Optimizing Computations and Allocations on High Performance and Cloud Computing Systems

by

D. Duplyakin

B.S., Samara State Aerospace University, 2008

M.S., University of Colorado at Boulder, 2012

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

2017

This thesis entitled:
Optimizing Computations and Allocations on High Performance and Cloud Computing Systems
written by D. Duplyakin
has been approved for the Department of Computer Science

Prof. Jed Brown

Prof. Kenneth Anderson

Prof. Shivakant Mishra

Prof. Rick Han

Prof. Robert Ricci

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.

Duplyakin, D. (Ph.D., Computer Science)

Optimizing Computations and Allocations on High Performance and Cloud Computing Systems

Thesis directed by Prof. Jed Brown

Over the last decade, many research and development projects have focused on Cloud Computing systems. After forming around the early research papers and the first commercial cloud offerings in 2006-2008, the field has seen a tremendous progress and has provided the primary infrastructure and technology for applications at small, medium, and large scales. Cloud Computing systems have provided diverse on-demand resources to individual researchers and developers, groups and entire institutions, as well as commercial companies and government organizations. Clouds have also found their niche in scientific computing applications, offering attractive alternatives to High Performance Computing models and systems. While cloud economics and technologies have significantly matured recently, there is much active research revolving around topics such as optimality, usability, manageability, and reproducibility in the latest studies. This dissertation presents our findings and relevant developments at the intersection of Cloud Computing and such "flavors" of computing as High Performance Computing and High Throughput Computing. We primarily focus on optimality issues in this area and propose solutions that address the needs of individual researchers and research groups with limited computational and financial resources.

Dedication

To my parents, Vyacheslav and Valentina.

Acknowledgements

I would like to thank my advisor, Prof. Jed Brown, and my committee for the support they provided when I worked on this thesis. Jed provided encouragement and guidance for my work at the University of Colorado. Prof. Robert Ricci became a mentor and a co-advisor when I interned at the School of Computing at the University of Utah and encouraged me to continue my collaboration with his research group. This collaboration has strengthened over the last several years and it is now leading to the next step in my career where I will join the Robert Ricci's research group as a postdoctoral researcher in the Fall 2017.

I would also like to thank my colleagues and collaborators at the University of Colorado for their feedback on my earliest research projects. I would specifically like to thank Prof. Henry Tufo, Michael Oberg, Matthew Woitaszek, and Paul Marshall for supporting my research efforts in the first several years of my Ph.D. program.

Finally, I would like to thank my undergraduate advisor, Prof. Vladimir Fursov, at the Samara State Aerospace University in Russia who encouraged me to pursue my interests in parallel programming and High Performance Computing.

Contents

Tables

Table

Figures

Figure

Glossary

AWS (Amazon Web Services) - Amazons suite of cloud computing technologies and offerings, including EC2 (Elastic Compute Cloud) and S3 (Simple Storage Service). 11

boto - Python library to interface with Amazon Web Services. 20

Chef - configuration management system which is written in Ruby and used to automate and streamline tasks for configuring and maintaining software environments in modern, often cloud-based, distributed systems. Chef was released and maintained by Opscode, the company that is currently also called Chef. 67

CloudLab - NSF-funded "meta-cloud" or a multi-site testbed facility that supports research in Cloud Computing and distributed systems. CloudLab federates computing equipment distributed across three sites: the University of Wisconsin, Clemson University, and the University of Utah. 34

ForestClaw - finite volume library for solving hyperbolic partial differential equations using forest-of-octrees Adaptive Mesh Refinement. 105

FutureGrid - set of NSF-funded cyberinfrastructures designed to provide diverse resources for computer science experimentations in the areas of Grid and Cloud Computing. It consists of both HPC and IaaS resources distributed across multiple sites. The original FutureGrid project ended in 2014, but some of the systems remained operational and served scientific community as part of of a new project called FutureSystems. 10

HPGMG - modern robust benchmark that is representative of High Performance Computing applications. HPGMG implements geometric multigrid methods and includes finite element and finite volume components. 68

HTCondor - open-source resource manager and scheduler for computational jobs which is specifically designed for executing high-throughput computing workloads across distributed cyberinfrastructures. 19

scikit-learn - Python module with a variety of tools for Machine Learning and data analysis. 89

SLURM (Simple Linux Utility for Resource Management) - open-source resource manager and scheduler for computational jobs, both interactive and batch, running on Linux and Unix-like operating systems. SLURM is installed on many supercomputers and computing clusters. 31

Chapter 1

Introduction

Cloud Computing has been a subject of a vast amount of academic and industry research over the last decade. While the earliest relevant work goes back to the 1970s and the IBM VM/370 time-sharing system [35], the field has started forming only in 2006-2008. In 2006, Amazon released the Elastic Compute Cloud (EC2) service and became one of the leaders of a commercial Infrastructure-as-a-Serivce (IaaS) cloud market. Early research papers include [58] and [49] which defined the necessary terminology and set the scope of the new field. In parallel with the tremendous industry development, many influential studies, for instance, [19] and [102], were published in the following years. Those papers significantly moved the field forward by providing comprehensive discussions and evaluations of such critical topics as cloud economics, security, performance and overhead, management, vendor lock-in problem, among others.

Along with further development and frequent release of new features, Cloud Computing systems provided diverse on-demand resources to individual researchers and developers, groups and entire institutions, as well as commercial companies and government organizations. Clouds have also found their niche in scientific computing applications, offering attractive alternatives to High Performance Computing models and systems. In a range of impressive experiments, it is worth mentioning the analysis performed by Cycle Comping where over 70K CPU cores were provisioned in the Amazon EC2 cloud and used to run over 1M hard drive design simulations in 2014 [36].

This dissertation presents our work at the intersection of Cloud Computing and computing "flavors" such as High Performance Computing (HPC) and High Throughput Computing (HTC).

Both types of computing have evolved in the recent years and adapted to use cloud resources. Thus, it was shown as early as in 2011 that improvements in hardware and virtualization software can reduce the virtualization overhead for CPU and memory down to 4% and almost eliminate it for paravirtualized networks [103]. Numerous studies produced similar results, and the performance barrier to using virtualized systems for HPC was essentially eliminated. At the same time, HTC applications, which typically do not have demanding hardware requirements, have seen a tremendous progress due to the wide availability of on-demand cloud resources.

While focusing primarily on management and optimality issues of computing in the cloud, we propose solutions that address the needs of individual researchers and research groups with limited, even scarce, computational and financial resources. In other words, the problems that can be solved at large scale, e.g., in commercial installations and at national laboratories, by assigning additional compute, human, and financial resources, require a different set of solutions at small scale. For instance, while redundant and special-purpose hardware can be acquired for larger projects, in small projects it is common to experience a shortage of resources and only operate commodity hardware. In presence of stringent allocation limits, researchers also need to put extra effort into carefully selecting the most "interesting" experiments in order to obtain the most knowledge using the available cycles.

In this context, we design and evaluate several inexpensive, open and extensible, easy to use solutions that aim to address the following challenges:

- Environments that consist of compute resources provisioned at multiple clouds may need to be periodically rebalanced: when the availability or user preferences change, some resources need to be terminated and replaced with different ones in order to best satisfy current needs. Optimization of the these processes and also interactions between portions of shared cyberinfrastructures is an important and non-trivial task.

- Management of modern cyberinfrastructures, such as clusters and private clouds, requires administering complex distributed software stacks and dealing with failures, including the

failures that impact administrative servers. Configuration management systems must remain operational in the majority of failure scenarios since they are the systems on which system administrators rely in performing most of the recovery actions. Such systems, if not equipped with reliability features by default, need to be augmented with redundancy and failover mechanisms.

- Management of resources provisioned in the cloud requires efficient, transparent, and flexible configuration management tools. Additional overhead related to administration of such tools can create a barrier to their adoption in research environments.

- Efficient evaluation of application performance in clouds and on High Performance Computing systems, especially where a wide range of hardware is available, is a difficult problem. Appropriate optimization techniques need to be found and employed for selecting the most informative experiments and avoiding unnecessary spendings.

1.1 Thesis Statement and Intellectual Contributions

Thesis Statement:

High Performance Computing and Cloud Computing systems offer a number of mechanisms for scientific computing with drastically varying cost and performance characteristics. Such "offline" analysis techniques as discrete event simulations and Active Learning methods can provide valuable insights into optimization of computations and allocations on these systems. With additional engineering effort, these techniques can be developed into "online" decision-making software components capable of providing significant practical improvements in the resource utilization and reuse.

This Ph.D. dissertation presents ours development and evaluation efforts that focus on this "offline" analysis and related infrastructure components that enable it in the context of the specific aforementioned challenges. We recognize that computational and financial budgets are a severe limiting factor in many types of research and development projects. Therefore, in our design

and implementation decisions, we aim to recognize the primary limitations and pursue only the limitation-aware strategies. This dissertation presents several areas of work that are not studied in the existing literature on High Performance Computing or Cloud Computing. We draw additional motivation from the literature on the research testbed systems such as, for instance, Emulab [98], APT [82], and CloudLab [81], and describe the novel capabilities and analysis techniques that can benefit the community of users of such systems.

The primary intellectual contributions that derive from this research are summarized below.

- We design a multi-cloud environment capable of processing High Throughput Computing workloads and, at the same time, rebalancing with the goal of matching user preferences without significant impact on the workloads. Similarly, with the goal of minimizing the negative impacts on the High Throughput and High Performance workloads, we enable the policy-managed resource sharing between the portions of a shared cyberinfrastructure managed by diverse computing frameworks. Several proposed policies are shown to be successful at minimizing the negative side effects of such reuse.

- We demonstrate how a configuration management system can help address customization issues encountered when developing complex distributed software stacks capable of running on multiple hardware platforms.

- We present a combination of Active Learning and Gaussian Process Regression techniques as a powerful method for acquiring experimental data in performance analysis. We show that it is feasible to use these techniques in studying regression problems and demonstrate the tradeoffs between several alternative implementations. We evaluate them in the context of performance benchmarking using a multigrid finite-element code and also for running Adaptive Mesh Refinement simulations, the computations for which the performance characteristics are difficult to predict even for experienced users.

The engineering contributions presented in the following chapters of this dissertation include:

- Development of a multi-cloud rebalancing-enabled computing environment and a set of policies that guide automatic rebalancing.

- Development of elasticity policies and controllers deployed on a prototype implementation running on CloudLab. These software artifacts, along with our experimental evaluation, provide a starting point for the production-level integration of CloudLab partitions with High Performance Computing systems that can take advantage of the cloud-bursting capabilities.

- Integration of a modern, popular configuration management system called Chef [4] into the process of creating new experiments on CloudLab, APT, and related testbeds. We provide a starting point for the development of community infrastructure code and streamlining the process of installing and configuring common software stacks.

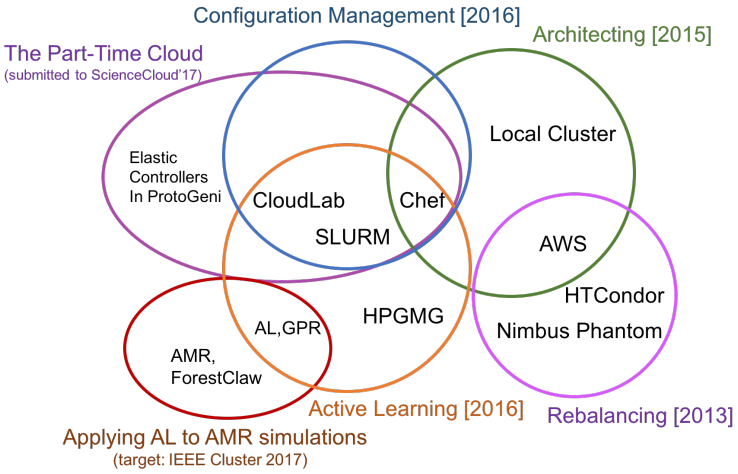
- Development of an Active-Learning-based prototype for empirical modeling of performance data. In addition to the single-realization mode, our prototype is capable of running in the batch-mode where such criteria as the average prediction error and the average cumulative experiment cost allow us to make reliable conclusions about the performance of the proposed Active Learning techniques.

1.2 Technology Landscape

In our research, we leveraged a number of stable and cutting-edge technologies that fit our experimentation and analysis needs. Since 2013, we have published four papers – [42], [41], [43], and [40] – each with its own analysis of the available technologies and the detailed discussion of the selected tools, and recently submitted one paper draft for review. Figure 1.1 depicts the main infrastructure and software components that we utilized in our work and their mapping to the individual studies.

It is worth mentioning that Chef became one of the key enabling technologies in our latest work. It allowed us to streamline and simplify the daunting, error-prone process of configuring

Figure 1.1: Previous and future projects described in this dissertation overlap with respect to the leveraged technologies.



computing clusters on the resources provisioned on CloudLab. Using the developed cookbooks (bundles of installation and configuration scrips in Chef), we build such environments consistently and with minimal effort. Chef will remain one of the technologies used in our future work where it will assist with creation and configuration of elastic experiment on CloudLab. As described in Chapter 8, Active Learning techniques are also expected to play an important role in the future studies. After experimenting with them and evaluating the tradeoffs associated with different Active Learning algorithms in Chapters 6 and 7, we have much stronger intuitions about what components can provide additional advantages with respect to the model fitting and experiment selection, and we will attempt to validate them via experimental evaluations in our future work. The following section describes the scope of the following chapters of this dissertation.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents our work on the rebalancing project, including the architecture of a multi-cloud computing environment with rebalancing capabilities and our design and evaluation of the proposed rebalancing policies. Chapter

3 focuses on our recent system design and analysis efforts for enabling elastic partitions on shared infrastructures. We describe our general-purpose architecture ideas and for implementation and evaluation purposes apply them to the CloudLab testbed. Chapter 4 describes our contributions related to using clouds for running backup components for a configuration management system. Our analysis includes the detailed discussion of the proposed highly available Chef-based architecture from the management, security, and cost perspectives. In Chapter 5, we present our script-based experiment management system that is meant to provide an alternative to the snapshot-based management commonly used on such testbeds as CloudLab. At the core of the proposed system are the Chef configuration management system and a set of developed cookbooks for building computing environments. Chapter 6 introduces a combination of Active Learning and Gaussian Process Regression which we use to model and predict performance of a parallel MPI-based scientific benchmark. Chapter 7 extends the work on Active Learning and our efforts on applying the relevant techniques to Adaptive Mesh Refinement simulations. Chapter 8 concludes this dissertation with a summary of our contributions and a discussion of future work.

Chapter 2

Rebalancing in a Multi-Cloud Environment

With the proliferation of infrastructure clouds it is now possible to consider developing applications capable of leveraging multi-cloud environments. Such environments provide users a unique opportunity to tune their deployments to meet specific needs (e.g., cost, reliability, performance, etc.). Open source multi-cloud scaling tools, such as Nimbus Phantom, allow users to develop such multi-cloud workflows easily. However, user preferences cannot always be achieved at the outset for a variety of reasons (e.g., availability limitations, financial constraints, technical obstacles, etc.). Despite this, it is possible that many preferences can be met at a later time due to the elastic nature of infrastructure clouds. Rebalancing policies, which replace instances in lower-preferred clouds with instances in higher-preferred clouds, are needed to meet these preferences. We present an environment that manages multi-cloud deployment rebalancing by terminating instances, in lower-preferred clouds and launching replacement instances in higher-preferred clouds to satisfy user preferences. In particular, users define a preferred cloud ratio, e.g., 75% instances on one cloud and 25% instances on another, which we attempt to achieve using rebalancing policies. We consider three rebalancing policies: 1) only idle excess instances are terminated, 2) excess instances are terminated gracefully, and 3) worker instances are aggressively terminated, even if they are running user jobs. To gauge the effectiveness of our rebalancing strategy, we evaluate these policies in a master-worker environment deployed across multiple NSF FutureGrid clouds and examine the ability of the policies to rebalance multi-cloud deployments appropriately, and analyze trade-offs.

2.1 Motivation for Rebalancing

Multi-cloud environments leverage infrastructure-as-a-service (IaaS) clouds to integrate resources from multiple cloud infrastructures. These deployments allow users to take advantage of differences in various clouds, including price, performance, capability, and availability differences. As an example, a user may leverage multiple clouds, including community clouds, such as FutureGrid [11], and for-pay public clouds, such as Amazon EC2 [8] (one of the services provided by AWS). A community cloud, for example, often is the user's preferred cloud because it is provided at a reduced monetary cost, or possibly even free. However, it may offer significantly fewer resources than larger for-pay public cloud providers. Users are then faced with a dilemma of choosing where to deploy their instances. In such a scenario, a user may choose to delay deployment, and thus delay processing his workload, until his preferred cloud is available. However, another option is to define an explicit preference for the clouds (e.g., specifying that the less expensive clouds should be used whenever available) but immediately deploy instances wherever possible and then rebalance the deployment at a later time as needed. For example, because community clouds with limited resources are not always available, especially if demand is high, the environment can launch instances on lower-preferred clouds, such as public cloud providers. As clouds with a higher-preference become available, the environment should rebalance, automatically downscaling, that is, terminating instances, in lower-preferred clouds, and upscaling, launching instances in higher-preferred clouds.

Upscaling is typically automated by auto-scaling services, such as Nimbus Phantom [57] or Amazon's Auto Scaling Service [2], which allow users to define the number of instances that should be deployed in each cloud. Auto-scaling services then launch the instances and monitor them, replacing them if they crash or are terminated prematurely. However, downscaling is not typically automated and presents a number of unique challenges that must be addressed. For instance, users must be able to clearly define their preferences for different clouds and policies must be created. These policies should identify which clouds to terminate instances in, which instances to terminate, and how the instances should be terminated (e.g., wait until instances are idle or immediately

terminate instances with running jobs, thus causing jobs to be rescheduled). Rebalancing policies should balance user requirements as well as workload and environment characteristics to avoid introducing excessive workload overhead. To accomplish this, rebalancing implementations must integrate with existing workload resource managers and provide the functionality required by the policies, such as identifying the state of instances (e.g., busy or idle) or marking workers as "offline".

In this chapter, we examine factors that influence rebalancing decisions. Specifically, we consider master-worker environments where resources are dedicated to processing user workloads. In this context, we attempt to identify whether, and if so under what conditions, we may be justified in causing jobs to be killed by terminating running instances for rebalancing purposes. We propose three rebalancing policies that use job and instance information to determine whether or not an instance should be terminated. The first policy waits until instances are idle before they are terminated. The second policy forcibly marks instances offline, allowing them to finish running jobs but preventing them from accepting new jobs, and then terminates the instances once they become idle. The third policy uses a "progress threshold" to decide whether or not to terminate the instances. For example, if the threshold is set to 25% and a job is expected to run for two hours, a node that has been running the job is only eligible for termination during the first 30 minutes of execution.

We also develop a multi-cloud architecture that incorporates these policies with master-worker environments. For experimental evaluation, we deploy our solution using multiple NSF FutureGrid clouds and use Nimbus Phantom for our auto-scaling service. Our evaluation examines the benefits and trade-offs associated with each policy. Less aggressive policies are able to provide zero-overhead rebalancing at the expense of leaving the deployment in a non-desired state for longer periods of time. More aggressive policies, on the other hand, rebalance the environment quickly but introduce workload overhead and delay overall workload execution. The most aggressive policy, however, appears to strike the best balance by rebalancing the environment quickly, reducing cost by up to a factor of 3, while only increasing workload execution time by up to 14.7%.

2.2 Approach

2.2.1 Models and Assumptions

We propose a multi-cloud environment that is capable of processing user demand and distributing work to resources deployed across multiple clouds. For example, such an environment might consists of a pool of web servers, distributed between different cloud data centers, responding to user requests for a single website. Another example is an HTCondor pool with workers distributed between multiple clouds, all pulling jobs from a single, central queue. When deploying such an environment, a user may define a desired request for how many resources to deploy across different clouds throughout the environment. These requests can be expressed in two forms: 1) in terms of absolute numbers of instances needed in selected clouds, e.g., R={32 instances in cloud A, 8 instances in cloud B}; 2) in terms of total numbers of instances and preferred ratios, e.g., R={40 instances total; 80% in cloud A, 20% in cloud B}. However, such requests may lead to situations where a deployment cannot be satisfied, at least initially. For example, instead of matching the request R={32 instances in cloud A, 8 instances in cloud B}, we may have 24 instances in cloud A (which may not be able to launch additional instances) and, thus, end up with 16 instances in cloud B. Therefore, as the environment adapts, additional instances should be launched in cloud A whenever possible and instances in cloud B should be terminated until the users' preferences are met.

We assume that the multi-cloud deployment is deployed and managed by a central auto-scaling service. In particular, we use the open source Phantom auto-scaling service, which is responsible for servicing requests to deploy instances across multiple clouds and monitoring those instances, replacing them if they crash or are terminated prematurely. We also assume that the environment uses a master-worker paradigm to process demand, such as an HTCondor pool, using a "pull" queue model. That is, workers distributed across multiple clouds request jobs from a central queue when they are available to execute jobs. The central scheduler reschedules jobs when workers fail or are terminated for rebalancing. The job scheduler must also be able to: 1) provide status

information about workers, including job state (e.g., busy or idle), jobs running on the workers, and up-to-date job runtimes; 2) provide information about the queue, including a list of running and queued jobs; 3) add an instance to the worker pool; and 4) remove an instance from the worker pool, either gracefully by allowing it to finish running its job or immediately by preemptively terminating the worker and its jobs. Many modern job schedulers, including Torque and HTCondor, provide these capabilities. Because we use a "pull" queue model, only embarrassingly parallel workflows are considered. In such workflows, jobs can be terminated, rescheduled, and re-executed out of order without consideration for other jobs in the set.

