshall, Matthew Woitaszek, Henry M. Tufo, Rob Knight, Daniel McDonald, and Julia Goodrich. Ensemble dispatching on an ibm blue gene/l for a bioinformatics knowledge environment. In Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS '09, pages 13:1–13:8, New York, NY, USA, 2009. ACM.

[97] Ronald Minnich and Jim Mckie. Experiences porting the plan 9 research operating system to the ibm blue gene supercomputers. Computer Science - Research and Development, 23:117–124, 2009.

[98] Justin Moore, David Irwin, Laura Grit, Sara Sprenkle, and Jeff Chase. Managing mixed-use clusters with cluster-on-demand. Technical report, Technical report, Duke University, Department of Computer Science, 2002.

[99] S.J. Mullender, G. van Rossum, A.S. Tananbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. Computer, 23(5):44 –53, May 1990.

[100] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. Intel Technology Journal, 10(3):167–177, 2006.

[101] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on, pages 124 –131, may 2009.

[102] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In Cloud Computing, volume 34 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 115–131. Springer Berlin Heidelberg, 2010.

[103] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The sprite network operating system. Computer, 21(2):23 –36, 1988.

[104] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from bell labs. In Proceedings of the summer 1990 UKUUG Conference, pages 1–9, 1990.

[105] Ruth Pordes, Don Petravick, Bill Kramer, Doug Olson, Miron Livny, Alain Roy, Paul Avery, Kent Blackburn, Torre Wenaus, Frank Wrthwein, Ian Foster, Rob Gardner, Mike Wilde, Alan Blatecky, John McGee, and Rob Quick. The open science grid. Journal of Physics: Conference Series, 78(1):012057, 2007.

[106] Morgan N. Price, Paramvir S. Dehal, and Adam P. Arkin. Fasttree: Computing large minimum evolution trees with profiles instead of a distance matrix. Molecular Biology and Evolution, 26(7):1641–1650, 2009.

[107] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the simos approach. Parallel Distributed Technology: Systems Applications, IEEE, 3(4):34 –43, 1995.

[108] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the chorus distributed operating systems. Computing Systems, 1:39–69, 1991.

[109] P. Ruth, P. McGachey, and Dongyan Xu. Viocluster: Virtualization for dynamic computational domains. Cluster Computing, IEEE International Conference on, 0:1–10, 2005.

[110] Spencer Shepler, Mike Eisler, David Robinson, Brent Callaghan, Robert Thurlow, David Noveck, and Carl Beame. Network file system (nfs) version 4 protocol. Network, 2003.

[111] G. Singh, C. Kesselman, and E. Deelman. Adaptive pricing for resource reservations in shared environments. In Grid Computing, 2007 8th IEEE/ACM International Conference on, pages 74 –80, 2007.

[112] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. Performance impact of resource provisioning on workflows. University of Southern California available at http://www. cs. usc. edu/Research/TechReports/05-850. pdf, pages 05–850, 2005.

[113] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In Proceedings of the 16th international symposium on High performance distributed computing, pages 117–126. ACM, 2007.

[114] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In Proceedings of the 16th international symposium on High performance distributed computing, HPDC '07, pages 117–126, New York, NY, USA, 2007. ACM.

[115] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. Internet Computing, IEEE, 13(5):14 –22, sept.-oct. 2009.

[116] A. Stamatakis, T. Ludwig, and H. Meier. Raxml-iii: a fast program for maximum likelihood-based inference of large phylogenetic trees. Bioinformatics, 21(4):456–463, 2005.

[117] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Beowulf cluster computing with linux. chapter Condor: a distributed job scheduler, pages 307–350. MIT Press, Cambridge, MA, USA, 2002.

[118] S. Vinoski. Advanced message queuing protocol. Internet Computing, IEEE, 10(6):87 –89, 2006.

[119] Matthew Woitaszek, Travis Metcalfe, and Ian Shorrock. Amp: a science-driven web-based application for the teragrid. In Proceedings of the 5th Grid Computing Environments Workshop, GCE '09, pages 1:1–1:7, New York, NY, USA, 2009. ACM.

[120] Matthew Woitaszek and Henry Tufo. Developing a cloud computing charging model for high-performance computing resources. In 10th IEEE International Conference on Computer and Information Technology, Bradford, UK, June 2010.

[121] Jia Yu, R. Buyya, and Chen Khong Tham. Cost-based scheduling of scientific workflow applications on utility grids. In e-Science and Grid Computing, 2005. First International Conference on, pages 8 pp. –147, 2005.

[122] Qian Zhu and Gagan Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 304–307, New York, NY, USA, 2010. ACM.

Appendix A

Individual Traces of FutureGrid Deployments for the Denoising Bioinformatics Workload

The denoising workload evaluations presented in Chapter 8 examines the use of the N-preemptive provisioning policy with values from N=1 to N=64. Individual traces of these evaluations are included here, showing job information and VM information, including the amount of data transferred from the head node to worker nodes (TX) and the amount of data transferred from the worker nodes back to the head node (RX).