In this context, we define the following terms:

- Multi-cloud request: a request that specifies the configuration of a multi-cloud deployment either using absolute numbers (e.g., 32 instances in cloud A and 8 instances in cloud B) or as a ratio (e.g., 40 instances with 80% in cloud A and 20% in cloud B). It is specified by the user and typically represents his desired preferences for various clouds.

- Desired state: when the multi-cloud deployment matches the specified multi-cloud request. For example, if the user specifies a multi-cloud request with 32 instances in cloud A and 8 instances in cloud B and the running deployment matches this request, then it is in the desired state.

- Rebalancing: the process of transitioning the deployment in an attempt to reach the desired state. This occurs when the deployment is not in the desired state so instances are terminated in some clouds and replacement instances are launched in other clouds in order to reach the desired state. Even after the desired state is reached, the system must maintain it. However, it is possible for the system to return to an undesired state. For example, if failed instances cannot be replaced on the preferred cloud and are instead deployed on a less-preferred cloud, then the system will again attempt to reach the desired state.

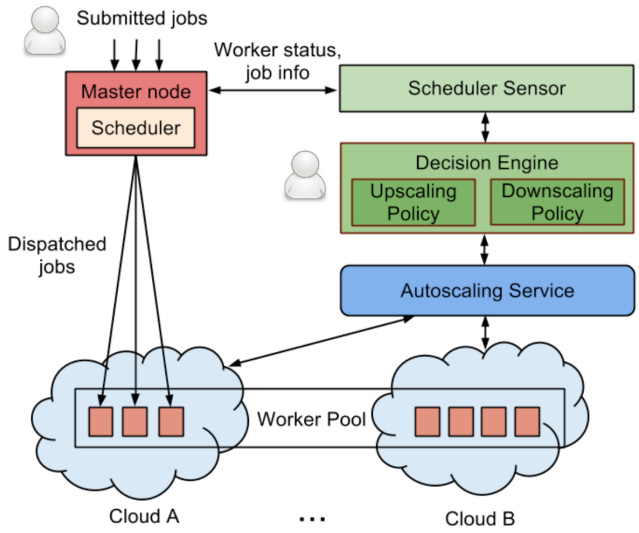
- Downscaling: the termination of running instances that occurs during rebalancing. De-

pending on the difference between the current deployment and the request, there may be a need to terminate many instances (e.g., terminate all instances in cloud A) or only a particular one (e.g., terminate an instance in cloud A with a particular ID). Selecting the appropriate instance to terminate is a non-trivial task. This process has two main components: 1) real-time information about all instances is required, such as jobs currently running on those instances, and 2) specific instances need to be identified for termination according to the policy. For example, a policy may select an instance for termination based on the progress of the job it is executing. An instance that is close to completing its job is considered to be more valuable than an instance that has just started execution of its job. Additionally, if an instance is idle (i.e., it's not currently executing a job) then it may be terminated immediately.

- Upscaling: the process of deploying instances throughout the multi-cloud environment to ensure that the total number of instances specified in a multi-cloud request is satisfied. Upscaling attempts to launch instances on clouds with a higher preference, however, if such clouds are unavailable then upscaling will deploy instances on clouds with lower preferences. Excess instances: instances in a particular cloud that exceed the desired amount. For example, if a multi-cloud request specifies that 15 instances should be deployed with 10 instances in cloud A and 5 instances in cloud B, but at the given time all 15 instances can only be deployed in cloud B, then cloud B would have 10 excess instances.

While there may be cases where it is crucial for multi-cloud deployments to satisfy user preferences before job execution begins, we believe stalling job execution until the environment is in the desired state is not necessary. Instances that are already deployed in less desirable clouds can yield partial results while rebalancing occurs instead of delaying deployment and providing no results at all. A multi-cloud deployment that is running jobs should continue to process jobs in less desirable clouds and rely on rebalancing to achieve the desired state.

Figure 2.1: Multi-cloud architecture with workload and management components where rebalancing is guided by upscaling and downscaling policies.



2.2.2 Architecture

The multi-cloud architecture that we designed extends the architecture presented in [65]. It deploys workers across several cloud resource infrastructures and dynamically balances this deployment based on user-defined preferences. The architecture is depicted in Figure 1 and consists of four main components: (1) a workload management system (including a job scheduler and workers), (2) sensors to monitor demand, (3) policies to scale the number of deployed instances up or down, and (4) an auto-scaling service to enforce the chosen policy.

In this environment, the master node hosts the job scheduler for the workload management system that accepts user jobs and schedules them to run on worker nodes deployed across multiple clouds (see Figure 2.1). The sensor periodically queries the job scheduler and gathers information about the queued jobs and their runtimes as well as worker status. The sensor provides this information to the decision engine, which then executes a policy that decides how to adjust the deployment, potentially downscaling it on one cloud and upscaling on another, to satisfy user preferences.

Users define their preferences for different clouds in their multi-cloud requests, e.g., specifying that they want 20 instances, with half deployed on one cloud and half on another cloud. The decision engine also includes per-cloud timers and predefined time-outs to avoid undesired scenarios where upscaling and downscaling are performed on the same cloud within a short period of time.

To enact upscaling or downscaling, the auto-scaling decision engine instructs the service to deploy, maintain, or terminate workers across the different clouds. The job master and workers operate relatively independently of the multi-cloud architecture, integrating new workers that are deployed, processing jobs on available instances, and rescheduling any jobs that are terminated prematurely. However, in some cases, the decision engine may need to make specific requests of the workload management system. For example, policies that elect to mark workers offline, allowing them to finish their jobs and preventing them from accepting new jobs, must be able to communicate with the workload management system. In our architecture this is accomplished by communicating the request through the sensor, which then performs the operation on the master node.

2.2.3 Rebalancing Policies

Rebalancing alters multi-cloud deployments and may be disruptive from the workload's perspective. For example, terminating an instance may cause a running job to be terminated prematurely, causing the job to be rescheduled for execution on a new worker. The workload's total execution time may increase as a result. On the other hand, executing a user's workload in a multi-cloud environment that is not in the desired state might lead to other problems, including overall performance degradation, unexpected expenses, additional data transfers, etc. Therefore, rebalancing policies are needed for multi-cloud environments; their objective is to achieve the desired state, if possible, and only minimally impact user workloads.

In this work, the existing auto-scaling decision engine performs upscaling and tries to maintain the total number of instances as specified in the multi-cloud request. For downscaling, we propose the following policies:

- Opportunistic-Idle (OI): waits until excess instances in less desired clouds are idle (i.e., not running jobs according to information from the sensor) and then terminates them. This policy continues to terminate excess idle instances until the deployment reaches the desired state. It begins with clouds that have the most excess instances before proceeding to the clouds with fewer.

- Force-Offline (FO): is similar to OI but excess instances are terminated gracefully, that is, jobs are allowed to complete before the instances are terminated. To terminate instances gracefully, the auto-scaling service notifies the jobs scheduler which instances are to be terminated, and the job scheduler then marks them "offline". Graceful termination allows those instances to complete currently running jobs. Once the jobs complete, the workers do not accept any new jobs and can be terminated. This policy requires that the job scheduler support the ability to "offline" workers. FO does not terminate instances once the desired number of instances is reached in each cloud.

- Aggressive Policy (AP): sacrifices work cycles and discards partially completed jobs in order to satisfy requests in the shortest amount of time possible. This policy terminates excess instances even if those instances are currently running jobs. To minimize overhead associated with job re-execution, this policy proceeds in termination from instances with jobs that have been running for the least amount of time to instances with jobs that have been running for a longer time. Additionally, since the amount of work to discard may vary, we include a tunable parameter, work threshold (measured in percent). Work threshold specifies how much progress on its current job with respect to the job's walltime (i.e., expected job's runtime) an instance has to make before this instance may no longer be terminated. For example, if we choose the 25% threshold and one of the instances executes a two-hour job, the policy is only allowed to terminate it for up to 30 minutes after beginning of execution. While job runtimes may be predictable, we avoid relying on the accuracy of such predictions and consider walltimes that are explicitly provided. Job

walltimes are typically limited by the cluster administrator (e.g., to 24 hours), and prevent users from specifying excessive walltime requests.

2.3 Implementation

We leverage a number of existing technologies for our multi-cloud deployment. Specifically, we use infrastructure clouds, such as Nimbus [13] and Amazon EC2 [8], which provide on-demand resource provisioning. Our environment integrates with the open source Phantom service for auto-scaling. Phantom [57] is responsible for servicing multi-cloud requests and deploying the instances across the specified clouds. It also continually monitors the deployment and maintains the requested number of instances. We also rely on a master-worker workload management system, specifically HTCondor [12], which monitors and manages a pool of workers across distributed resources. This includes the ability to submit jobs to a central queue and schedule jobs across distributed workers. HTCondor also includes job resubmission and migration capabilities, which guarantee that every job eventually completes even if some workers are terminated prematurely. Lastly, HTCondor provides the required information about workload execution, specifically, which jobs are queued or running, as well as the amount of time they have been running.

To integrate with the leveraged technologies, we develop two additional components for the implementation: the sensor and the rebalancing policy. The sensor, written in Python, communicates with HTCondor master nodes and obtains necessary job and worker information for the policies using the `condor_status` and `condor_q` commands. This provides three pieces of information: 1) the number of HTCondor workers running in each cloud (e.g., `condor_q -run`), 2) information required to identify idle workers (e.g., `condor_status`), and 3) a list of workers and their current runtimes and walltimes. However, some policies require that the sensor also communicates with the workload management system (e.g., when marking nodes offline). Therefore, the sensor is both able to send information to the policy and receive instructions from it. When the policy instructs the sensor to remove a worker from the pool, the sensor issues the `condor_off` command. To terminate an instance gracefully (e.g., when using FO), the sensor executes the

following command: `condor_off -peaceful <hostname>`. AP removes instances from the pool instantly using: `condor_off -fast <hostname>`.

In addition to workload information, the system must also query the auto-scaling service, Phantom, and the individual IaaS clouds to identify all of the instances in the auto-scale group as well as their distribution across the clouds. The instance IDs reported by Phantom also need to be compared with the instance IDs reported by individual clouds in order to identify the specific clouds auto-scale instances are running on. This information is used by the policy, along with workload information, to guide rebalancing decisions. The policies, written in Python, collect the necessary information and then attempt to match the deployment with the user's multi-cloud request, downscaling on clouds with excess instances and upscaling on high-preferred clouds when needed. To accomplish this, the downscaling component of the policy communicates with the auto-scaling service, using the Python boto library [3], and makes adjustments to the configuration in order to satisfy the request. Phantom then provides an ordered cloud list and capacity parameters that can be adjusted for this purpose. The ordered cloud list specifies the maximum number of instances in each cloud and can be set to match the request for a specific cloud, while the capacity parameter controls the total number of instances across all clouds and can be set independently from the ordered list.

Upscaling is performed by an existing Phantom decision engine, which tries to maintain a total number of instances deployed across the preferred clouds. Phantom implements its own `n_preserving` policy, which allows it to maintain the requested number of running instances, replacing failed instances when needed. When Phantom replaces failed or terminated instances, it does so according to the ordered cloud list, meaning that if instances are terminated in a lower-preferred cloud, it first attempts to deploy replacement instances in a higher-preferred cloud. It only resorts to lower-preference clouds if it is unable to deploy instances in higher-preferred clouds (e.g., due to unavailability).

2.4 Evaluation

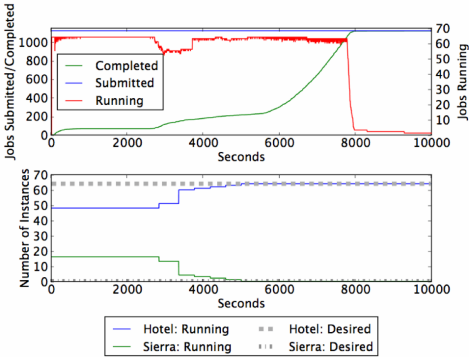
To evaluate proposed rebalancing policies, we examine the ability of the environment to rebalance the deployment in order to satisfy multi-cloud requests and reach the desired state. That is, the policies attempt to adjust the deployment to match user preferences as quickly as possible while avoiding excessive workload overhead. Specifically, we consider the scenario where cloud availability changes over time due to external factors. For example, when other users terminate instances it may be possible to deploy additional instances in higher-preferred clouds and downscale in lower-preferred clouds. We choose not to simulate this type of unexpected change in availability directly; instead we focus on the behavior of our policies once such changes in availability occur. Therefore, in each of our experimental evaluations, we assume that the evaluation begins with the deployment in an undesired state but that higher-preferred clouds now have additional capacity available, allowing the policies to attempt to reach the desired state.

For our environment, we use NSF FutureGrid [11] and two workload traces from the University of Notre Dame's Condor Log Analyzer [7]. HTCondor is used as the workload management system. We leverage its ability to reschedule jobs that are terminated prematurely. On FutureGrid we use the Hotel cloud at the University of Chicago (UC) and Sierra at the San Diego Supercomputer Center (SDSC). Both clouds use Nimbus as the IaaS toolkit and Xen for virtualization. The master node and all the worker nodes run Debian Lenny images, approximately 1 GB compressed, with HTCondor 7.8.0 as the workload manager. The master node VM has two 2.93GHz Xeon cores and 2 GB of RAM. Workers have one 2.93GHz Xeon core and 2 GB of RAM. Instances are contextualized as the master or a worker automatically at boot. In particular, the IaaS userdata field is used to provide the hostname of the master, in which case, the node configures itself as a worker and attempts to join the master. If the field is empty, the instance configures itself as the master.

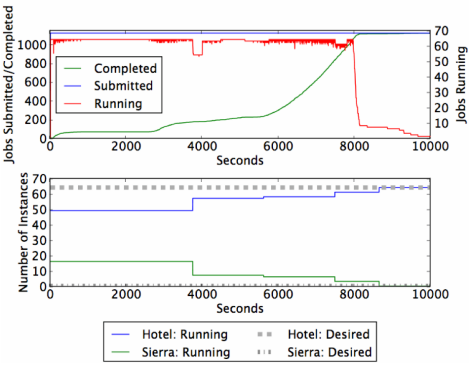
In this evaluation, we differentiate between Hotel and Sierra by specifying different costs for their instances. Specifically, we assume instances on Hotel have 1 unit of cost and instances on

Figure 2.2: Visualization of rebalancing processes guided by the proposed policies.

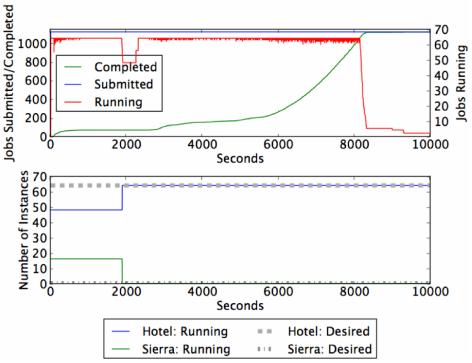
(a) OI terminates 16 instances when jobs complete.



(b) Gradual downscaling with FO.



(c) Considerate aggressive downscaling with AP-25.



(d) Fast aggressive downscaling with AP-100.

Sierra have 2 units of cost. We also specify a multi-cloud request for a total of 64 instances with 100% of the instances on Hotel and 0% on Sierra, representing the case where a user desires that all of his instances be deployed in the less expensive cloud. However, for each experiment, we initialize the environment to have 75% of the instances in Hotel (48 instances) and 25% of the instances in Sierra (16 instances), requiring rebalancing to occur in order to reach the desired state.

For a job trace, we combine two traces from the HTCondor Log Analyzer [7], one consisting primarily of smaller jobs and another that contains longer running jobs. The traces are combined into a single workload that is submitted immediately at the beginning of the evaluation by selecting jobs randomly from each trace. With this approach we consider a workload that consists of a variety of job runtimes. Individual jobs are submitted as sleep jobs, which sleep for the runtime specified

in the trace. The combined workload consists of 1120 jobs, with a minimum runtime of 54 seconds and a maximum runtime of 119 minutes. The median runtime is 106 seconds and the mean is 443 seconds.

The system can be configured to execute the policy at any user-defined interval. For this evaluation, we configure the policy to execute every 30 minutes, beginning 30 minutes after the initial job submission, in order to rebalance the deployment regularly while still allowing for some jobs to complete between rebalancing intervals. Exploring different policy execution intervals and their impact is left for future work. We expect overly aggressive intervals (i.e., a short policy execution interval) and relatively passive intervals (i.e., a long policy execution interval) to negatively impact the environment by either rebalancing too frequently and preventing jobs from finishing or not rebalancing often enough, causing high excess cost. However, further experimentation is needed to identify the appropriate balance between the policy execution interval and workload characteristics, including the rate that jobs are submitted and the duration they execute.

For experimental evaluation, we define the following metrics:

- Workload execution time: the amount of time required for the entire workload to complete, that is, the amount of time from when the first job is submitted until the time the last job completes.

- Workload overhead percent: the total percent of time jobs run before being terminated prematurely. For example, if a two-hour job runs for 30 minutes before it is terminated, causing it to be re-queued and needing to be rerun, then the job experiences 25% overhead (assuming that the second run finishes).

- Convergence time: the amount of time it takes for the deployment to rebalance from a non-desired state to the desired state once the evaluation begins. For example, if the system is operating in an undesired state at the moment the first job starts execution, convergence time is the time from that moment until the system reaches the desired state (i.e., the specified multi-cloud preference is satisfied).

- Excess cost (EC): the total user-defined cost associated with running excess instances. EC is described as:

$$EC = \sum_{r}^{R} \sum_{i}^{I_r} c_r * p_r * (t_i - l_i).$$

The variables are defined as follows: R – set of all cloud resources used for the deployment, I_r – set of all excess instances for every cloud resource r, c_r – user-defined instance cost for cloud resource r, p_r – time accounting function for cloud resource r, t_i – termination time for instance i, l_i – launch time for instance i.

R consists of all clouds that have running instances. I_r is a set containing all of the excess instances for cloud resource r. c_r represents the user's definition of cost associated for an excess instance in cloud resource r, for one unit of time. Depending on the cloud resource r, pr may differ to represent various time accounting methods, such as per-second usage or rounding up to the nearest hour, etc. t_i and l_i are the specific instance termination and launch times. Intuitively, EC is intended to represent the cost of running excess instances, that is, using instances on clouds beyond the amount specified in the multi-cloud request. EC offers a fine-grained metric compared to convergence time, which only provides a course representation of when the environment finally reaches the desired state.

In addition to these metrics, we also include a set of job traces that show the number of instances running as well as the number of jobs submitted, running, and complete. As described earlier, we specify a multi-cloud request with a preference for 64 workers in Hotel and 0 in Sierra, but the environment is initialized in an undesired state with 48 workers running in Hotel and 16 workers in Sierra. Therefore, the rebalancing policies attempt to terminate all workers in Sierra, while Phantom replaces the instances in Hotel until it has 64 running worker instances.

2.4.1 Understanding Deployment Transformations

All policies pursue the same goal: downscaling 16 instances on Sierra, the lower-preferred cloud, and upscaling on Hotel, the higher preferred cloud. Traces are included for the four different

policies, OI (Figure 2), FO (Figure 3), AP-25 (Figure 4), and AP-100 (Figure 5). AP-25 uses a 25% threshold and AP-100 uses a 100% threshold. The traces show job information (jobs completed, submitted and running), as well as the distribution of worker instances across Hotel and Sierra.

In the experiment shown in Figure 2, OI only attempts to terminate idle instances. OI is first able to perform downscaling approximately 9000 seconds after initial job submission, that is, when all 16 instances in Sierra are idle and can be terminated after the entire workload has been processed. This downscaling has no effect on the workload since the few jobs that are still running at the time of downscaling occupy several of Hotel's instances (where downscaling does not occur) and continue to run unaffected. This downscaling policy may be valuable if the user continues to use the same deployment and executes another workload at a later time. Then, the deployment has already been adjusted and, if no failures happen, all new jobs are executed in the rebalanced environment. In general, OI is only useful in situations where user jobs are submitted in a series of batches that are interleaved with periods of idle time when rebalancing can occur (i.e., OI will not work on a fully utilized system).

Figure 3 illustrates an experiment where FO terminates instances gracefully. This policy marks all 16 excess instances "offline" during its first evaluation at 1800 seconds after initial job submission. Instances that are executing jobs at that time must wait for jobs to finish before the instances can be removed from the worker pool and terminated. Thus, after 2300 seconds, 3 instances are terminated, after 2800 seconds 8 additional instances are terminated, etc. Rebalancing continues until there are no remaining instances in Sierra, which occurs when the last instance is finally terminated after 4600 seconds. Every termination causes a noticeable drop in the trace showing the number of running jobs. These drops indicate temporary reductions in the worker pool that follow downscaling actions and last until replacement instances boot and join the worker pool. Since there are less than 64 running instances at certain periods of time, it takes longer for the workload to complete in this experiment than for the OI experiment, which rebalances after the workload completes. However, we do observe a faster convergence time for FO than OI; all 64 instances are running on Hotel after 4600 seconds. FO demonstrates an average execution time

increase of 12.6% and an average convergence time decrease of 48.4% with respect to OI's time characteristics (Figure 6).

We evaluate two variations of AP: AP-25 and AP-100. The first, AP-25, terminates excess instances only if jobs running on those instances have been running for less than 25% of their requested walltime. We choose to use the 25% threshold to demonstrate a considerate implementation of AP, which attempts to achieve low overhead rather than fast convergence. In contrast, AP-100, having no threshold, terminates all excess instances without consideration of job progress (specifically, AP-100 terminates instances with jobs that have been running for less than 100% of their requested walltime). This is a special case when the user prefers to rebalance as fast as possible regardless of the amount of work that is discarded.

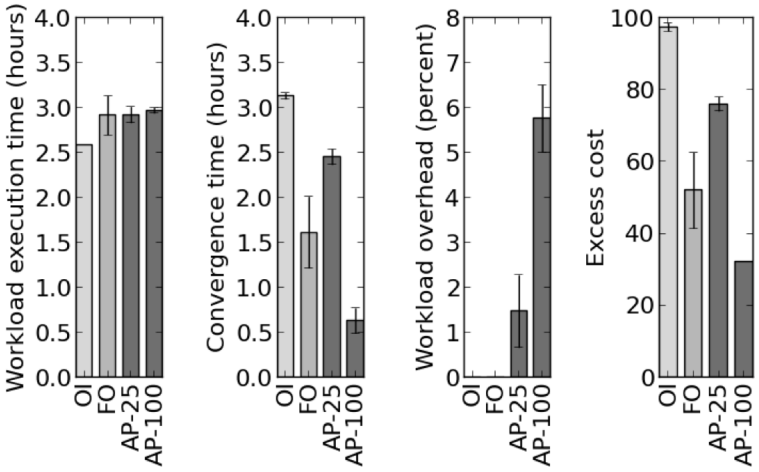
Figure 4 shows that AP-25 cannot perform the necessary adjustments all on the first try, and thus, rebalancing is a gradual process, similar to FO. A 25% threshold yields premature termination of 16 jobs that are running on Sierra's instances, which individually have been running for 267 seconds on average. The maximum runtime among those jobs is 933 seconds. In this experiment the total amount of discarded work cycles is about 71 minutes, which is only 0.9% of the total CPU time used by all 64 instances throughout the evaluation.

Figure 5 shows AP-100, which converges to the desired state the fastest. This policy terminates all 16 instances in Sierra during its first evaluation after 1800 seconds. At that time, 16 jobs are terminated prematurely after running for an average of 28 minutes. The maximum runtime is over 30 minutes. 458 minutes of discarded work result in 5.2% workload overhead.

2.4.2 Understanding Trade-offs

Figure 2.3 shows workload execution time, convergence time, percent workload overhead, and excess cost for all four policies. We calculate mean values and standard deviations for these metrics using a series of experiments with three iterations for each policy. OI provides the shortest workload execution time, while other policies introduce a noticeable increase in the execution time. This is because OI does not affect the workload, as described earlier, and waits for the entire workload

Figure 2.3: Mean workload execution time (hours), convergence time (hours), workload overhead (percent), and excess cost. Three iterations are run for each policy and no error bar indicates very little deviation. OI and FO also have 0% workload overhead.



to complete before rebalancing occurs. Switching from OI to FO, AP-25 and AP-100 introduces an average workload execution increase of 12.6%, 12.9% and 14.7%, respectively. FO, AP-25 and AP-100 all appear to have comparable execution times, while OI consistently provides the lowest execution time with negligible variance (indicated by no error bar in the graph).

OI's convergence, shown in Figure 6, happens after workload completion, specifically, at 3.1 hours. FO's gradual downscaling completes after 1.6 hours. AP-25's convergence completes after 2.5 hours. As expected, AP-100 provides the fastest convergence time, which is approximately 0.6 hours.

For workload overhead (that is, the amount of discarded work), OI and FO have none (indicated with zero-height bars in the rightmost graph). By design, these policies never attempt to terminate instances when it leads to premature job termination. AP-25 and AP-100, however, are designed to accomplish downscaling via premature termination, demonstrate average workload overheads of 1.5% and 5.7%, respectively. The highest observed overheads are 2.4% for AP-25 and 6.6% for AP-100.

Finally, we also consider the excess cost, EC, of the experiments (Figure 6). As described

earlier, we assume c_r for Hotel to be 1 unit of cost and c_r for Sierra to be 2 units of cost. However, it should be noted that this cost may differ for users and could instead correspond to a user's weighted preference or a dollar amount for instances. We consider a pr function that rounds up instance usage to the nearest hour and obtain ti and li times for instances from our experiment logs. In Figure 6, OI has the highest EC at 97.3 units of cost on average, since rebalancing does not occur until after the workload completes and, thus, requires excess instances to run for the majority of the experiment. Other policies incur lower EC than OI because they perform rebalancing earlier. FO has an average EC of 52.0 units of cost, AP-25 has an average EC of 76.0 units of cost and AP-100 has an average EC of 32.0 units of cost since it rebalances completely at the first policy execution, 30 minutes into the experiment.

OI, FO, and AP-100 each have different advantages and trade-offs, for example, FO converges faster than OI and both have no workload overhead but OI has the shortest workload execution time. AP-25, on the other hand, doesn't offer significant advantages; it has comparable workload execution time to FO but higher convergence time, workload overhead, and EC. AP-100, however, appears to strike the best balance between quick rebalancing and minimizing excess workload overhead and execution time. Specifically, AP-100 reduces EC by a factor of 3 over OI while introducing only 6.6% workload overhead and 14.7% workload execution time.

2.5 Related Work

Much work has been done on leveraging elastic capabilities of infrastructure clouds based on user preferences, performance, and scalability. To upscale or downscale cloud deployments, researchers have proposed systems that predict workload computational needs during execution [85], [91], [101]. These systems take two approaches: model-based and rule-based [51]. Our purpose was not to predict workload execution, but to monitor the workload execution through HTCondor sensors and rebalance the environment effectively based on our policies. Typically, however, rebalancing is motivated by cost, where the environment terminates idle instances to avoid excessive charges. As another example, some policies govern rebalancing by triggering live migration from

one cloud to another [92]. Our work is more general than such live migration approaches; in our research, workload migration is based on predefined user preferences for different clouds and aims to satisfy these preferences before the workload completes. Existing policy frameworks that execute auto-scaling policies can be adapted to include our policies [57]. Our framework relies on similar services to guide the deployment toward the user's desired state. There are also projects that examine different policies for efficient demand outsourcing from local resources to IaaS clouds [68], [71], but our work focuses on policies governing downscaling behavior in multi-cloud environments, gradually achieving the user's desired state.

2.6 Acknowledgments

This material in this chapter is based on work supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Chapter 3

Enabling Balanced Elasticity Between Diverse Computing Environments

Clouds, HPC clusters, HTC systems, and testbeds all serve different parts of the computing ecosystem: each are designed for different types of workloads and suited to different types of research and commercial users. We propose that an effective way to share resources among these diverse applications is to not shoehorn them all into the same resource management framework, but to partition a common hardware substrate among different frameworks: for example, to have part of the cluster managed by a cloud framework such as OpenStack, part of it managed by an HPC scheduler such as SLURM, etc. In order to efficiently manage such a shared resource, it must be possible to adjust the set of resources controlled by each in an elastic manner.

While resource allocation and scheduling within each of these types of environments are well studied, what we consider in this chapter is elasticity between them. Our goal is to enable each management framework to separately manage the resources currently within its own domain, scheduling jobs, VMs, etc. according to its own needs and policies. At the same time, the frameworks can all coordinate with one another so that when resources must be moved between them, it can be done in the most fair and efficient manner possible. We evaluate our ideas using a prototype that shares resources between a testbed and an HPC cluster, and with simulations using real workload traces. What we find is that with only minimal information flow, it is possible to elastically adjust resource assignments while each framework optimizes for its own internal criteria.

The design and analysis efforts described in this chapter are similar in style to the work presented in the previous chapter. In Chapter 2 and [42], we focus on reprovisioning processes that

can occur in multi-cloud systems, whereas below we describe the interactions between portions of the same shared cyberinfrastructure.

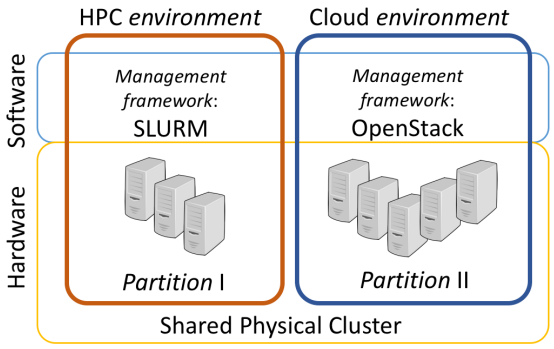
3.1 Motivation for Balancing Elasticity

Resource management frameworks for multi-tenant compute environments are generally best-suited to the types of jobs for which they were originally designed. For example, HPC schedulers such as SLURM [87] are specialized to schedule batch jobs of various sizes and durations within a tightly-coupled cluster (e.g. with a high-speed Infiniband or Ethernet network). HTC systems such as HTCondor [53] focus on less tightly-coupled jobs. Cloud management software such as OpenStack [77] focuses on virtualization and is primarily designed for web services, general server hosting, and multi-tier applications. Testbeds such as those based on the Emulab [98] framework focus on bare-metal allocation and physical isolation between tenants.

While it is possible, to some extent, to run jobs intended for one of these environments on top of another, the results are often suboptimal. For example, the virtualization overheads associated with cloud computing often add variable overhead to the networking stack and scheduling noise to the CPU, interfering with tightly-coupled computations (see e.g., [55] and [90]). Other combinations are impossible; many testbeds require users to have a level of control that is simply not possible in a virtualized cloud environment [81]. Thus, an attractive way to share a cluster between multiple types of workloads is to share it rather than to stack them; that is, to have a single hardware pool, where individual compute nodes are, at different times, controlled by a cloud management framework, an HPC scheduler, a testbed manager, etc.

In our design, the physical cluster is divided into a number of partitions: each physical resource is assigned to one partition at a time. Each partition is associated with a management framework (such as OpenStack, SLURM, HTCondor, etc.) that is responsible for scheduling jobs on the resources that are assigned to its partition. Each management framework manages its own policies (e.g., job priorities, resource limits, preemption, autoscaling, etc.). We also refer generically to the environment provided by a particular framework; e.g., SLURM is said to provide

Figure 3.1: Multiple environments sharing the same cluster.



an "HPC environment". Figure 3.1 illustrates the logical relationships between environments and management frameworks on a cluster with two partitions. Since the fundamental goal of this system is to dynamically assign resources to partitions based on time-varying demands, there must be policies regarding how to handle elasticity between the partitions: for example, to decide when resources should be moved from the cloud partition managed by OpenStack to the HPC partition managed by SLURM. Each management framework has its own complex methods for determining the "value" of a particular resource at a particular time. For example, SLURM may know that a particular node has been running code that is part of a long, non-preemptable job for a long time, and much work would be wasted if the node was removed from its control (thereby killing the job). On the other hand, a cloud may know that a particular server is not currently hosting any VMs, or the ones that is is hosting could be easily migrated, and thus there would be very little "cost" to removing it from the cloud's partition. The values that frameworks place on the nodes under their control will change over time, as different jobs start, complete, change priority, etc. Thus, it is critical that the high-level scheduler have enough visibility into the relative values of each resource to make decisions that help each framework optimize its own resource scheduling, without having to understand the full internal details of how each computes that value. Our hypothesis is that we can establish simple, general-purpose communication between each computing framework and the overall cluster management process, and use it to avoid wasteful behavior on the cluster as a whole.

We have built a prototype of this system (described in Section 3.3), running on one of our

clusters and available to selected users, to share resources between users of a network testbed and an HPC cluster. In Section 3.2, we discuss the related work in elastically scaling workloads within one compute environment, which contrasts with our work on elasticity of resources between partitions supporting diverse computing environments. In Section 3.4, we use a simulator with real workload traces to look at different high-level scheduling policies, and show that it is possible for an HPC framework to optimize for metrics such as minimal wasted cycles and to implement its own priority policies. We finish up with conclusions in Section 3.5.

3.2 Background and Related Work

There is much work in the literature, e.g., [32] and [96], that looks at dynamic scaling of individual applications. This is commonly referred to as elasticity. As described in [50], horizontal elasticity allows applications to "grow" or "shrink" the pools of their computing resources by adding or removing physical or virtual machines. This is in contrast to vertical elasticity, where physical resources are added to or removed from a single virtual machine. In this section, we survey some of this related work.

3.2.1 Targeted Computing Environments

In this work, we target modern research cyberinfrastructure environments. These take many different forms, including supercomputers [46], clusters [27], grids [48], and clouds [18]. Each of these classes of systems can be characterized by unique load characteristics, resource allocation models, software stacks, administrative practices, etc. Another class of cyberinfrastructure, *testbeds*, which serve the research community by providing access to special-purpose resources and controlled environments, blurs the traditional class distinctions. They operate much like clouds, allowing users to provision groups of resources with software stacks. Unlike clouds, testbeds may provide a variety of non-uniform resource types, advanced research-driven environment pre-configuration, and controlled environments that foster reproducible research. Testbed users may deploy a wide variety of software on the same testbed, resulting in a diverse set of co-existing workloads. The

NSF-funded testbeds that have been established in the last several years include CloudLab [81], GENI [22, 73], Apt [83], Jetstream [94] and Chameleon [95].

3.2.2 Backfilll

In [67], the authors describe a mechanism called *backfill* for increasing the utilization of clouds via opportunistic provisioning of cloud resources for HTC workloads. Our work on elastic partitions incorporates the following distinctions. First, in contrast with their model with a single backfill per environment, we allow multiple elastic partitions to co-exist on the same system; a system-wide manager service needs to communicate with them and ensure fairness. Second, while they only consider abrupt terminations of the resources provisioned with the backfill, we propose a delayed, graceful preemption where the resources being preempted get a chance to complete their jobs within a specified time period. Third, comparing to the backfilled virtual machines in multiple clouds, our experiments consist of co-located physical machines that are more suitable for running HPC workloads.

3.2.3 Cloud Bursting

Cloud-bursting techniques for augmenting clusters with additional computing resources running in clouds are described in many studies, including [63], [72], and [66]. Much attention in these studies is paid to the analysis of how quickly the resources can be provisioned and configured at moments of peak utilization. In contrast, we consider upscaling scenarios to be thoroughly investigated and focus on downscaling, and supporting management frameworks and workloads for which bursting is non-optimal. Specifically, we investigate involuntary downscaling scenarios where the partition manager "steals" resources from partitions in response to greater demand from other partitions. This downscaling is unusual on commercial clouds because of their enormous sizes and the guarantees given to their paying customers. However, on academic clusters, testbeds, and clouds with limited capacity, steady-state behavior tends to be that all resources are near full utilization. In this case, the problem becomes moving resources between compute environments in such a way

as to cause minimal disruption to the jobs managed by their associated management frameworks.

3.2.4 HPC Resource Management

In the large body of research on resource management strategies for HPC systems, the most relevant concepts are investigated in the following studies. In [62], the authors measure quantities that reflect the *Value of Service* (VoS) for each computing resource—an individual virtual machine in their environment. They consider such factors as completing a job within the requested wall time and staying under the specified energy consumption limit (based on additional information that user provides to the scheduler at job submission); if these conditions are violated, then programmable penalties are applied and VoS values are reduced. An importance factor that specifies the relative significance among tasks is also considered. Similarly, [60] describes penalty functions that are applied in situations where computing jobs are discarded or aborted. In our work, we do not optimize scheduling algorithms in the environments with such penalties, but rather use a common scheduling algorithm to investigate tradeoffs between scheduling-related preemption policies that influence the penalties. We will also return to the idea of penalizing jobs that run longer than expected in our discussion of future work.

3.3 Enabling Elastic Partitions

We now describe our system for sharing a single hardware cluster among multiple partitions, each providing a different compute environment. Our system has two major components that operate as follows. First, a *hosting cluster* is responsible for assigning individual compute resources to specific partitions. The software managing the hosting cluster must be capable of bare-metal provisioning, so that it can boot nodes into the hypervisor used by cloud hosts, the software build used by an HPC cluster, etc. Second, *elastic partitions (EP)* grow and shrink on demand, as they obtain and release resources from the hosting cluster. While we do not place any restrictions on the management framework that runs an EP, in this work we focus on frameworks where a job scheduler, compilers, and libraries are installed to allow users to run computational jobs of their

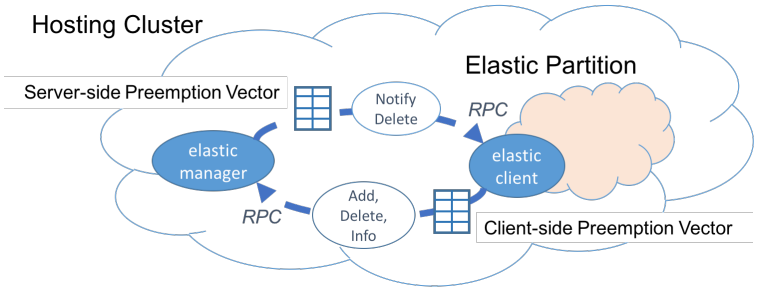


Figure 3.2: Experiment's elasticity facilitated by the exchange of preemption vectors.

choice. Users interacting with the EP see only the user interface of the framework that runs within it; for example, the SLURM job queues or the OpenStack Horizon web interface.

EP elasticity is implemented through the following three operations. First, when an EP experiences high utilization, it should acquire additional resources from the hosting cluster to increase its workload throughput; we refer to this as *partition-driven upscaling*. Second, EPs may also react to drops in utilization with *partition-driven downscaling*: being a conscious tenant of the hosting cluster implies releasing unused resources in a proactive and timely manner. The hosting cluster will reclaim these resources and make them available to other partitions. Third, *host-driven downscaling* allows the hosting cluster or its administrators to re-balance resources between partitions, by forcibly reclaiming resources from one EP and allowing another EP to request them. Specifically, we refer to the desired scenario as *graceful preemption*: the hosting cluster contacts an EP and gives it a fixed period of time, the *Grace Period* (GP), to free up P out of N EP's resources. After GP is exceed, P resources will be withdrawn from the EP without further notice.

These upscaling and downscaling operations must be defined such that they intelligently select which particular machines are shifted between partitions. For instance, in an upscaling scenario, an EP may request machines of a specific type (e.g., type of process or quantity of RAM), or with a necessary feature (GPU, high-performance interconnect, etc.). In contrast, in host-driven downscaling, the magnitude of the negative impact caused by preemptions may depend on which particular machines are preempted in the EP. Therefore, it is advantageous to allow EP to periodically report back to the hosting cluster how valuable its resources are to its applications.

Given this information, the hosting cluster can make informed decisions and preempt the least-valuable resources. In the following sections, we describe how EPs report resource values to facilitate elasticity.

3.3.1 Architecture

Figure 3.2 depicts the relationship between the hosting cluster and EPs, including the types of interactions between them. Each partition is equipped with a controller: the *elastic manager* runs in conjunction with the hosting cluster, and one *elastic client* runs in conjunction with each EP. They interact via Remote Procedure Calls (RPC) and frequently exchange resource values in the form of *Preemption Vectors* (PVs). These vectors include (*resource ID*, *value*) pairs for all resources under each partition's control. (We limit the range of these values to floats in the interval $[0, 1]$, where 1.0 is most-valued, and 0.0 is least, without sacrificing the expressive power of the interface.) PVs reported by the hosting cluster reflect the probability with which EPs can acquire specific resources; in the simplest case, 0.0 can represent available resources (those not currently assigned to any EP), and 1.0 can be used for resources in use by another EP. Responsibility for providing client-side PVs falls to the framework managing each EP, and the mechanism for calculating them depends on the goals of compute environment provided by that framework, and on framework-specific knowledge of the jobs that are currently running. For example, a testbed, which simply views resources as "allocated" or "unallocated" might use the same binary distinction described for the hosting cluster. As we will see in the sections below, frameworks that host long-running compute jobs may use values that reflect how many cycles would be "wasted" if a particular job is killed. Other frameworks may use their own notions of job priority, preemptability, etc. to decide which jobs—and therefore, nodes— are the most valuable.

Elastic clients use RPC or a GUI interface to create a partition, and later invoke the `AddNodes` and `DeleteNodes` RPCs to add or remove one or more nodes to or from their partition. They can call the `DiscoverResources` RPC at any time to list the hosting cluster's resources and metadata, such as availability. Clients send updated preemption vectors to the elastic manager at the hosting

Table 3.1: Parameters of the selected workloads.

Workload	# of Jobs	Total node-hours	Runtime, seconds						Node Count					
			mean	min	25%	50%	75%	max	mean	min	25%	50%	75%	max
HTC	11200	1379.05	443.27	54.00	84.00	106.00	134.25	7140.00	1.0	1.0	1.0	1.0	1.0	1.0
PEREGRINE	7275	26838.58	7119.07	33.00	439.50	1921.00	6840.5	85945.00	1.42	1.00	1.00	1.00	1.00	16.00

cluster via `SetResourceValues`.

Elastic clients also host an RPC endpoint through which the hosting cluster's elastic manager notifies them of its own resource values, and resource preemption. The elastic manager invokes the client's `NotifyDeletePending` method to inform the client that one or more of its nodes are being reclaimed by the hosting cluster. This method is invoked every minute for a pending node reclamation until either the client informs the manager that it has finished with the node, or until the grace period has expired for node.

3.3.2 Implementation

Our implementation of this architecture uses the Emulab cluster-management software [98, 45] in the role of the hosting cluster. The API calls for exchanging PVs and moving nodes between partitions are implemented as an extension to the ProtoGENI [59] API, one of the RPC interfaces supported by the Emulab software, and partitions are modeled using the "slice" abstraction that this API inherits from the GENI [22] design. We implemented an elastic manager that supports multiple elastic clients, and are running it in a restricted-use mode on the Apt cluster of the CloudLab testbed [34], where it is used to share that cluster between testbed and HPC users.