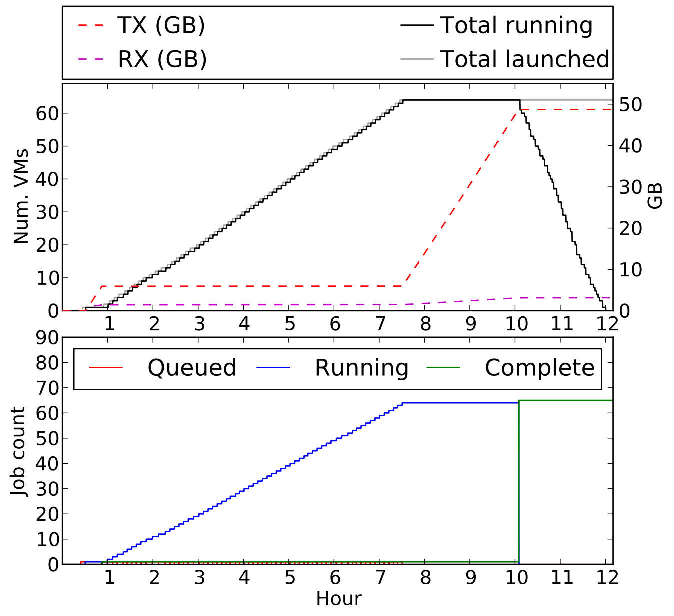

Figure A.1: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=1 (i.e., on-demand).

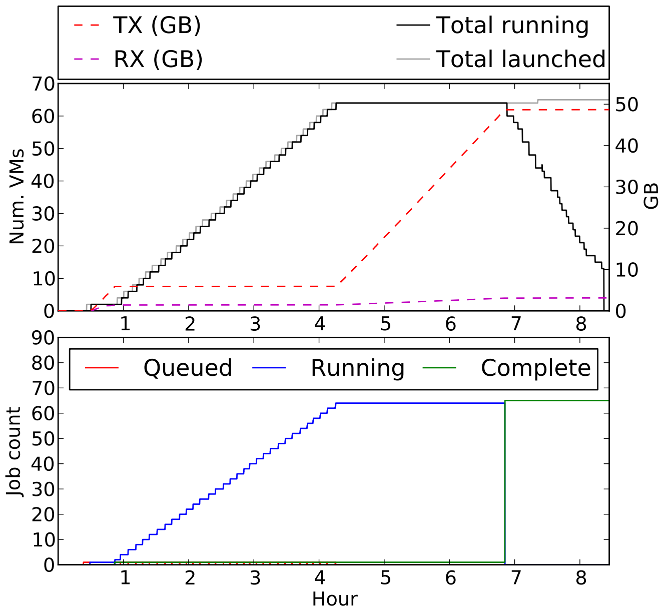

Figure A.2: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=2.

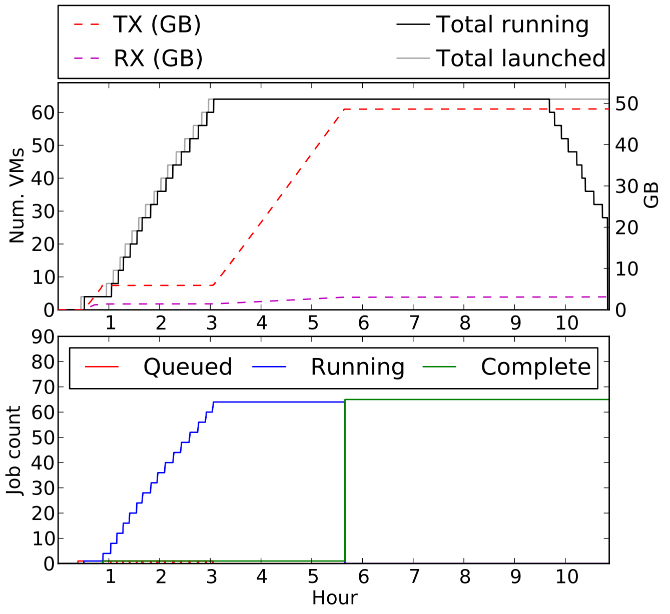

Figure A.3: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=4.