We also implemented two elastic clients: a general-purpose, reference implementation, and one that interacts with a SLURM job scheduler in order to create PV values for the nodes it manages. The SLURM client uses Chef for configuration management of its nodes, and the Chef cookbooks developed in [43]. While most management frameworks are not designed to have nodes dynamically added and removed from them, we note that most do have mechanisms for disabling nodes for maintenance, power savings, etc. It is the latter mechanism that our implementation

hooks into in SLURM: we give SLURM a view of the entire hosting cluster, but tell it that all nodes not belonging to its partition are currently "off". In order to downscale, we tell SLURM that the effected nodes are to be turned off for maintenance, causing it to drain jobs from them. To upscale, we tell SLURM that the affected nodes have been turned back on, which makes them available for scheduling and running new jobs from the work queue.

3.3.3 Policies

In our elastic client for SLURM, we developed five policies that report client-side PVs as follows. The RANDOM policy randomly assigns preemption values. We implement this policy to evaluate the base case in which the hosting cluster has no knowledge of how much the EPs value their nodes, and therefore selects arbitrarily for downscaling. Under the FIFO (First-In, First-Out) policy, the longer a node has been running its current job, the lower the value that gets assigned, and therefore the more likely it is to be reclaimed during host-driven downscaling. Under the LIFO (Last-In, First-Out) policy, the longer a node has been running its current job, the higher the value it gets, making it less likely to be reclaimed. For workloads with multi-node jobs, the PAP (Parallel-Aware) policy takes into account the number of nodes used by each job; preemption values are calculated as products of job runtimes and node counts scaled to [0,1]. This makes nodes running smaller jobs more likely to be reclaimed. PAP is identical to LIFO for single-node workloads. Finally, the PAP+ policy is an extension of PAP that weights jobs by their priority; this policy prefers to put high PV values on nodes that are currently running high-priority jobs.

We use RANDOM as the baseline value, since without our PV exchange protocol, the hosting cluster would have no way to know what jobs are running on the each partition's nodes and would have no basis for making efficient or fair decisions about which nodes to reclaim during host-driven downscaling. By comparing the other policies to RANDOM, we can quantify the effectiveness of exchanging information between the hosting cluster and EPs.

3.4 Evaluation

To study the effectiveness and efficiency with which the hosting cluster and EPs cooperate to share compute nodes, we developed a discrete event simulator for simulating job execution. The simulator approximates SLURM's *sched/backfill* policy by using the First-Come-First-Served job scheduling with backfilling. With subtle differences in job ordering (which we consider insignificant in this evaluation), the simulations closely resemble real-time execution experiments performed on the prototype. To draw reliable conclusions about aggregate characteristics, we simulate the behavior of EPs which stay occupied with processing jobs from realistic scientific workloads for long periods of time. Lengthy time periods are necessary to illustrate the behavior of developed policies in a variety of situations corresponding to many mixes of simultaneously running jobs with diverse runtimes and node counts. The simulator helps us understand the behavior of the policies in multiple system configurations, as well as evaluate fairly large EPs, without requiring the intractable usage of the hosting cluster's resources which would be otherwise needed for real-time experiments with the selected workloads.

Each run of the simulator takes as input a set of compute jobs and treats it as a single batch submitted at the beginning of the run. It simulates job execution until all jobs complete and records the state of execution every time one of the jobs completes. While advancing through the simulated time between these moments, the simulator also records the state every T seconds; in this evaluation, we use $T = 30$. Each of these recorded states is passed to the functions that return PVs for different policies. Using these PVs along with the information about job runtimes and used node counts, we compare the proposed policies and their variations based on the negative effects caused by preemptions, as described in detail in Section 3.4.1. In this chapter, we focus on the host-driven downscaling scenarios with $N = 20$ (EP node count) and $P = 10$ (EP nodes preempted) and briefly mention several simulation results for larger EPs.

In our analysis, we do not explicitly model sequences of preemptions followed by reductions in the EP's capacity and job throughput. Without access to real preemption event logs that are

required for such modeling, we would need to make assumptions about how often and when with respect to the beginning of batch processing the host-driven preemptions are likely to occur. We choose an alternative approach and focus on the analysis of a single preemption where we accept the fact that it can happen at any time. In other words, we treat all aforementioned execution states as the moments at which the preemption has the same probability of occurrence. By capturing and processing as many such execution states as possible, we aim to characterize distributions of the policy efficiency metrics and, for instance, understand their ranges, the means, and the median values. Thus, we compare the proposed policies based on their behavior during tens and hundreds of thousands of simulated independent preemptions.

We use two traces of scientific computing workloads to drive the simulator. We obtain and process these traces as follows. Our HTC workload consists of short- and long-running single-node job traces obtained from HTCondor Log Analyzer [7]; we previously studied this trace in [42]. To prolong the simulated execution and observe more variability in the policy behavior, we create the HTC workload with over 11K jobs by concatenating 10 shuffled copies of the original trace from our study in [42]. The PEREGRINE dataset is a large volume of accurate job information from the HPC energy efficiency research at the National Renewable Energy Laboratory (NREL), available at [80]. We parse the dataset with 10K jobs randomly selected from two years of execution on the NREL's flagship supercomputer called Peregrine and select a subset of jobs based on the following criteria: node count ≤ 20 (otherwise, jobs will not run on the simulated EP), runtime ≤ 24 hours (longer jobs may be viewed as bad candidates for running in shrinking environments), successfully finished and have no missing fields. These criteria yield 7,275 jobs from 24 applications and 7 job queues.

Table 3.1 provides the detailed statistics for the jobs in these workloads. The real-time execution of PEREGRINE would require over 55 days on a 20-node HPC environment, whereas our simulator completes the processing in several hours. Our Jupyter notebooks with complete simulation, post-processing code, and all results are made available at [38].

3.4.1 Counting Wasted Cycles

We use wasted cycles as our primary efficiency metric. When a job is terminated before completion, the cycles spent so far on it are wasted, and the computation must be performed again when the job is restarted. Note that for parallel HPC jobs, if any node on which the job is running is preempted, the entire job must be restarted, so all cycles on all nodes running the job are considered wasted.

The simulator treats the $P = 10$ out of $N = 20$ nodes with the lowest values in PVs as the sets of nodes recommended for preemption by the elastic client. It counts wasted cycles (WC) for each of the jobs j from the set of jobs J_p that currently run on the preempted nodes as:

$\mathrm{WC}_j = 0$, if $j.remaining\_time < \mathrm{GP}$,

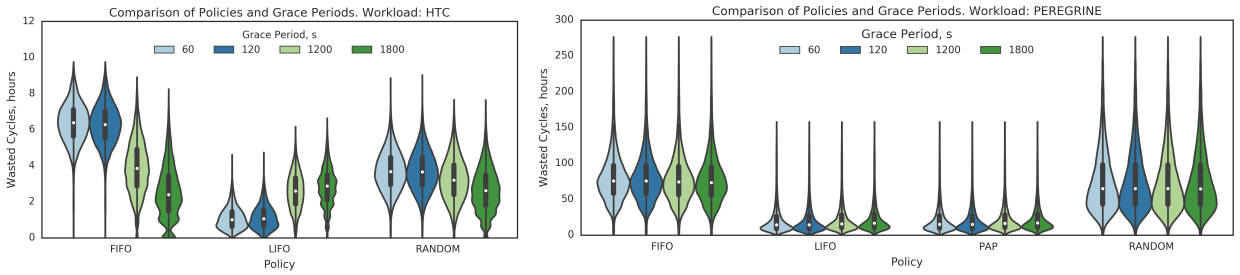
$\mathrm{WC}_j = (j.elapsed\_time + \mathrm{GP}) * j.node\_count$, otherwise.

The former rule corresponds to the cases where the grace period (GP) gives jobs sufficient time to complete. In the latter cases, all cycles spent on these jobs so far and also the future cycles within GP are wasted because the jobs are unable to complete on time. In theory, we could also count the cycles spent in the former case between the job completion and the end of GP, and treat such idle cycles as another evaluation metric. However, our prototype's elastic client terminates the nodes that are marked as draining (i.e. running their last allowed job before preemption) immediately after they become idle. Reducing the waste across the EP in this notation can be formulated as: $\mathrm{WC} = \sum_{j \in J_p} \mathrm{WC}_j \rightarrow \min$.

3.4.2 Varying Preemption Grace Period

In engineering the waste-minimizing host-driven downscaling, we must carefully select the preemption grace period GP, the parameter that directly affects WC. It is intuitive to believe that the larger GPs yield lower WC values as they allow more jobs to complete. However, as we describe in the following section, this statement is not statistically accurate. We choose specific GPs for our simulations as follows: we start with GP=120s, which corresponds to the notification period for

Figure 3.3: Distributions of Wasted Cycles for the developed policies and selected Grace Periods.



(a) HTC: single-node workload.

(b) PEREGRINE: multi-node workload.

preemption of spot instances on the AWS cloud (referenced in [99]). We also consider GP=60s as an example of a hosting cluster with fast preemptions. In addition to the larger values GP=1200s (20m) and GP=1800s (30m), which can still be considered "responsive", we include in our analysis a range of less responsive values between 1 and 6 hours. This responsiveness relates to the fact that its interactive users need to wait for the requested resources to become available when all resources are utilized by EPs: the larger the GP, the longer the maximum waiting time. Finding efficient combinations of GPs that are not prohibitively large and the policies that minimize WC is one of the primary goals of our evaluation and the subject of the following section.

3.4.3 Quantifying Tradeoffs

For each of the selected policy-GP combinations, our simulations yield over 14K samples for HTC and 168K samples for PEREGRINE. The distributions of the WC values that correspond to these samples are depicted in Figure 3.3. These violin plots characterize the probability of the observed WC values: the width of the violins represents the relative frequency of the corresponding y-axis values in the collected samples, the thick vertical lines depict interquartile ranges (IQRs), and white dots show median values (for heavily skewed data, such as the shown distributions, the median is viewed as a more representative measure of central tendency than the mean). Based on these distributions, we draw the following conclusions.

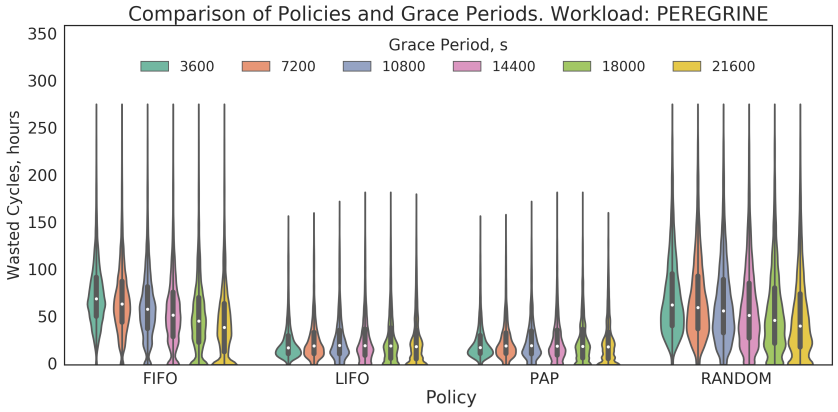


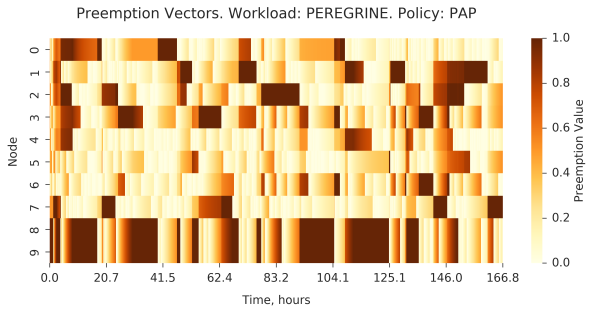
Figure 3.4: Distributions of Wasted Cycles for the developed policies coupled with Grace Periods between 1 and 6 hours.

For HTC, LIFO provides significant reductions in WC compared to FIFO and RANDOM. LIFO's GP=60s and GP=120s statistically perform better than the larger GPs. This can be explained by the fact that the long-running jobs will often incur penalties from premature terminations (second rule for WC) shortly after they start running, and the advantage of increasing the probability of job completion does not outweigh these penalties for LIFO.

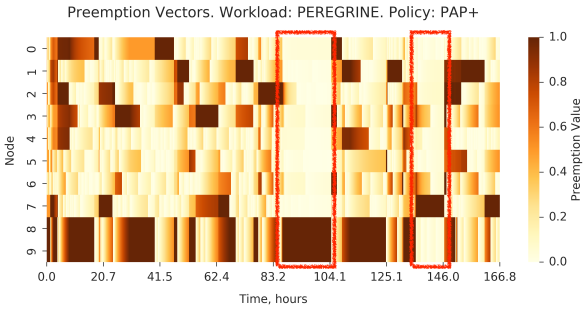
For PEREGRINE, LIFO and PAP demonstrate similar results, although they are significantly more efficient than FIFO and RANDOM. In fact, in more than 75% of the samples, the nodes preempted by LIFO and PAP are the same (therefore, WC is identical in these cases). This is due to the fact that the node rankings based on $j.elapsed\_time$ and the product $j.elapsed\_time * j.node\_count$ are similar for PEREGRINE where over 82% of the jobs use a single node. We obtained similar results in simulations of much larger environments with $N = 200$ and $P = 100$. Since PEREGRINE is dominated by the long-running jobs, GP is a factor with negligible impact on WC, at least for GP $< 30m$.

In Figure 3.4, we show the impact of much larger GPs, between 1 and 6 hours, on WC for PEREGRINE simulations. As we can see, the median values for WC with FIFO and RANDOM monotonically decrease with the growth of GP because an increasing number of jobs can complete execution; IQRs slide towards zero. At the same time, the WC median values for LIFO and PAP

Figure 3.5: Heatmaps that visualize Preemption Vectors for a 10-node elastic partition running the PEREGRINE workload. The darkest areas represent the nodes that are most valuable. Only a small fraction, under 15%, of the entire simulation is shown.



(a) PAP policy.



(b) PAP+ policy. Red rectangles highlight the moments when the running high-priority jobs reduce the relative importance of the running default jobs.

remain constant, and IQRs do not change much. It appears that the growth of the frequencies of the lower, near zero WC values is counterbalanced by the increased frequencies of the larger values. In the extreme case where GP is so large that it exceeds the majority of job runtimes, WC will approach zero; however, such GPs are unresponsive and impractical from the testbed perspective. We conclude that the large but practical GP values up to 6 hours do not provide noticeable advantages in terms of the WC's median and IQR for workloads similar to PEREGRINE.

3.4.4 Improving Parallel-Aware Policy

We can enable practical optimizations in PAP if we consider relative significance among jobs. For instance, two jobs that use the same number of nodes and start roughly around the same time may have drastically different values that characterize their importance: small for cycle-scavenging, low-priority computations and much larger for applications with strict deadlines. Similarly, the relative importance can be expressed using job queues. In PEREGRINE, the dataset suggests that the NREL's system has 7 job queues. We can view *bigmem* or *large* queue as more important than *debug* and help jobs in these queues to have fewer terminations.

We developed the PAP+ policy that considers two priority classes, but the proposed PV

exchange with the [0,1] values can support an arbitrary number of classes. To evaluate the potential tradeoffs, we choose 76 jobs in the *bigmem* queue in PEREGRINE (∼1% of the total number) to be the highly valuable subset with $j.priority = 10.0$; for the rest of the jobs, we assume the default class: $j.priority = 1.0$. The elastic client calculates PVs as:

$$\text{PV} = \text{scale}(j.elapsed\_time * j.node\_count * j.priority),$$

where scale() divides vector components by their maximum value (in contrast, PAP treats the last factor as 1.0 for all jobs).

Figure 3.5 shows the difference between PAP and PAP+ using heatmaps for the PVs they return during a simulation of a fraction of the PEREGRINE workload. We can clearly see several intervals of time (marked with red rectangles) when the high-priority jobs run. These jobs gain importance after they start running much faster than other jobs and also reduce the relative importance of the rest of the simultaneously running jobs (i.e. the rest of the red rectangles appear mostly white, indicating little relative value).

With the same procedure for estimating WC, we compare PAP and PAP+ based on the cumulative amount of WC. In Figure 3.6, we can see how the cumulative WC values change throughout the simulated executions and compare these values at the end of PEREGRINE simulations. PAP+ reduces cumulative WC for high-priority jobs by 82.9% comparing to PAP. This happens at the expense of the default-priority jobs: PAP+ increases cumulative WC for default-priority jobs by 14.3% comparing to PAP. The vertical axes on the shown plots use different scales, indicating a large contrast between the corresponding absolute differences. However, the 7.5% growth of the combined cumulative WC is well justified by the significant reduction of the negative impact on the high-priority jobs. In the similar manner, PAP+ can focus on preserving jobs that represent a particular application. In our experiments where we assign $j.priority = 10.0$ to jobs with $j.application =$ gaussian (341 jobs, $\sim$ 5% of total), we observe the 97.2% reduction in the cumulative WC for high-priority jobs provided by the 7.1% increase for default-priority jobs (only 3.8% growth in total).

3.4.5 Discussion

Below, we comment on the limitations of our analysis and the nuances which we must consider when moving our prototype into production.

Using a more advanced scheduler simulator such as the SLURM simulator described in [61] can enable experimentation with schedules that better represent production HPC systems with fairness and quality-of-service optimizations. However, the referenced system takes as input continuous SLURM event logs, and running it on the sampled, anonymized, and formatted records published at [80] is infeasible. To experiment with the SLURM simulator, we will need to find alternative sources of HPC traces. Alternatively, we can switch from simulations back to real-time execution experiments on the developed prototype. It will never be practical to run 55-day experiments on our shared testbed resources, but we can gain valuable experience with the optimized scheduling algorithms in SLURM using modestly long experiments.

Do large Grace Periods provide any advantage? In our evaluation, they "freeze" preempted nodes for long periods of time without reducing the amount of wasted computations for the most efficient policies. If this conclusion holds for other workloads – candidates for running on elastic partitions, we may consider reducing GP to its minimum value, on the order of several seconds. Thus, SLURM will receive the time that is only enough to kill the currently running jobs and mark the preempted nodes as unavailable. Such preemptions will still proceed gracefully, and, from the testbed perspective, will eliminate the need to maintain a pool of idle resources unavailable to elastic partitions. On the contrary, with larger GPs, if their use is justified, the testbed needs to have such pool to quickly respond to requests from its interactive users. This idle pool is likely to reduce the overall utilization of the testbed.

Another practical concern relates to stale PVs, i.e. PVs that represent states of execution at moments in the past that are far from the current time. Making preemption decisions based on such PVs is likely to cause non-optimal terminations. Stale PVs may adequately represent long-running jobs, but for the jobs that have recently started running they will include faulty, unreliable

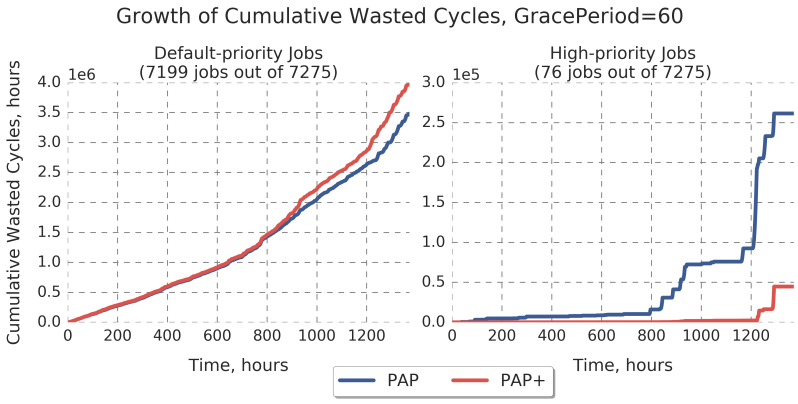


Figure 3.6: Using cumulative Wasted Cycles to compare PAP and PAP+ that prioritizes jobs from the specified queue. At the expense of a small growth for default-priority jobs, PAP+ significantly reduces Wasted Cycles for high-priority jobs.

values. The longer the delay between the moments when PVs are reported and the moments when preemption actions are triggered, the faster LIFO, PAP, and PAP+ degrade to the RANDOM policy. In this degradation, the gains provided by these policies are diminished. Therefore, the hosting cluster needs to ensure that it indeed uses the recently received node value information; otherwise, preemptions should be delayed or aborted.

3.5 Summary

In this chapter, we have described our investigation of interactions between portions of a shared cyberinfrastructure managed by diverse computing frameworks, and presented a novel infrastructure for resource sharing and optimization of resource usage. In our analysis, we use an elastic prototype—a SLURM-managed elastic cluster—deployed on the CloudLab testbed and also a discrete event simulator we developed for simulating workload traces from real HPC systems. By simulating a trace from the Peregrine supercomputer at NREL, we obtained important insights into how these elastic compute clusters operate under realistic HPC workloads with long-tailed distributions for runtimes and numbers of compute nodes. In balancing their batch and interactive workloads, we rely on a minimal information flow based on the exchange of preemption vectors that abstractly represent environment-specific resource values. Using wasted cycles as a metric, we

demonstrated that, with this minimal information sharing, we are able to give each environment the ability to optimize for its own goals. We also found that, somewhat counter-intuitively, shorter grace periods when downscaling lead to fewer wasted cycles, as most jobs do not finish within the grace period anyway.

Chapter 4

Architecting a Persistent and Reliable Configuration Management System

Streamlined configuration management plays a significant role in modern, complex distributed systems. Via mechanisms that promote consistency, repeatability, and transparency, configuration management systems (CMSes) address complexity and aim to increase the efficiency of administrative procedures, including deployment and failure recovery scenarios. Considering the importance of minimizing disruptions in these systems, we design an architecture that increases persistency and reliability of infrastructure management. We present our architecture in the context of hybrid, cluster-cloud environments and describe our highly available implementation that builds upon the open source CMS called Chef and infrastructure-as-a-service cloud resources from Amazon Web Services. We demonstrate how we enabled a smooth transition from the pre-existing single-server configuration to the proposed highly available management system. We summarize our experience with managing a 20-node Linux cluster using this implementation. Our analysis of utilization and cost of necessary cloud resources indicates that the designed system is a low-cost alternative to acquiring additional physical hardware for hardening cluster management. We also highlight the prototypes security and manageability features that are suitable for larger, production-ready deployments.

4.1 Background

Computing clusters are collections of computers, so-called nodes, which collaboratively work on computational problems. In addition to compute nodes, clusters typically have service nodes,

including admin, front-end, gateway, and storage nodes. Each type of service node has a unique software stack and functionality. During the cluster deployment process, individual nodes are provisioned, their software stacks are installed and configured, and inter-node connections are established. On small clusters necessary administrative operations can be performed manually, step-by-step. In contrast, on large clusters such as, e.g., systems in the Top500 list (with 2K to 16K nodes), automation of deployment and management procedures is necessary.

High frequency of failures is another characteristic of large-scale systems. For example, in [88], the authors reveal that on average a 4096-node system experiences 3 failures per day. They show that the failure rate does not grow significantly faster than linearly with system size. In a recent study [100], the authors analyze the record of failures on the K computer (the world's fastest supercomputer in June 2011), and indicate that the 82,944-node system on a monthly basis experienced a maximum of 25 hardware and 70 software failures in the first year of operation. The study highlights that improvements in the software stack (specifically, related to the file system and the job scheduler) can help significantly reduce the rate of software failures.

An alternative to investing effort into hardening software stacks is to employ management systems that provide functionality for responding rapidly to failures and system transformations. Considering the accelerated pace of the modern software development, investigating management systems that are compatible with a variety of software stacks is more promising than researching improvements tied to particular software. In failure scenarios, the necessity to perform recovery operations as quickly as possible dictates a strong need for a high degree of automation in the management systems.

A class of system software called configuration management systems (CMSes) is designed to satisfy this need. CMSes introduce consistency, repeatability and transparency into infrastructure management. The key principle behind these systems is the "infrastructure as code" approach, according to which every administrative procedure should be implemented as code. These systems can manage any layer of the software stack, from the operating system to application-specific libraries and packages, and can orchestrate various types of resources, including virtualized and bare-metal

servers, both local and remote. In the context of increasingly common hybrid computing systems, where cloud resources augment and work in collaboration with on-premise resources, CMSes are particularly useful for managing dynamic cloud resources that are launched and terminated based on demand.

A CMS running on an administrative node holds significant management power over the rest of nodes in the environment where it is employed. The environment's system administrators rely on its operability to perform administrative operations, possibly with stringent deadlines. Therefore, the admin node or nodes responsible for the CMS need to be protected from disruptions similar to other system critical services. To the best of our knowledge, existing off-the-shelf CMSes are not equipped with fault tolerance features that meet this standard. When deployed on commodity hardware, they are as exposed to hardware failures as the rest of the environment.

In this chapter, we describe our work on architecting a cluster management system with increased persistency and reliability. We propose leveraging infrastructure-as-a-service clouds as a source of on-demand virtual machines and storage volumes. We augment a local commodity cluster with those remote resources to build a geographically distributed, highly available (HA) configuration based on a general-purpose, off-the-shelf CMS. We present our implementation that extends the open-source Chef CMS and leverages Amazon Web Services (AWS) virtual machines and storage resources. In the implementation's evaluation, we examine security and manageability aspects, as well as investigate utilization of the cloud resources and incurred aggregate costs. We summarize our experience with managing a 20-node group's Linux cluster using the developed prototype and demonstrate the type of analysis that can be applied to larger installations.

The chapter is structured as follows. Section 2 discusses the motivation for this study. Section 3 describes possible approaches to architecting a CMS with cloud components. Sections 4 and 5 present our architecture and implementation of the HA-enabled CMS. In section 6 we summarize our evaluation of the prototype implementation. Section 7 discusses related work. We conclude and propose directions for future work in section 8.

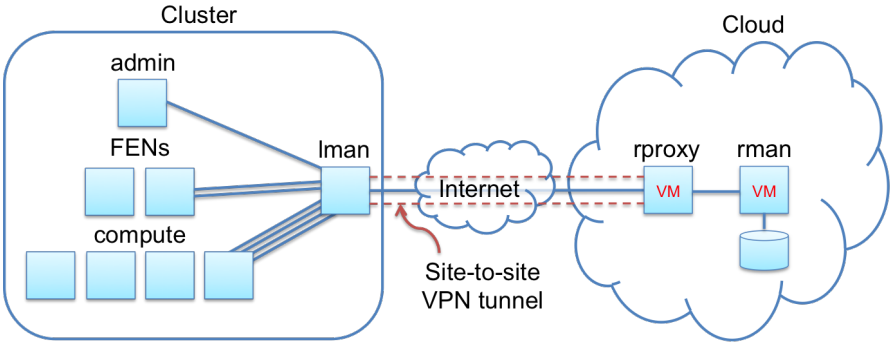
4.2 Motivation

In this study we focus on hybrid systems that comprise computing environments that are deployed across local and remote computing infrastructures. Hybrid systems with cluster and cloud components have been researched in many recent studies, including our group's previous work (e.g., [70],[69]) where we focused on scalability and cost optimization aspects. In this chapter we propose using the cloud not only for additional compute cycles (as in studies of outsourcing of high-throughput computing workloads) but also as a platform for hosting administrative services. Focusing on the system administration domain, we design a system that augments a private configuration management system (CMS) with virtualized resources running in a public cloud in order to increase persistency and reliability of automated infrastructure management. We leverage infrastructure-as-a-service (IaaS) cloud resources on two main premises: (1) those resources are highly available (e.g., AWS guarantees the unprecedented availability of 99.95% for its compute and storage offerings), and (2) operators of hybrid environments are already familiar with launching and managing cloud resources. With the help of a small set of additional dedicated cloud resources, we pursue establishing a CMS configuration that exhibits persistency and reliability. The former means that the system is capable of persisting significant architectural transformations (i.e., remains operational when underlying components are decommissioned or added), while the latter requires an ability to persist component failures. We design a system that transparently extends functionality of an existing CMS, without altering its user interface, and can satisfy security and manageability requirements of production environments.

4.3 Approach

IaaS clouds provide an ability to provision resources and build configurations that satisfy diverse customer needs. In our case, we augment local, on-premise resources with on-demand, customizable and highly available cloud resources in order to increase persistency and reliability of a general-purpose, off-the-shelf CMS. We consider three approaches to architecting this hybrid

Figure 4.1: The highly available cloud-based cluster management system.



environment: (1) Naive approach: the CMS runs locally and copies its state and data into reliable cloud storage, (2) Outsourcing: the CMS runs in the cloud and uses reliable cloud storage at the back-end, and (3) High-Availability (HA): the CMS runs in a highly available configuration where its core services run both locally and in the cloud, and their states and data are continuously synchronized; on the cloud side, the system still uses the storage offering with the desired level of reliability. The risk associated with a failure of a critical component is not addressed in either of the first two approaches. In the second approach, the CMS operability also inevitably depends on a stable connection between the local system and the cloud infrastructure. The HA approach addresses these issues and introduces a system where both local and cloud environments are sufficiently independent and can sustain individual failures or losses of connectivity. In this study, we adhere to the HA approach and architect a highly available CMS for managing cluster-cloud environments.

4.4 Highly Available Architecture

The proposed architecture is depicted in Figure 4.1. It consists of three high-level components: (1) a local cluster formed by interconnected nodes with specific roles, (2) a remote cloud environment with nodes and storage volumes, and (3) a link between the two environments tunneled over the Internet.

In this architecture, a typical computing cluster that consists of the admin, front-end and compute nodes is augmented with the lman (local manager) node that is dedicated to providing connectivity with the cloud and hosting the instance A of the CMS head node. On the cloud side, the rproxy (remote proxy) VM acts as a gateway for connecting to the cluster, and the rman (remote manager) VM hosts the instance B of the CMS head node. The storage volume attached to rman is used for storing CMS state and data. The instances A and B comprise an HA pair of CMS head nodes. We configure this pair as an active/passive failover pair where lman (A) is the active node and rman (B) is the passive one (which becomes the new active if A fails). Seamless integration between the two environments is achieved using a site-to-site virtual private network (VPN) system deployed on lman and rproxy. The established VPN tunnel eliminates environment boundaries and makes nodes on one side accessible from the other.

4.5 Implementation

To develop a prototype, we leverage a number of existing technologies. Specifically, we use:

- Open source Chef 11 [5], the CMS we already use for managing the group's local cluster. The proposed architecture is general enough to be implemented with other CMSes (e.g., Puppet, SaltStack, Ansible, etc).

- Amazon Web Services resources: VMs in Elastic Compute Cloud (EC2) and storage volumes in Elastic Block Store (EBS). The designed prototype can be deployed on resources from other cloud providers.

- Openswan 2.6.32 [6], an open source implementation of the IPsec-based VPN for Linux.

- Distributed Replicated Block Device (DRBD) [7], the open source kernel driver and user-space utilities for storage mirroring in HA clusters. DRBD is installed on lman and rman. The use of DRBD in the prototype is explained in detail in section 6.2.

- RedHat 6.6 (Santiago) across all cluster and cloud nodes. Selected software is distributed

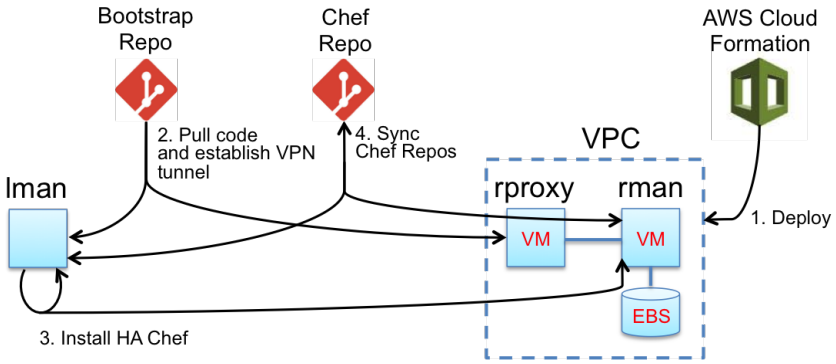
as RPM- and Debian-based packages (source code is also available), which allows deploying the prototype on a variety of Linux distributions.

Integration of the aforementioned distributed technologies presents a significant management challenge when individual components are managed manually. We streamline the process by leveraging AWS CloudFormation [8]. This service allows describing cloud resources "as code" in the form of CloudFormation templates. We developed a template that defines an AWS Virtual Private Cloud (VPC) incorporating rproxy and rman with an attached EBS volume. This sharable template facilitates consistent deployment of the prototype's cloud components from the AWS web console with minimal effort. We supplement the CloudFormation template with a bootstrap repository (containing configuration scripts) and a Chef repository (with cookbooks, roles, configuration files, and other Chef management artifacts).

We deploy the prototype in the following four steps, as demonstrated in Figure 4.2: (1) launch cloud resources using the developed CloudFormation template; (2) establish the connection between lman and rproxy using scripts from the bootstrap repository; (3) run another bootstrap script to install the Chef HA pair on lman and rman and establish continuous back-end synchronization; and (4) clone the Chef repository on lman and rman. After these steps are completed, the HA-enabled Chef is operational and ready to perform CMS operations. The rest of the environment can be managed from lman and rman using the Chef's command-line utility called knife:

- `knife bootstrap <hostname>` – installs Chef client on a selected node.
- `knife node run_list add <hostname> "role[<role's name>]"` – assigns a specific role to the selected node. Each role prescribes running specific cookbooks, while individual cookbooks consist of recipes that can be viewed as independent configuration scripts.
- `knife ssh -m <hostname> chef-client` – triggers the update process on the new node. All recipes inside the assigned roles and cookbooks will be executed. Another version of this command can update pools of nodes, e.g., nodes with specific assigned roles, allowing

Figure 4.2: The deployment of the HA-enabled Chef with AWS components.



simultaneous configuration processes. In Chef, all these knife commands are sent to the server and then server contacts the clients. Additional information regarding the Chef architecture and knife commands is presented online [9]. In our case, the cluster-side Chef Server on lman acts as the primary (active) server in the HA pair, while rman is the secondary (passive) server in the cloud. Right now, if the primary server fails, we need to manually promote the secondary server to act as the new primary, although this step can be automated in future work using the Linux HA daemon called Heartbeat [10].

4.6 Experience

We successfully deployed the prototype and used it for over 2.5 months for managing the group's research and development environment. This environment is a Linux cluster that comprises 12 physical machines and 8 KVM-based VMs. We used the deployed HA-enabled Chef to perform cluster configuration management in such areas as networking, security, package management, and contextualization of VMs. In all these areas we continued to use and develop cookbooks, recipes and roles in the Chef repository that we started over a year ago. We transitioned smoothly from a single Chef server to the HA Chef server pair using a bash script we developed (available at [11]) that seamlessly integrated the cloud-based extension with the existing Chef installation on the cluster.

Below we discuss the prototype in detail from the security and manageability perspectives, as well describe the cloud utilization and incurred costs observed during the prototype evaluation.

4.6.1 Security

The prototype's VPN configuration is based on IPSec, which is a popular standard for providing network layer protection. It provides strong encryption and multiple authentication methods. We chose the "shared secret" method, while production installations should use RSA signatures or X.509 Digital Certificates to avoid unnecessary key management and reduce the risk of a potential exposure.

Firewalls on both ends of the VPN tunnel support fine packet filtering and restrict node access on interfaces with external connectivity. Only rproxy is allowed to send traffic to the cluster network, while, in the opposite direction, only traffic from lman enters the VPC network. The standard networking mechanisms such as Network Address Translation (NAT) and static routing expand the connectivity to allow administrative nodes in one network to connect to nodes in the other one. For other external incoming connections, lman and rproxy only allow ssh connections (with public key authentication). The cloud nodes are as secure as their cluster counterparts.

The prototype uses the VPC with a private subnet only to allow extension of our local network into the AWS cloud without exposing it to the Internet, as recommended in the AWS VPC best practices guide [12]. To reduce the operational expenses, we use the Openswan-based software VPN instead of the AWS Hardware Virtual Private Gateway offering (charged at $0.05 per connection hour at the current rate set by AWS).

DRBD, which we discuss in detail in the following section, enables asynchronous data mirroring between the cluster and cloud infrastructures. In order to protect the copy of the mirrored data in the public cloud, we can use the standard OS block device encryption on top of the DRBD devices. This security improvement will be investigated in future work.

4.6.2 Manageability

The transparency of the designed HA extension to Chef is its most prominent quality. Not only are we able to smoothly transition from the single-server to the HA configuration, but we can also survive termination of the cloud components, which will downgrade the system back to the original configuration without disruption. In fact, if we terminate only rproxy, we suspend the VPN tunnel and stop communication between the HA nodes. Afterward, we may operate the two Chef head nodes independently, i.e., manage the cluster and cloud environments with the corresponding Chef instances. Because the two head nodes are nearly identical, this event forms two smaller clusters with independent head nodes. More importantly, though, this forms two independent networks, and allows for one full side of the configuration to be brought down in the event of a failure without affecting the rest of the nodes. This also allows the deployment of an entirely new cloud component to the system without affecting the operability of the nodes that are still running on the cluster side.

Chef provides excellent support for managing complex systems. Configuration updates in Chef are idempotent, which means that an update can be applied to a node multiple times, and, regardless of the node's previous state, those updates will always yield identical configurations. Additionally, Chef provides rich functionality for node orchestration, monitoring, and provisioning. The Chef's knife command-line utility can be viewed as an orchestration tool for managing pools of nodes that is capable of replacing pdsh, a Parallel Distributed Shell [13] that is often used on clusters. For example,

```
knife ssh "role:  vm" chef-client
```

triggers application of all specified configuration actions on all VMs (i.e., nodes with the assigned "vm" role). For monitoring,

```
knife ssh "role:  fen" "service iptables status"
```

prints out all firewall rules from all front-end nodes. Similarly, knife can be used for polling performance counters.

In the context of hybrid environments, the provisioning capability of knife is powerful and convenient. With the Amazon EC2 plugin enabled, a single command:

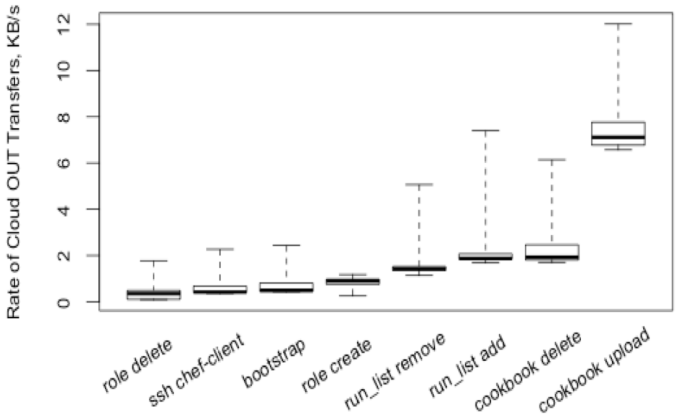
```
knife ec2 server create -I ami-XYZ -G SecurityGroup
-r "role[base]" --node-name CloudVM
```

launches a new on-demand VM in AWS Elastic Compute Cloud (or in a particular AWS VPC if specified), uses the provided image and security group IDs, waits until it boots, installs Chef client on it, registers the VM with the Chef server under the given name, and assigns the given role to it. We often used this powerful command to launch development VMs for testing prototype components.

As prescribed by the popular "infrastructure as code" model, we implemented infrastructure components "as code" and developed the CloudFormation template, installation and integration scripts in the bootstrap repository, and cluster management code inside the Chef repository.

DRBD, the open source kernel driver and user-space utilities for storage mirroring in HA clusters, allows further outsourcing of administrative functionality into the cloud, specifically in area of image management. DRBD supports asynchronous data replication and can operate efficiently across WAN and Internet links. In the prototype, the EBS volume attached to rman stores not only the Chef's back-end data but also backup copies of VM images that are used on our cluster. Specifically, we utilize 11 qcow2 sparse images of KVM-based VMs which in aggregate occupy approximately 41 GB. DRBD allows us to reliably store up-to-date copies of these images in the cloud and transfer them back to the cluster if they aren't available locally (e.g., due to a file system failure). This optional image backup addition integrates well with configuration management and enables schemes where backup images can be transferred back to the cluster and deployed in recovery scenarios via Chef cookbooks, where these operations are implemented along with the rest of administrative procedures (although they are currently performed manually in the prototype).

Figure 4.3: The rate of OUT* traffic generated by common knife commands.



4.6.3 Cloud Usage and Cost

The proposed cloud-enabled CMS is viable as long as the total cost of used cloud resources is not prohibitive. The developed prototype uses three types of AWS-provided cloud services: on-demand instances, storage volumes, and data transfers. Assuming that the free tier is exceeded, two m1.small instances, rproxy and rman, running persistently at $0.0071/hour generate a monthly bill of $10.22. For the rman's EBS volume we use a 45GB General Purpose SSD EBS volume, which is sufficient for storing both Chef's data and necessary cluster VM images. At $0.10 per GB/month, this volume's monthly cost is $4.50. While the cost of instances and storage is fixed, the cost of transfers depends on the level of utilization of the deployed instances and storage.

AWS only charges for outbound data transfers, hereinafter referred to as"OUT" transfers, where data is sent from AWS to the Internet. To determine the cost of individual Chef operations and the aggregate cost of utilizing Chef over time, we ran a series of 160 experiments and collected logs capturing all Chef operations executed in over 2.5 months of prototype evaluation. For each of the evaluated operations we performed 20 tests in which we measured OUT traffic using the Linux ifstat utility. The OUT rate, calculated as the ratio between the volume of the OUT traffic and the command's run time, for the evaluated knife commands is shown in Figure 4.3. All rates

Table 4.1: Distribution of knife Commands and OUT* transfers.

knife command	Count	Avg. OUT transfer, KB	Total OUT transfer, KB
ssh -m <NODE> chef-client	179	17.77	3181.64
node run_list add	69	17.79	1227.58
cookbook upload	62	65.38	4053.37
bootstrap <NODE>	47	22.33	1049.68
node run_list remove	38	14.76	560.80
role create	23	6.72	154.55
role delete	21	3.15	66.13
cookbook delete	21	19.48	409.05

are consistent, despite the outlying maximum values. Table 4.1 shows the distribution of knife commands in the history log, average volumes of OUT transfers obtained in our experiments, and the calculated estimates of the total OUT transfer volumes. By adding those estimates, we determine that Chef in our prototype generated the total of 10.7MB of OUT transfers, which incurs a negligible cost of less than 1 cent (at the rate of $0.09/GB for the first 10TB/month). This confirms our assumption that in the proposed architecture data is moved primarily into the cloud (free of charge), not outside.

If we are ever required to transfer the copies of VM images back to the cluster, we will incur the cost of $3.70 to transfer all our images. In exchange for the combination of this insignificant transfer cost and the cost of the corresponding EBS space, we obtain confidence in that we are always able to use the images when necessary, even if the local copies are unavailable, and avoid the significant overhead of rebuilding them.

During the prototype evaluation, the total cost of all utilized cloud services (instances, storage, and transfers) remained under $14.72/month.

4.6.4 Local Highly Available Configuration

In addition to the proposed cluster-cloud implementations, we envision useful configurations that leverage HA Chef pairs that are deployed on either side. Trading off the reliability benefits

of geographically distributed systems, systems with two local Chef head nodes gain simplicity and further manageability. Those systems do not require such components as the VPN tunnel, NAT, and the custom static routing rules, and therefore their deployment and maintenance require less administrative effort.