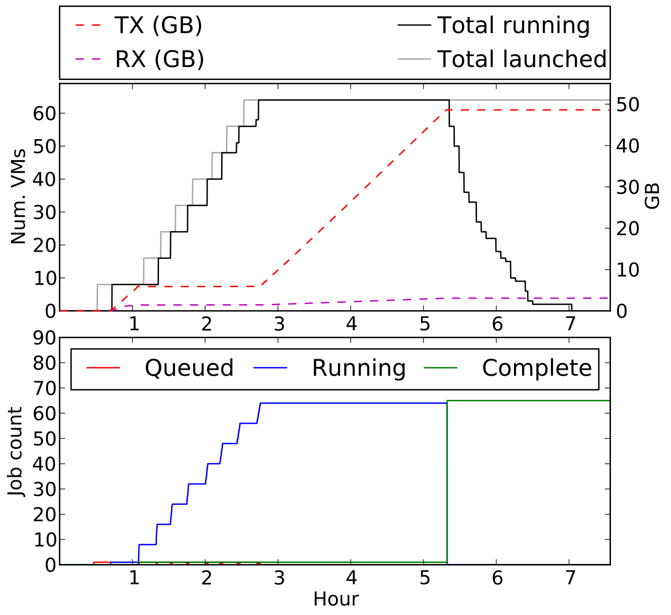

Figure A.4: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=8.

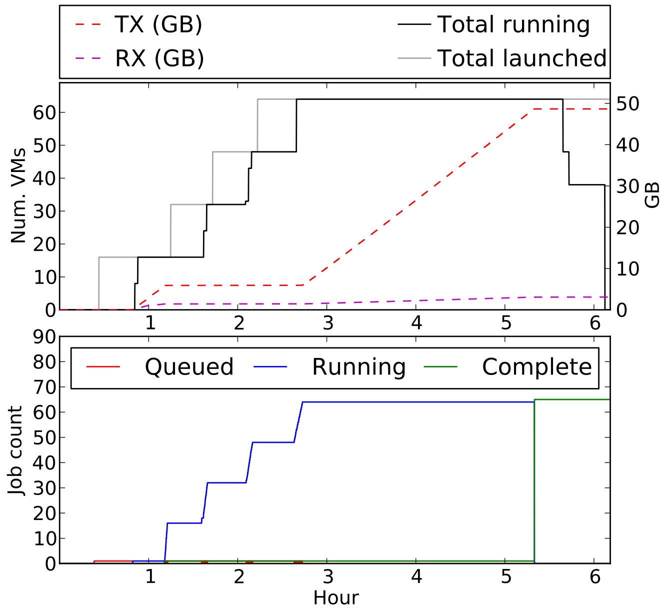

Figure A.5: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=16.

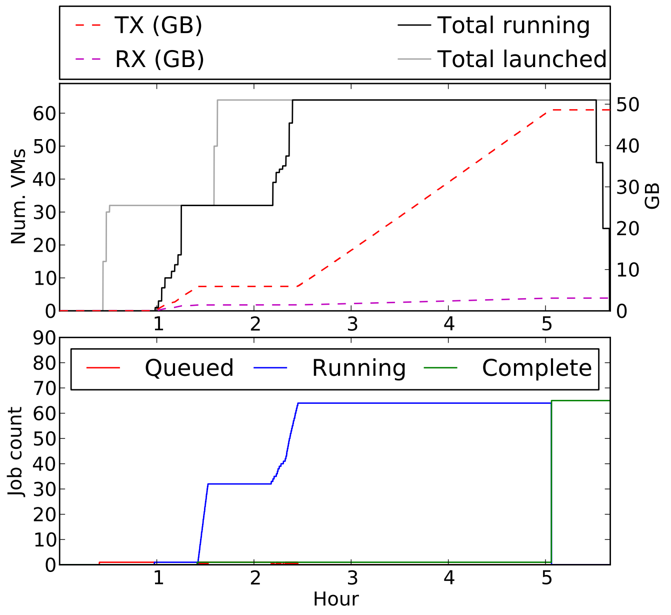

Figure A.6: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=32.

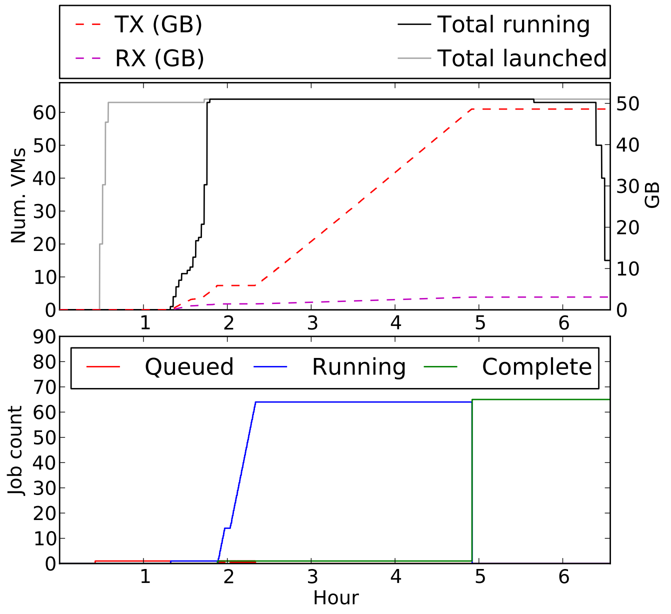

Figure A.7: Job and VM trace for the denoising workload using 64 instances on the Hotel cloud at the University of Chicago and the N-preemptive provisioning policy with N=64.