The most significant advantage provided by the local HA CMS configurations is the ability to upgrade CMS head node's hardware or move one of the heads from one physical server to another without CMS downtime. For instance, when the primary head node's server needs to be serviced or decommissioned, the secondary head node is promoted to be the primary, and then the serviced server or a brand new one is configured to join the new primary in the contemporary HA pair. In our opinion, leveraging the described scheme on our cluster over the last year (prior to developing the HA Chef prototype) would have simplified and accelerated transitions where on two different occasions we moved the Chef head node from one physical server to another.

4.7 Related Work

In the large body of previous work on configuration management, the following studies have strong ties to HPC. In [14], the authors describe LosF, a Linux operating system framework, and Lmod for user software management. The Engage system in [15] addresses structural dependencies and offers unique functionality for static checks and configuration verification. The Wallaby system in [16] provides high-level semantic control over Condor high-throughput clusters. In [17], the authors describe a CMS that distinguishes between the "same" and "different" relationships and achieves code reduction. The authors of [18] present TOSCA, the Topology and Orchestration Specification for Cloud Applications, and investigate integration of novel model-driven approach with traditional configuration management techniques and frameworks. In [19], the authors investigate dynamic configuration management for industry applications in the cloud and apply staged configuration techniques, as well as methods form software product line engineering. Another approach is demonstrated in [20], where the REpresentational State Transfer (REST) mechanism is shown to be useful for building and integrating configuration management systems in the context

of cloud applications and services.

Our work differs from previous research by our emphasis on CMS architecture and implementation rather than experimentation with semantics of CMS operations. Additionally, to the best of our knowledge, this is the first effort that investigates the configuration management in the context of hybrid, cluster-cloud environments and leverages cloud resources to increase persistency and reliability of an open-source configuration management solution.

Chapter 5

Introducing Configuration Management Capabilities into CloudLab Experiments

Users of CloudLab (and other GENI-derived testbeds) commonly use image snapshots to preserve their working environments and to share them with other users. While snapshots re-create software environments byte-for-byte, they are not conducive to composing multiple environments, nor are they good for experiments that must run across many versions of their environments with subtle differences. This section describes our initial work on an alternative experiment management system. This system is built on expendable instances of the Chef configuration management system, and can be used "on top of" existing testbeds.

5.1 Experimentation Using Testbeds

Software environments in modern distributed systems demonstrate great diversity and high complexity. Hypervisors, virtual machines (VMs), cloud frameworks, containers, databases, and computing frameworks – these are some of the types of components used in modern software stacks. Individual components need to efficiently utilize available hardware resources as well as interact with one another. Integration of independently designed and developed tools with cutting-edge versions and capabilities is particularly challenging.

CloudLab [1] is an NSF-funded flexible scientific infrastructure for supporting fundamental advances in cloud architectures and applications. It is designed to facilitate research and development involving cutting-edge software environments. CloudLab provides root-level access to hetero-

geneous resources at three geographic locations: datacenters at the University of Utah, Clemson University, and the University of Wisconsin. CloudLab users combine resources from these sites into environments to experiment with architectures and software of their choice. CloudLab is built on the Emulab software base[16], which has evolved to incorporate ideas from the GENI [5] facility.

The Adaptable Profile-driven Testbed (Apt) [2], also developed and hosted at the University of Utah, is designed with an emphasis on consistency, transparency, and repeatability. Apt and CloudLab are tightly integrated: experiments on Apt and CloudLab can be managed through the CloudLab portal [3], and in this study we treat Apt as another cluster within the CloudLab environment. The two systems share the following concepts:

- profiles – XML-based RSpec [4] documents (developed as part of the GENI [5] project) describing specific experiment hardware and software. The purpose of these profiles is to describe an environment that is either used repeatedly by one user to run many experiments, or is used as a way for users to share software artifacts with one another.

- experiments – instances of profiles. When a profile is instantiated, its specification is realized on available resources that satisfy the environment described in the RSpec. The experiment owner is granted root-level access to these resources for a fixed period of time.

- snapshots – saved copies of node filesystems [6]. Users create these full-disk custom images when they need to preserve custom configurations in their experiments. They create new ones or update the original profiles to launch future experiments with their snapshots rather than images provided by the facility.

The resource allocation model on CloudLab is built around temporary leases: every resource, after being utilized by one user, is reclaimed and given to another user. If "setting up" the environment to run an experiment is complicated and/or time-consuming, the user is encouraged to snapshot the configured nodes in order to instantiate identical configurations with no additional effort. The snapshot model fits will the "build once, test repeatedly" experimentation, where users

perform the bulk of necessary configuration work at once, snapshot the customized nodes, and proceed to testing: simulate workloads, run benchmarks, gather utilization statistics, visualize it, etc.

In contrast, while developing experiments to analyze performance and energy efficiency of the CloudLab hardware, we encountered a situation where the selected software stack continually evolved. We frequently added new and modified existing software components. We confronted the following drawbacks:

- Custom images cannot be automatically "merged". If Image A and Image B each have a different set of tools installed, there is no simple way to create an Image C that combines both sets of tools.
- Much effort is required to evaluate multiple versions of the selected software. For instance, to evaluate the impact of a set of compiler options on the performance of a particular tool, we need to rebuild all performance-sensitive tools multiple times with different option sets, with each requiring an additional custom image.

The coarse granularity of the snapshotting mechanism is the root cause of these drawbacks. With no fine-grained mechanisms, users encounter the described overhead when developing complex software environments, contributing to the rapid growth of the number of snapshots. More importantly, if we share our snapshots with other CloudLab users, we cannot expect them to reassemble those snapshots in order to reuse individual tools. This raises a platform-level question: how can CloudLab support efficient tool reuse among its users? Since CloudLab has over 800 users and 1,000 profiles after one year of operation, the importance of this question cannot be overstated.

In this section, we describe our design and implementation of the CloudLab infrastructure for efficient management and reuse of software components. We describe the software artifacts we have developed, which CloudLab users can leverage and customize. Using the developed profiles and custom scripts, we streamline the creation of experiments with instances of a configuration management system called Chef [31]. We demonstrate how we use expendable, short-term instances

of Chef to manage nodes inside CloudLab experiments and consistently build complex software environments. By describing our progress with orchestration of software of our choice using Chef, we aim to set an appealing example for other users and encourage them to invest effort in the development of infrastructure code. We also demonstrate how we structure the developed infrastructure code to increase its usability.

5.2 Motivation for Improved Configuration Management

The experiment that we use as motivation and example in this section is one that facilitates performance analysis of the CloudLab hardware using the GCC compiler, OpenMPI for message passing, the PETSc [8] library with routines for scientific computing, and HPGMG [17], a robust and diverse benchmark that is representative of HPC applications. (We do not assume reader familiarity with these tools in this study.) We suppose that each tool can be properly administered (e.g., using instructions from the corresponding websites) and treat them as components of a complex software environment. In such environments, issues related to customization and composability are among our main concerns.

After we install and configure these tools on a CloudLab node, we can snapshot the node. With this snapshot, we can recreate this configuration in future experiments. However, as soon as we add higher-level tools for execution management, in our case SLURM [10], we need to create two images: one for the SLURM controller node, and another for compute nodes. A distributed filesystem, such as NFS, requires the server- and client-side images as well. The number of images begins to multiply even further if we consider running experiments on different hardware architectures. Thus, the Utah cluster is built with 64-bit ARMv8 CPUs, Wisconsin – Haswell CPUs, Clemson – Ivy Bridge CPUs, and APT – Sandy Bridge CPUs. To optimize performance, we must build libraries and benchmarks with cluster-specific compilation flags. The desired performance analysis requires the configuration of 4 experiments, each with at least 2 SLURM nodes. This amounts to 8 node configurations with unique optimizations applied.

We expect the software environment to evolve over time. In the worst-case scenario, we

will encounter the overhead of applying each modification to the created images up to 8 times. For instance, over the course of our experiments we evaluated three versions of GCC: 4.8.2, part of the the default Ubuntu 14.04 image, the 5.2 version with performance improvements for the hardware on the Utah clusters, and also the latest 5.3 version. We performed limited testing of several configurations with these compilers, but chose to search for an efficient alternative to the manual creation of all 24 node configurations. We also plan to equip our experiments with tools for tracking instantaneous power draw and the total energy consumed over time (described in section 5). Similarly, a stand-alone image with such tools will be insufficient; instead, we intend to run the energy-focused tools alongside the performance tools. We would need to create many configurations by composing these sets of tools, each time with unique modifications applied to either of the sets.

Aiming to reduce the overhead in the configuration of our experiments, we decided to invest effort in automation of installation procedures. We consider components, such as SLURM and NFS, for which the installation processes are exactly the same across all node types and independent of the rest of the software environment. After we develop the automated installation scripts once, we can run them in numerous configurations with minimal effort. In contrast, we need to incorporate platform-specific flags into our scripts for building high-performance libraries and benchmarks. Despite the difference in the flags, many actions in those scripts should be similar for different platforms. A single investment in configuration scripts will help save time in configuring future experiments, even if we need to adapt script parameters for particular platforms. We are confident that automation will reduce the amount of overhead, as well as greatly increase consistency. The developed code should also yield more transparency comparing to a growing number of custom images.

5.3 Approach

In order to systematize the outlined automation effort, we leverage a configuration management system (CMS). CMSes are built around the "infrastructure as code" principle, which dictates that all administrative procedures must be implemented as code and maintained, versioned, shared,

and reused similar to other types of software. Embracing this code-centric vision, we plan to establish a public repository with the CloudLab infrastructure code, encourage community members to contribute, and incrementally expand the set of supported components.

In our previous work [11], we used a CMS called Chef (developed by Chef Software, Inc.) to manage a small group-owned experimental HPC cluster. In that environment, Chef proved to be a powerful tool with support for a variety of administrative procedures. We showed how Chef can be deployed in highly available configurations with increased reliability and longevity, which are desired in production scenarios. In contrast, this study investigates creation of expendable, short-term CMS deployments, which correspond to the CloudLab resource model, where nodes are allocated to users on a temporary basis. We enable the following workflow for the CMS-based experiment management: 1) a user launches an experiment with a fully functional instance of the selected CMS, 2) he or she develops infrastructure code and uses the CMS to execute it on the nodes under his control, 3) before the experiment expires, preserves his or her code, and 4) at later times, creates new instances of the CMS, obtains copies of the code, proceeds to manage new experiments, and continues the code development.

Our experience suggests that Chef is an appropriate system for this workflow. We attempt to automate the installation of necessary components and dependencies to create complete, working Chef environments. Much of the deployment process will be hidden from the user. To obtain identical instances of Chef in steps 1 and 4, CloudLab users will simply choose a profile and instantiate it via the CloudLab portal. They may run several instances of the profile simultaneously on the same or different clusters.

We must pay special attention to how we develop infrastructure code in order to increase its usability. In Chef, infrastructure code is arranged into so-called cookbooks, which include one or more Ruby scripts called recipes. Recipes typically use Chef resources, which are high-level constructs specifically designed to describe many common administrative tasks such as install a package, download a file, mount a filesystem, etc. Even though Chef recipes can include arbitrary Ruby code, resources help reduce the amount of Ruby code and spare the recipe developers from

being Ruby experts.

While developing cookbooks in step 2, we focus on the following goals:

- Develop the cookbooks which run on the nodes of user choice with no modification. While running these cookbooks, novice users will install the corresponding components and practice the workflow that is the same all cookbooks: assign cookbooks to nodes, run the cookbooks, and check the output.

- Ensure that advanced users can perform customization with minimal effort. Such parameters as version numbers, URLs of packages, compilation flags, installation paths, etc., need to be easily customizable rather than hard-coded.

- Transparently support different platforms. Where possible, moving from one cluster to another should require only slight parameter changes (e.g., platform-specific compilation flags) and minimal modification in the code. Additionally, we should avoid code redundancies: rather than developing cookbooks such as ARM-X, IvyBridge-X, and Haswell-X with similar code, we should develop a single cookbook for X which is capable of supporting X on all appropriate architectures.

Preservation of the code in step 3 of the workflow is straightforward, since all Chef cookbooks and other code artifacts are developed inside chef-repo, a directory that is version controlled by default. The code can be "pushed" to a public repository, e.g., hosted at GitHub, and later "pulled" on the same or ad different Chef server.

5.4 Achitecture

A minimal Chef deployment includes four components– a server, a client, a workstation, and a chef-repo – all installed on the same node. The command line utility called knife plays the role of the workstation and provides an interface to server operations: bootstrap a client (i.e. install necessary packages and register the client with the server), assign cookbooks to the client, and

query the environment, etc. The server is a central authority for storing cookbooks and cookbook-to-node assignments. The client contacts the server to obtain the latest configuration when the "chef-client" command is issued (Chef supports the pull model, as opposed to the push model used in other CMSes). The server returns the assigned cookbooks, and the client executes them. In multi-node environments, each node runs a client and receives updates from the server.

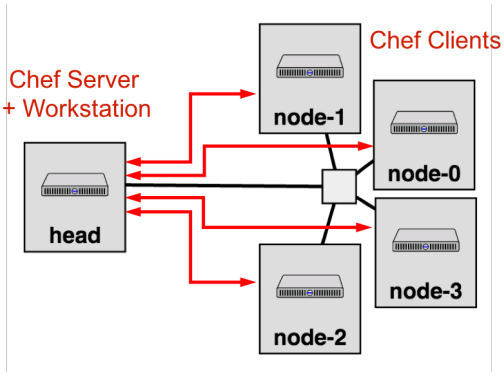
We developed a CloudLab profile called Chef-Cluster [12]. It provides uncomplicated control over the deployment and configuration of Chef environments using the profile parameters, including:

- N, the desired number of client nodes,
- URL for the chef-repo used for this experiment,
- cookbooks obtained from Chef Supermarket [13] – the repository with many cookbooks developed by the global community of Chef developers,
- a set of cookbooks assigned to all clients.

When instantiating Chef-Cluster via the portal, we choose appropriate values for these parameters and proceed to the experiment placement step. We select the CloudLab site on which we prefer to instantiate Chef-Cluster with the specified parameters. The portal displays the experiment's physical topology where all created nodes are interconnected with a single experiment network. In contrast, the logical structure inside the experiment is such that all client nodes can be configured from the server node. Figure 5.1 illustrates both the physical (in black) and logical structure (in red) inside an experiment with $N = 4$. Since the server node also runs a client, we can use knife on the server node to configure all five experiment nodes.

Once the cluster is selected and the experiment name is specified, the experiment is launched. At the back end, a Python script generates an RSpec using geni-lib [14]. The RSpec is populated with the XML elements that describe the environment for the experiment. Additional elements are appended to the RSpec for the described parameters, and the node attributes "Install Tarball" and "Execute Command" are used to download and run startup scripts.

Figure 5.1: Physical and logical structures in a 5-node experiment – instance of the Chef-Cluster profile. The small box in the middle represents a networking switch connecting all nodes.



To produce a working Chef environment with the shown logical structure, we need to convert a pool of interconnected but independent nodes into a cluster environment with centralized management. The startup scripts perform two tasks: enable the server to issue remote commands to the clients, and inform the server about the available clients. For the former, our scripts use the geni-get key call to generate experiment-specific public and private keys. These keys are the same among all nodes, enabling secure host-based authentication within the experiment. The rest of our scripts run on the server node and leverage this authentication. For the latter, our scripts process experiment metadata included in the manifest. In contrast with the abstract requests in the profile RSpec, experiment manifests contain information about the actual experiment resources. Obtained via the geni-get manifest call, this information allows our scripts to register the client nodes with the Chef server. The scripts also assign the specified cookbooks to the nodes based on the specified parameter value, which becomes available in the manifest.

The configuration process usually takes several minutes. When it completes, the script sends a notification via email to the user who launched the experiment. To reduce the learning curve for novice users, the email includes several knife commands for querying the created Chef experiment and their output. For instance, the output of knife cookbook list and knife node list reports which cookbooks and nodes are recognized by the server. After inspecting the environment, the user can

Table 5.1: Developed Chef Cookbooks, Their Dependencies, and Components.

Cookbooks	Description	Required Cookbooks	Recipes	Attributes
emulab-gcc	Install the latest available version of GCC	apt	1	N/A
emulab-openmpi	Install and make available OpenMPI	emulab-gcc	1	4
emulab-slurm	Configure SLURM controller and compute nodes	apt, mysql	3	18
emulab-nfs	Install NFS server and clients, export/mount directory	nfs	3	12
emulab-petsc	Obtains and builds PETSc	emulab-openmpi	1	2
emulab-hpgmg	Obtains and builds HPGMG	emulab-hpgmg	1	4
emulab-R	Install the latest available version of R package	emulab-gcc	3	1
emulab-shiny	Install Shiny server and necessary R library	emulab-R	1	2
emulab-powervis	Install PowerVis and its dependencies	emulab-shiny, apache2	1	1

proceed to configuring the nodes using their assigned cookbooks. We intentionally do not automate this step in our scripts, allowing the user to log into the nodes in the desired order, trigger the configuration processes by issuing the chef-client command, and examine the output.

5.5 Implementation

We established the emulab/chef-repo repository [15], which provides public access to the cookbooks developed in accordance with the goals from section 3. Currently, the repository includes 13 Chef cookbooks and 24 recipes, which focus primarily on the configuration of performance- and energy-related tools. CloudLab users can fork this repository, customize the existing cookbooks, and use the URLs of their derived versions of the repository when they create their Chef-Cluster experiments. Pull requests can also be submitted; the new code will be thoroughly tested before becoming a part of the official repository.

If [15] is selected when Chef-Cluster is instantiated, the repository is cloned and the cookbooks listed in Table 1 become available on the created server. Even though we developed these cookbooks to configure our specific software stack, some of them can be immediately useful to other CloudLab users. For instance, emulab-gcc, emulab-R, and emulab-nfs are general-purpose cookbooks which install the latest versions of the corresponding components with minimal effort. To enable easy customization, cookbooks optionally include attributes – the variables which change the cookbook's behavior. We added attributes in our cookbooks for setting version numbers, URLs of packages,

compilation flags, installation paths, among other parameters.

We use the emulab prefix in the names of our cookbooks to emphasize that they are developed for Emulab [16] and derived testbeds such as CloudLab and Apt. This helps distinguish them from the Supermarket cookbooks, which are installed in the same directory on the server node. Our cookbooks leverage four Supermarket cookbooks – apt, mysql, nfs, apache2 – which are sufficiently documented and actively supported by many Chef developers.

The cookbooks emulab-R, emulab-shiny, and emulab-powervis demonstrate an example of cookbook chaining. CloudLab provides access to power consumption traces from every physical server. To simplify the usage of this data, we developed PowerVis, a dashboard for analysis and visualization of power traces. This dashboard is an application in Shiny, the environment for interactive web applications powered by the R statistical language. With the help of dependencies expressed in the cookbook metadata files, emulab-powervis calls emulab-shiny, which, in its turn, calls emulab-R. Therefore, emulab-powervis is a top-level cookbook which installs all necessary underlying components and enables analysis and visualization of the experiment-wide power data. Similarly, emulab-hpgmg consolidates the installation of HPGMG, PETSc, OpenMPI, and GCC.

Since a single model cannot fit perfectly all scenarios, Chef provides the developer with enough flexibility to organize the code into structures that match specific environments and applications. Some of our cookbooks support different platforms with the help of multiple recipes. For instance, inside emulab-R, the default cookbook identifies the node architecture and calls either the aarch64.rb (on ARMv8 nodes) or the x86.rb (on the rest of the nodes) recipe to take platform-specific actions. In contrast, emulab-hpgmg uses a single recipe which selects optimal compilation flags from its attributes and runs the same code on all platforms.

5.6 Experience

We used Chef to manage configurations inside experiments on each of the CloudLab sites and turned many experiments into fully functional computing clusters with the described performance and energy tools. We often incrementally built our clusters: we started by installing HPGMG (and

its dependencies); after single-node tests, we enabled multi-node runs by installing SLURM and NFS; then, we installed PowerVis to perform power and energy analysis of the performed runs.

Using the node names shown in Figure 5.1, below we illustrate how we configured our clusters. Initially, we ran:

```
# knife node run_list set head "emulab-hpgmg"
```

Then, the `chef-client` command executed on head triggered the execution of the assigned emulab-hpgmg cookbook. These two steps, the assignment and the configuration, can easily be repeated many times throughout the experiment lifecycle. We can use the add function to modify run_lists, the lists which define what is executed on individual nodes:

```
# knife node run_list add head emulab-slurm"
# knife exec -E 'nodes.find("name:node-*") {
|n| n.run_list.add("emulab-slurm"); n.save}'
```

While the first command assigns emulab-slurm to the server node (head), the second command assigns the same cookbook to the client nodes (node-[0-3]) using the wildcard matching for the node names. The client nodes can now be configured one at a time. Alternatively, if the order is insignificant, we can trigger the batch update by running "knife ssh 'name:*' chef-client" on the server node (emulating the push-based model). We can assign additional code artifacts to our nodes:

```
# knife exec -E 'nodes.find("name:node-*") {
|n| n.run_list.add("<artifact>") ; n.save}'
```

where ⟨artifact⟩ can take one of these forms: "recipe [⟨cookbook name⟩::⟨recipe name⟩]", "recipe [⟨cookbook name⟩]", or "⟨cookbook name⟩". In the last two cases the cookbook's default.rb recipe is used. Moreover, we can also use the "role[⟨role name⟩]" notation to assign Chef roles – the higher-level code artifacts which set attributes, complementing and possibly overriding attributes specified inside individual cookbooks, and consolidate ordered groups of cookbooks.

In our computing clusters configured inside CloudLab experiments, we typically ran emulab-

hpgmg, emulab-slurm, emulab-nfs, and emulab-powervis (along with their dependencies) on the server nodes and executed emulab-hpgmg, emulab-nfs, and emulab-slurm on the clients. After performing the aforementioned incremental development and thoroughly testing each component, we streamlined this assignment with the help of custom roles. The developed controller and compute roles (available at [15]) consolidated the server- and the client-side cookbooks. We also ensured that the individual cookbooks included in these roles work together. For instance, the roles set the attributes such that HPGMG, PETSc, and OpenMPI are installed inside the directory used as the cluster NFS share. These tools, once installed on the server, become available on all clients when the NFS share is mounted. Consequently, we can reduce the number of cookbooks included in the compute role.

The controller and compute roles have sufficiently satisfied our configuration needs. We used these roles to consistently create computing clusters at different CloudLab sites and performed many benchmarking experiments. While running HPGMG, we gained better understanding of the benchmark code and the properties of the available hardware. In addition to the performance analysis, we processed power traces using PowerVis. Not only can we identify the CloudLab site that is the most appealing for running HPGMG in terms of Watts and Joules, we can use energy efficiency as the deciding factor. Thus, for our 8-core HPGMG runs, the estimated energy efficiency of the Utah nodes is between 0.06 and 0.11 GF/W (gigflop per second per watt). For the same runs, this efficiency is higher by 42-55% at Clemson and 62-70% on Apt. In future work, we will perform a comprehensive analysis of the energy efficiency of the CloudLab resources using a number of diverse benchmarks. From the infrastructure standpoint, we will reuse the existing components and develop additional cookbooks only for the new benchmarks and their dependencies.

5.7 Related Work

In [17], the authors introduce cloudinit.d, a utility for contextualization of VMs, i.e. assignment of application roles to individual nodes in virtual clusters. AWS CloudFormation templates [18] allow launching virtual clusters on the AWS cloud in pre-configured states. In both cases, the

specified configurations are realized when nodes boot. Similarly, CloudLab allows users to create and use custom images. In contrast, we developed a profile where Chef naturally supports recontextualization, i.e. incremental development and modifications in node configurations, throughout the experiment lifecycle.

In [19], the authors propose custom configuration and reconfiguration capabilities. Kameleon in [20] is a software appliance builder which promotes reconstructability using a custom configuration specification syntax. In contrast, we demonstrate how users of CloudLab can create instances of Chef, a modern CMS with an active and diverse global user community. They can use Chef to develop custom infrastructure code and also take full advantage of the publicly available code developed by the community.

Among the GENI-focused projects, GENI Experiment Engine [21] and LabWiki [22] are the most relevant experiment management solutions. GENI Experiment Engine provides support for configuring experiments with Ansible, a free, open-source automation tool. While Ansible is considered intuitive and easy to operate, we anticipate that hierarchies of roles, cookbooks, and recipes in Chef are more conducive to supporting customization and composability of configurations in multi-node software environments. Additionally, Chef recipes equipped with fully featured Ruby code will likely provide more flexibility in complex experiments than Ansible playbooks implemented in YAML.

5.8 Acknowledgments

This material in this chapter is based upon work supported by the National Science Foundation under Grant Nos. 1338155 and 1419199.

Chapter 6

Active Learning in Performance Analysis

Active Learning (AL) is a methodology from machine learning in which the learner interacts with the data source. In this section, we investigate application of AL techniques to a new domain: regression problems in performance analysis. For computational systems with many factors, each of which can take on many levels, fixed experiment designs can require many experiments, and can explore the problem space inefficiently. We address these problems with a dynamic, adaptive experiment design, using AL in conjunction with Gaussian Process Regression (GPR). The performance analysis process is "seeded" with a small number of initial experiments, then GPR provides estimates of regression confidence across the full input space. AL is used to suggest follow-up experiments to run; in general, it will suggest experiments in areas where the GRP model indicates low confidence, and through repeated experiments, the process eventually achieves high confidence throughout the input space. We apply this approach to the problem of estimating performance and energy usage of HPGMG-FE, and create good-quality predictive models for the quantities of interest, with low error and reduced cost, using only a modest number of experiments. Our analysis shows that the error reduction achieved from replacing the basic AL algorithm with a cost-aware algorithm can be significant, reaching up to 38% for the same computational cost of experiments.

6.1 Introduction

Active Learning (AL) is a set of Machine Learning techniques in which the learner interacts with the source of data being learned [89]. One use of AL is to query a human or run an "expensive"

algorithm to label a datapoint for which correct labeling is difficult to predict. Intuitively, the learning algorithm makes decisions about which parts of the space it does not "know enough", and selectively invokes the expensive process to gather more supporting data to make predictions in that area. This type of learning is typically applied to classification problems, as described in [89]. The authors of [28] emphasize that only a limited number of studies have investigated AL applied to regression problems.

In this study, we apply techniques from AL to regression problems, creating regression models for performance and energy consumption data from parallel computing experiments. In this setting, our goal is to take data from existing empirical studies of the performance of HPC codes and perform regression analysis to discover the relationships between controlled variables (such as the size of the problem, the number and the speed of the used CPU, etc.) and the performance of the computation measured using such common metrics as runtime (in seconds) and total amount of consumed energy (in Joules). The constructed regression models can be used to predict the amount of time, energy, etc. that will be needed to run future computations, or as part of a performance model for a larger system. Depending on the complexity of the computation, each experiment might take seconds, minutes, or hours. Therefore, it might be infeasible to use experiment designs that run all or most of the combinations of variables (which can number in the hundreds, thousands, or more), making iterative designs that selectively run experiments that will yield the most information attractive. This is particularly true when the benchmarking budget is limited: for instance, there exists a fixed allocation on an HPC machine or a fixed maximum budget in a cloud environment.

To apply AL to regression analysis in performance studies, we need a process that both (a) produces regression models for datasets with multiple controlled and response variables and (b) produces estimates of the uncertainty associated with any point in the input space. We use Gaussian Progress Regression (GPR), also known as *kriging* in literature on geostatistics, for this purpose since it comes with a posterior probability distribution for predictions. With GPR, we can obtain the mean and the variance of the predictive distribution at every input point. The former is used to assess the quality of the investigated computational algorithm, while the latter is used in

AL to select a sequence of points where the code associated with algorithm needs to run (we also investigate an AL algorithm which uses both for point selection).

We apply the selected AL techniques to a specific performance analysis problem where the performance of the HPGMG-FE benchmark [17, 16] is investigated. In this analysis, an impact of several controlled variables on performance metrics is studied based on measurements of performance and energy usage across a number of CPU and problem-size settings. We look at two different algorithms for guiding experiment selection: one that seeks to run experiments that are expected to purely reduce prediction uncertainty, and another one that weights experiments by the predicted cost (completion time) in order to learn the most information in the shortest amount of time. In the process of running these AL algorithms on a fixed dataset (which we collected by running a series of HPGMG-FE runs on available testbed hardware), we analyze the properties of the involved mathematical and algorithmic components to ensure that the resulting system demonstrates reliability and optimality.

The main contributions of this study are:

- We propose a new framework for performance analysis based on Active Learning and Gaussian Process Regressions. This framework helps identify optimal sequences of experiments for reducing uncertainty about various quantities of interest.

- Under this framework, we develop a prototype which can be used to construct a number of diverse performance models, including models for application runtime, energy consumption, memory usage, and many others. We show how one can efficiently learn relationships between these metrics and multiple controlled variables.

- Using a dataset of measurements from over 3K parallel jobs, we compare the performance of two AL strategies, as well as their ability to reduce the prediction uncertainty for a given cumulative execution time. We confirm that the basic AL algorithm with no adjustment for the experiment cost can help efficiently construct regression models using only a modest number of experiments. However, the proposed cost-aware AL algorithm can provide sig-

nificant advantages: for a range of total cost values, it will greatly reduce prediction errors, with the maximum reduction of 38% observed on the selected datasets.

The remainder of this chapter is organized as follows: In Section 6.2, we provide a brief summary of the related work. Section 6.3 introduces the model that we use to construct GPRs over sets of measurement points, and Section 6.4 describes the particular technologies that we leverage to implement the model and also collect experimental performance and energy consumption data. Section 6.5 presents our application of several compelling AL algorithms to the collected datasets. We pay special attention to the AL "progress", and describe several metrics which help track how AL algorithms converge as the number of experiments increases. We conclude the chapter and propose directions future work in Section 6.6.

6.2 Related Work

6.2.1 Active Learning and Regression Training

AL is also sometimes referred to as optimal experimental design. In [89], the author summarizes existing approaches in AL, primarily in the context of classification problems. The existing AL methods are referred to as *Query Synthesis*, *Stream-based Sampling*, and *Pool-based Sampling*.

In [78], the authors develop an AL system for learning regressions in studies where remote sensing data is analyzed to provide insights about biophysical parameters.

Regression analysis is typically used to fit a curve – linear, quadratic, cubic, etc. – through a dataset to predict future data. It is often assumed that the dependent variable, also known as *reponse*, behaves roughly according to one of these types of relationships with respect to controlled variables (sometimes called *features*). GPR provides a finer approach to regression learning where no such assumption is made and this relationship might be more complex (e.g., linear in one region of the input space and non-linear in adjacent regions). In GPR theory, the learning of a regressor is formulated in terms of a Bayesian estimation problem (summarized in Section 6.3).

6.2.2 Performance Analysis

Raj Jain in [56] lays the foundation for experimental design in computer and network performance studies. The following classes of designs are described: simple designs (vary one factor at a time), 2^k full factorial design, and 2^{k-p} fractional factorial designs. Fractional factorial designs present a way to run less than the full number of possible experiments, but, unlike our work, do so in a **static** way: the set of desired experiments is determined *a prioi* from the factors and their levels. These designs do not change as measurements become available. The advantage to this type of design is that experiments can be arranged to specifically test for the effects (or lack of effect) of individual factors and combinations of factors, and multiple experiments can run in parallel. The downside is that the number of experiments does not typically represent the variation present in empirical data, and the selected experiment points do not necessarily represent the most "interesting" parts of the input space in terms of deviating from expected behavior. It is also difficult to create designs for factors that have more than two or three levels, and especially difficult when different factors have different numbers of level. The approach described in this chapter is much easier to apply to experiments with many factor levels and is naturally adapted to running fewer experiments in situations where there is less variation in the results.

6.2.3 Response Surface Method

In [97], the authors introduce the Response Surface Method (RSM), which is also a type of "optimal experiment design" that is related to our work in that it helps choose future experiments based on the measurements recorded so far. However, it fundamentally differs from our work: we seek to characterize the entire problem space with reasonably high accuracy, while RSM is designed to search for combinations of factors that allow reaching specified goals. In this respect, it resembles an optimization process, where "sampling" from the space corresponds to running more (possibly expensive) experiments.

6.3 General Formulation for Non-deterministic Active Learning

In [28], the authors provide a foundation for AL in regression problems. They introduce the Expected Model Change Maximization (EMCM) scheme which is based on the following selection criterion:

$$x^* = \arg\max_{x \in pool} \left(\frac{1}{K} \sum_{k=1}^{K} \|(f(x) - f_k(x))x\| \right) \tag{6.1}$$

This formula uses differences between $f(x)$, the values predicted using the primary regression model learned from the data, and $f_k(x)$, the predictions obtained using k "weak" learners, at points x. Functions $f(x)$ and $f_k(x)$ differ in the amount of training data they utilize: $f(x)$ is trained using all of the available data, while the ensemble $\{f_k(x)\}_{k=1..K}$ is trained with K subsets of the data generated via random sampling (with replacement). That study shows that the expression being maximized in (6.1) approximates the "model change" – the change of the regression parameters θ which occurs after a new datapoint x is added to the dataset and the model is retrained (where the model is fit to the data by minimizing the sum of squared differences between measurements and predictions).

Below we provide our analysis of EMCM and list the potential shortcomings of the method. It is difficult to use this method, as it is described, in performance studies for the following reasons:

- The quality of the learned regression models is evaluated using the widely-used Root Mean Squared Error (RMSE) metric:

$$RMSE = \sqrt{\frac{1}{|T|} \sum_{k=1}^{|T|} (f(x_i) - y_i)^2} \tag{6.2}$$

  where $|T|$ denotes the size of the test set and y_i is the ground truth of the example x_i. This evaluation, in principle, implies a single prediction $f(x_i)$ and a single ground truth value y_i for every value of x_i. A more general approach is required for problems where the regression learning algorithm produces a distribution of predictions (i.e. predicts the probability density function p_{x_i}) and where the algorithm needs to be compatible with

datasets with multiple y values for the same x, representing repeated measurements of a noisy function.

- Once a point (x_i, y_i) is added to the training set, it is excluded from the pool of potential future candidates for AL. In contrast, when noisy functions are studied, AL algorithms need to be able to return to points already included in the dataset and recommend obtaining additional measurements if the variance observed or predicted at those points is high.

These limitations make it difficult to run EMCM on a wide range of datasets with measurements of performance, energy consumption, memory usage, and similar quantites. In many cases, the non-deterministic nature of computer and network performance requires obtaining repeated, often numerous, measurements in order to draw reliable conclusions. Moreover, the EMCM's K weak learners effectively provide a Monte Carlo estimate of variance, which is especially noisy when the training set is small.

An attractive alternative is documented in [78]. The authors leverage Gaussian Process Regression (GPR) for selecting the most "difficult" samples. That study relies on the Bayesian principle for estimating the posterior distribution of predictions given a simple prior for regression parameters. In that process, the authors pay special attention to *hyperparameters* – the parameters that define the prior distribution and characterize the noise in the data. Those hyperparameters have a significant impact on how GPR-based models are created, and influence what experiments are selected for learning in AL algorithms. However, that paper does not explain what algorithms can be used to find optimal values of GPR hyperparameters.

Chapter 5 in [79] provides a detailed description of optimization techniques for "model selection", which refers to the process of fitting hyperparameters. Two approaches are considered: the Bayesian inference with marginal likelihood and the cross-validation (CV) with pseudo-likelihood. While the former attempts to maximize the probability of the observations given the assumptions of the model, the latter typically uses the leave-one-out scheme (LOO-CV) and the squared error loss function. In its approach, the latter greatly resembles the EMCM method which constructs an

ensemble of weak learners. Below we summarize the Bayesian hyperparameter fitting, the method we choose to focus on in this study (we leave the empirical comparison of the two methods for our future work).

Let X and y be the design matrix (where the columns are the vectors of input values) and the vector of the response measurements, respectively. The objective is to an find the underlying function $f(x)$ which best "explains" the measurements which are assumed to include Gaussian noise:

$$y = f(X) + \mathcal{N}\left(0, \sigma_n^2\right), \tag{6.3}$$

where σ_n^2 is the variance of the noise. For an unknown input vector $x_\star$, the posterior distribution for individual predictions $f(x_\star)$ takes form of a multivariate Gaussian distribution:

$$p(f(x_\star)|x_\star, X, y) \sim \mathcal{N}\left(\mu_\star, \sigma^2 x_\star\right), \tag{6.4}$$

where

$$\mu_\star = k_\star^\mathsf{T} K_y^{-1} y, \tag{6.5}$$

$$\sigma_\star^2 = k_{\star\star} - k_\star^\mathsf{T} K_y^{-1} k_\star. \tag{6.6}$$

$$K_y = K + \sigma_n^2 I. \tag{6.7}$$

The critical role in the estimation of these parameters is played by the covariance function $k(x_p, x_q)$. A covariance matrix:

$$[K]_{ij} = k(x_i, x_j), \text{ for all columns } x_i \text{ and } x_j \text{ in } X, \tag{6.8}$$

a covariance vector:

$$[k_\star]_i = k(x_\star, x_i), \text{ for all columns } x_i \text{ in } X, \tag{6.9}$$

and a scalar,

$$k_{\star\star} = k(x_\star, x_\star) \tag{6.10}$$

are calculated using the selected function $k(x_p, x_q)$.

Both [78] and [79] describe the squared exponential (also referred to as the radial basis function or RBF) as a common choice of the covariance function:

$$k(x_p, x_q) = \sigma_f^2 \mathsf{exp}\left(-\frac{|x_p - x_q|^2}{2l^2}\right), \tag{6.11}$$

where the length scale l and the amplitude σ_f^2 are hyperparameters, along with the noise level σ_n^2. Appropriate values need to be selected for all three hyperparameters in order for the regression to match the given dataset (X, y) and be able to produce reliable predictions.

According to the Bayesian inference with marginal likelihood, the following quantity, referred to as the *log marginal likelihood* (LML), is minimized with respect to the hyperparameters:

$$\log p(y|X, l, \sigma_f^2, \sigma_n^2) = -\frac{1}{2}\left(y^\mathsf{T} K_y^{-1} y + \log|K_y|\right) + \mathsf{C}, \tag{6.12}$$

$$(l, \sigma_f^2, \sigma_n^2) = \underset{l, \sigma_f^2, \sigma_n^2}{\arg\min}(\log p(y|X, l, \sigma_f^2, \sigma_n^2)). \tag{6.13}$$

After solving this optimization problem and fitting the hyperparameters, we can use the constructed GPR model to obtain the mean and the variance of predictions for every input point x. Instead of the Monte Carlo criterion (6.1), at every iteration of AL we propose choosing x for which the variance of predictions is the highest, with or without adjusting for the expected cost. In Section 6.5, we present two AL algorithms which we develop based on this idea and describe their performance. Our analysis shows that the proposed AL algorithms run as expected even when the number of data points is small, for instance, only a single measurement of a quantity of interest is available at the beginning (the situation where EMCM is unlikely to perform well).

6.4 Implementation

We developed a prototype capable of choosing experiment candidates based on AL with GPR for noisy performance and power data. The prototype, which is implemented in Python, allows us to analyze characteristics of large collections of computational jobs, as well as provides insights

into properties of the implemented AL algorithms. Although a detailed analysis of computational requirements and the scalability of these algorithms is out of the scope of the current work (planned for further research), we successfully analyzed numerous sequences with hundreds and thousands of experiments.

The prototype, given a dataset with the design matrix X and the vector of response values y, partitions it into 3 sets: Initial (for initial regression training), Active (for one-at-a-time experiment selection with AL), and Test (for prediction quality analysis). In our evaluation, we use random partitioning with selected ratios between these sets. Thus, we typically used the Initial set with a single experiment, which corresponds to realistic scenarios where in many performance studies an application is first run on a new platform to verity correctness and the subsequent runs are used to evaluate the performance. The Active and Test sets in our analysis split the remaining experiments roughly with the 8:2 ratio.

In addition to single realizations of AL, our prototype is capable of running batches of random partitions of the same dataset. The aggregate results, such as the average error and the average cumulative cost of experiments, provide insights into how the AL process behaves independent of the initial state (i.e. the partition properties) and allows us to compare alternative algorithms for candidate selection.

To test the prototype and investigate the available tuning mechanisms in the developed AL algorithms, we ran the prototype on two datasets which we collected from running parallel computational jobs on the hardware available for short-term dedicated experiments and provisioned via the CloudLab portal [33]. Below we describe in detail the leveraged technologies; in Section 6.5.1 we characterize the collected datasets and provide our intuitions about them.

6.4.1 Leveraged Technologies

In order to gather practical empirical data for our analysis, we leveraged several existing technologies:

- **CloudLab** [81]: Performance and power data were gathered on CloudLab, a facility that gives users exclusive bare-metal access to compute, storage, and network resources. Because access is exclusive, we could be certain that the data collected from our experiments was not significantly impacted by other users of the facility. CloudLab also provides power measurements at the server-level, a critical feature for the energy consumption modeling in this study. We collect power traces with frequent recordings of the instantaneous power draw (in Watts) from the on-board IPMI sensors and infer per-job energy consumption estimates (in Joules) using the recorded timestamps.

- **HPGMG** [17]: A robust and diverse benchmark that is designed to be representative of modern HPC applications. This benchmark is used to rank supercomputers and HPC systems based on geometric multigrid methods. We ran HPGMG-FE, the compute- and cache-intensive component which solves constant- and variable-coefficient elliptic problems on deformed meshes using Full Multigrid.

- **scikit-learn** [14]: A Python module with efficient tools for Machine Learning and data analysis. Specifically, we leverage the code for Gaussian Processes [15] in the latest development version (0.18.dev0).

We used the Wisconsin cluster on CloudLab to instantiate a 4-node homogeneous environment where every physical machine is equipped with 2 8-core Intel E5-2630 v3 Haswell CPUs, 128GB of RAM, and 10Gb NICs (full hardware description is provided at [6]).

We constructed a fully functional compute cluster out of the provisioned individual machines by installing and configuring the following software stack: NFS for sharing job input and output between the nodes, OpenMPI (ver. 1.10.0) for message passing, and SLURM (ver. 15.08) for resource management and scheduling. The latest stable versions of the PETSc library [20] and the HPGMG code were built and executed. In this configuration process, we relied on the infrastructure related to the Chef configuration management system [31] – the Chef-Cluster profile and the relevant cookbooks (bundles of installation and configuration scripts) developed in the previous work and

Table 6.1: The Parameters of the Analyzed Datasets.

	Dataset: Performance	Dataset: Power
# Jobs	3246	640
Responses	Runtime (S)	Runtime (S), Energy (J)
Runtime, S	0.005 - 458.436	0.005 - 458.436
Energy, J	-	6.4e3 - 1.1e5
Variables	Operator: poisson1,poisson2,poisson2affine Global Problem Size: 1.7e3 - 1.1e9 NP: 1,2,4,8,16,24,32,48,64,96,128 CPU Frequency (GHz): 1.2,1.5,1.8,2.1,2.4	

documented in [44].

HPGMG-FE jobs with different parameters (e.g., operator type, problem size, etc.) were organized into batches and submitted to the job queue, after which SLURM managed their execution on the available nodes. After completion, we collected all relevant information, including benchmark output, error logs, SLURM accounting information, power consumption traces, and system information. Then, we transferred that data from the cluster to a local workstation to plot it and analyze it using the developed AL algorithms.

6.5 Evaluation

6.5.1 Understanding the Dataset

For experimental evaluation, we run our prototype "offline", consulting a database with the collected data. We demonstrate the advantages which AL algorithms can provide on these or similar datasets, while the target use case for practical applications is the "online" operation, where every iteration of AL includes selecting an experiment, running it, and using the experiment outcome to update the underlying GPR model.

Table 6.1 provides a high-level summary of the collected datasets. It lists all considered responses and controlled variables available in the datasets. It is worth mentioning that the intervals

in which Runtime, Global Problem Size, and NP (number of processes) change are relatively large, allowing us to explore the behavior of the application across a domain where the growth along some of the dimensions is significant. For instance, the runtime increases from its smallest recorded value to the largest by growing 5 orders of magnitude. Also, CPU Frequency varies from 1.2 to 2.4 GHz, which are the low and the high limits on the selected machines' CPUs. The datasets include up to 3 repeated experiments for every combination of the controlled variables. The complete datasets, with up to 46 attributes for each job – controlled variables, job execution properties reported by SLURM (e.g., memory usage on every node), and the listed responses – are available in the CSV format at [39].

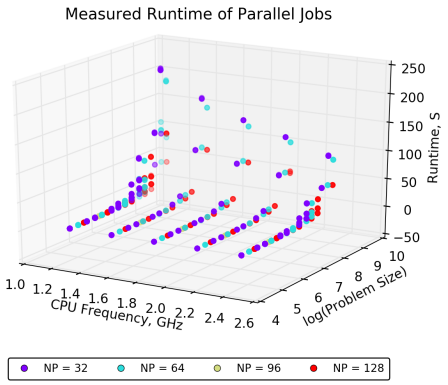
In order to validate our intuitions about this data, we fix the Operator variable (choose "poisson1"), select several levels of NP, and show these subsets on 3D plots in Fig. 6.1. These plots demonstrate that the variance in the Power dataset is much higher comparing to the Performance dataset. There are also fewer points in the Power dataset due to the fact that the collected power traces from the utilized machines had gaps and we excluded a number of jobs with insufficient number of corresponding power draw records (less than 10 for 60 seconds of computation). The per-job Total Energy estimates, obtained from the traces of instantaneous power draw via numerical integration, can be interpreted as approximation of the actual per-job energy consumption. Calibration of these estimates using measurements from a set of physical electrical power meters falls beyond the scope of the current analysis; it constitutes a context for an appealing future study where, in the AL terms, experiments for which physical measurements are available can be used in the modeling with higher confidence, while the derived IPMI-based estimates are marked with lower confidence and the corresponding experiments are recommended for repeated execution.

Fig. 6.2 shows the same subsets with log-transformed responses. The plot for the Performance dataset confirms the linear growth of Runtime along the problem size dimension, for which the plot also uses the log-transformed scale. One downside of this log-transformed representation is that it becomes difficult to see the growth along the frequency dimension, therefore, the original, non-log plot needs to be always referenced when AL decisions and GPR predictions in this space are

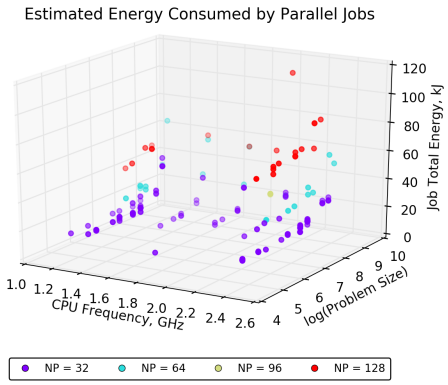
analyzed. As we can see, log-transformation does not significantly change the structure of the Power dataset.

In the remainder of the chapter, we describe how we use these datasets with log-transformed Runtime, Energy, and Global Problem Size for GPR and AL with one and two controlled variables.

Figure 6.1: Visualization of subsets from the analyzed datasets.

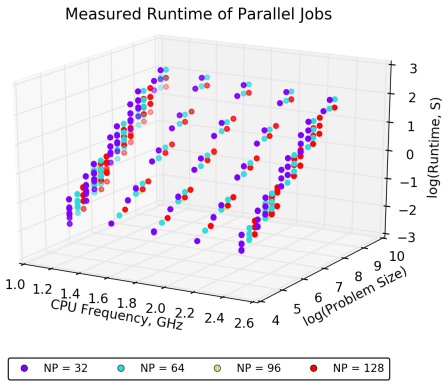


(a) Subset of jobs in *Performance*.

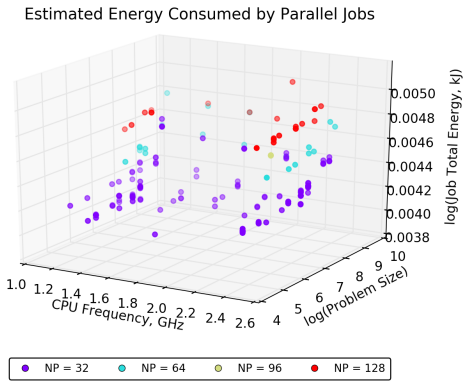


(b) Subset of jobs in *Power*.

Figure 6.2: Jobs from Fig. 6.1 shown with log-transformed Runtime and Energy.



(a) Subset of jobs in *Performance*.



(b) Subset of jobs in *Power*.

6.5.2 Regression Training and Active Learning

We aim to gain confidence about the reliability of the proposed AL algorithms by running our prototype on simple problems. We ensure that it works in 1D and 2D scenarios: the prototype receives subsets of the datasets where all but one and two controlled variables are fixed. Initially, we gain intuitions about the created one-variable GPR models by inspecting their visualizations (shown in Fig. 6.3). Then, we switch to the two-variable GPRs (depicted in Fig. 6.5) and describe the behavior of AL for pure variance reduction (without adjusting for the expected cost). An alternative strategy, which uses a cost efficient selection algorithm is developed and compared to the original strategy. Throughout this section we document the outlined progression and summarize our observations.

6.5.2.1 1D – Varying Problem Size

In 1D case, we choose to model how Runtime and Energy respond to changes in the Global Problem Size. We fix the rest of the variables (NP=32, Freq=2.4, Operator='poisson1'), and use GPR to produce a regression with associated prediction mean and variance as functions of the problem size.

Fig. 6.3(a) shows four GPRs with different values of two hyperparameters, the length scale l and the amplitude σ_f, which define the shape of the squared exponential defined in (6.11). The difference between the predictive mean curves is negligible. In contrast, the 95% confidence intervals (shown as: mean $\pm$ 2*standard deviation) are greatly impacted by the value of l: a decrease of l leads to a significant increase in the uncertainty, characterized by the standard deviation of predictions, in areas between measurement points.

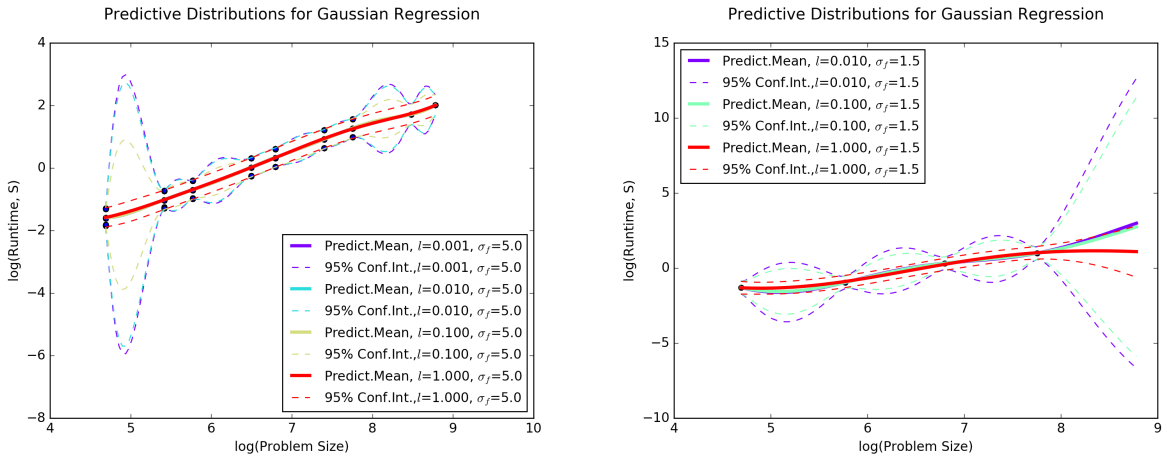
In Fig. 6.3(b), we can see that the growth of uncertainty can be exaggerated at the edge of the domain if there is no measurement nearby. "Clamped" at the existing points, the predictive distribution is so wide at the maximum problem size that we observe the difference not only for the confidence intervals but also for the predictive means with different hyperparamters. In choosing

values of hyperparameters to best fit the datasets, we rely on the scikit-learn's implementation which runs gradient ascent on the LML as described at [15]. This implementation finds hyperparameter values from a domain with specified boundaries and, in order to increase reliability, repeats this search multiple times, each time starting from a random point.

It is worth noting that when GPR is created using a small number of points, it may be erroneously assumed that the measurements are noise-free because the dataset does not include multiple y values for the same x. The produced GPR will optimistically consider its predictions to be exact, and the selected noise level may approach the machine precision. Further discussion of this point is provided in Section 6.5.2.4.

In general, GPRs constructed on datasets with many points should be more reliable. As described in [79], LML becomes more peaked with the growth of the dataset size. We can see that finding the peak in the landscape shown in Fig. 6.4, which corresponds to the Performance subset from Fig. 6.3(a), is a straightforward optimization problem with a unique global optimum. This peak can be found using gradient ascend with a single randomly selected starting point.

Figure 6.3: Predictive distribution for 1D cross section of *Performance* dataset.



(a) Random Subset of 4 Points.

(b) Subset of jobs in *Power*.

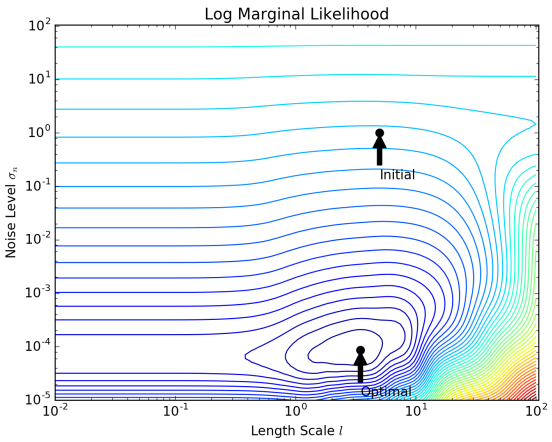


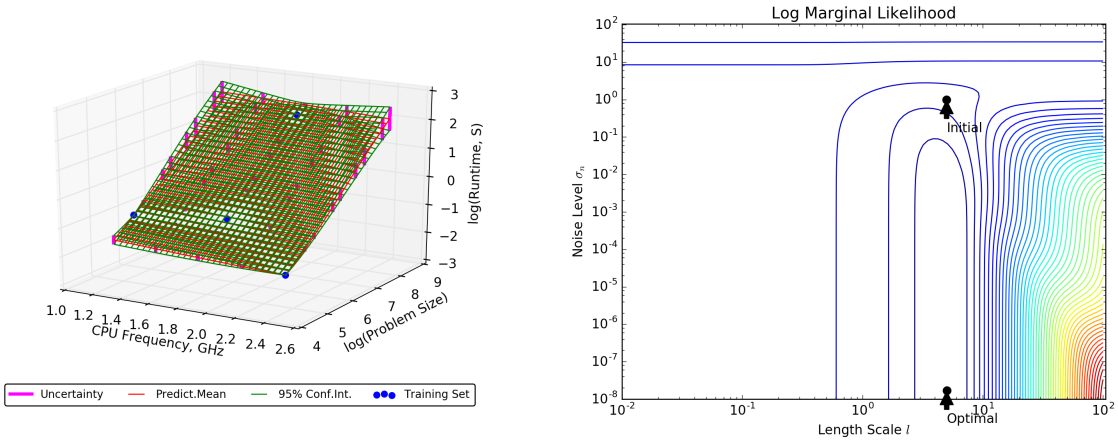
Figure 6.4: Contour plot of LML as a function of hyperparameters l and σ_n.

6.5.2.2 2D – Varying Problem Size and Frequency

Instead of curves, we obtain GPR surfaces when we vary two factors. For each GPR solution, defined by a set of specific hyperparameters, we obtain 3 surfaces: the high bound for the confidence interval, the predictive mean, and the low bound for the confidence interval. Fig. 6.5(a) shows the bounds with green wireframes and the mean with the red wireframe between the two bounds. In this particular case, the four randomly selected training points are aligned with each other well enough to define a tight predictive distribution. Nowhere in this space the confidence interval is particularly wide. However, further away from the training points, e.g., where both Frequency and Problem Size are near their maximum values, the confidence interval bounds are further apart. These are the areas where AL should select candidates for subsequent experiments. All experiment candidates are depicted on this plot with vertical magenta lines connecting the confidence interval surfaces.

Fig. 6.5(b) depicts LML for this GPR. Comparing to the aforementioned landscape for the case with abundant data, this one is significantly more shallow. However, the identified peak yields a GPR model which behaves as expected: it has a reasonable predictive distribution, and, we are confident that its standard deviation can be used as a selection criterion in AL.

Figure 6.5: GPR for a small dataset with two controlled variables.



(a) Regression trained on a small dataset.

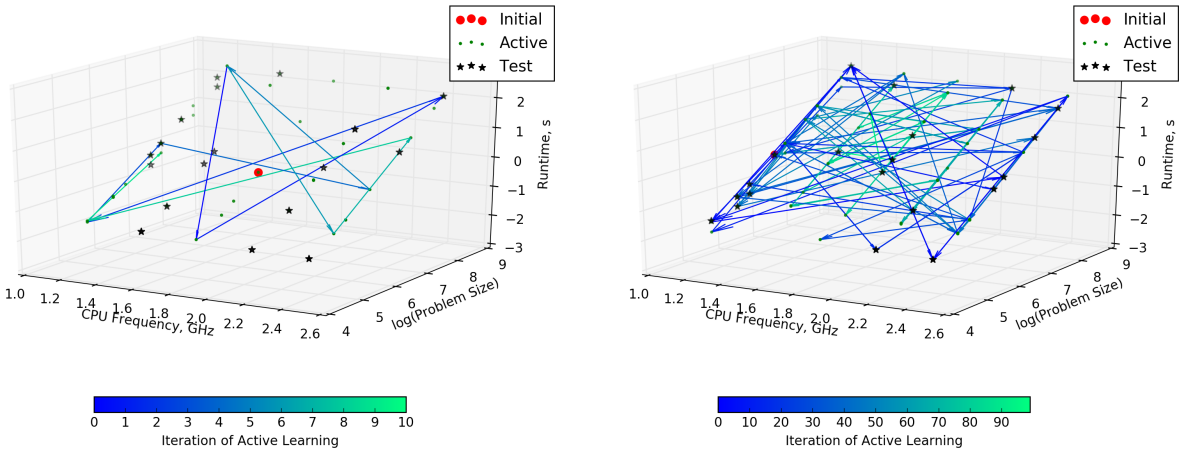
(b) Contour plot of a shallow LML function.

6.5.2.3 Monitoring Active Learning Progress

When we implement an Active Learning algorithm it is important to confirm that it indeed suggests an "interesting" sequence of experiments for learning. In other words, we need to monitor the steps taken by the algorithm and, at least in simple cases, compare its behavior with our expectations. In more difficult cases, e.g., when the number of varied variables is greater than two and we cannot easily visualize AL trajectories, we need to track metrics that characterize the learning progress. Such metrics will be used in making practical decisions such as whether to keep AL running or terminate it because the desired prediction accuracy is achieved.

We start by visualizing how AL chooses points in the 2D input space where Problem Size and Frequency are varied. Unlike the previous example, we consider a much larger dataset: a subset of points from the Performance dataset for which NP = 32 and Operator='poisson1'. This selection yields 251 jobs, which we randomly split into the Initial, Active, and Test sets. Fig. 6.6 depicts these sets and visualizes how AL suggests exploring this domain for 10 and 100 iterations. Each transition from one point to another in these sequences is shown with a colored arrow, starting with the blue-colored and progressing to the green-colored arrows. We can see that initially points from

Figure 6.6: AL with Variance Reduction.



(a) 10 Iterations.

(b) 100 Iterations.

the middle of the domain are not selected. In a star-like pattern, AL chooses experiments at the edges and, only after exhausting all edge points, progresses toward the middle. This is exactly the type of exploration that is intuitively employed by human experimenters, typically when there is no understanding of the relationships captured in the studied data, and often described in performance studies.

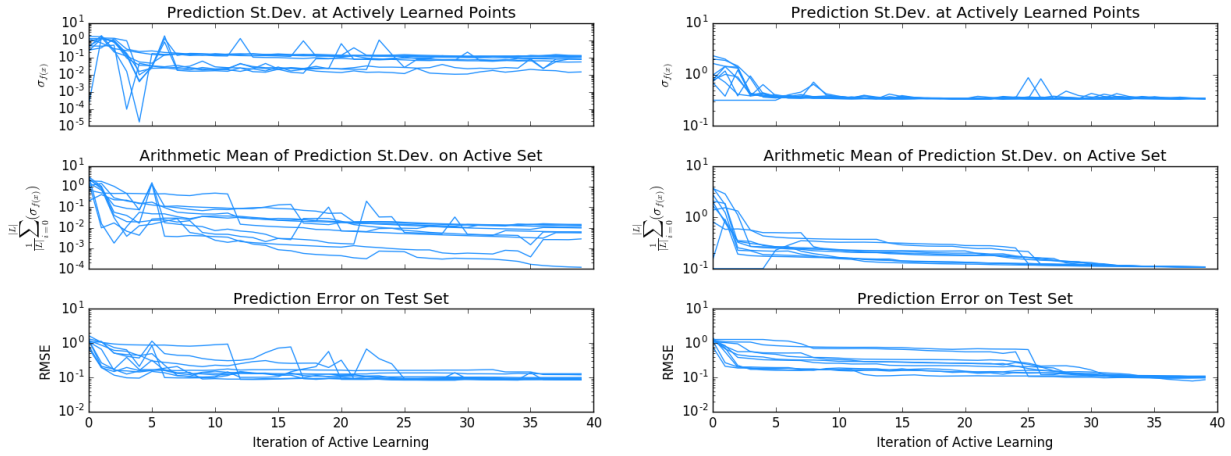
6.5.2.4 Interpretation of Results and Discussion

To understand AL's progress, we monitor 3 selected quantities as follows:

- $\sigma_{f(x)}$ – Standard Deviation (SD) of the prediction distribution at selected AL candidates (should not be confused with σ_f, one of the GPR hyperparameters). In the AL algorithm discussed so far, these are the highest values found across all candidate points.

- $\frac{1}{|L|}\sum_{i=0}^{|L|}\sigma_{f(x)}$ – Arithmetic Mean of the Standard Deviation (AMSD) calculated across all points in the Active Set. Our initial analysis shows that a geometric mean can be used as an alternative metric, but it does not provide any significant advantages.

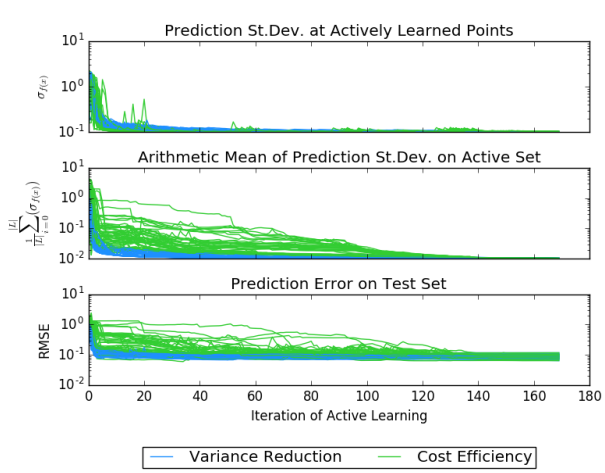
Figure 6.7: Strong influence of the limit on the noise-level σ_n on the quality of AL. The increased limit helps eliminate the overfitting problem.



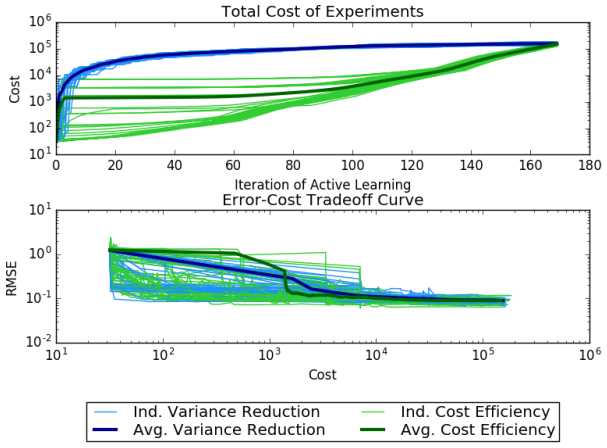
(a) $\sigma_n \geq 10^{-8}$

(b) $\sigma_n \geq 10^{-1}$

Figure 6.8: Comparing AL strategies: Variance Reduction and Cost Efficiency.



(a) Reduction of Error and Uncertainty.



(b) Growth of Cumulative Cost and Cost-Error Tradeoff Curves.

- RMSE: Aggregate error on the Test set defined by (6.2).

All three metrics are evaluated for 10 repetitions of AL for different random partitions of the same Performance subset, and the results are depicted in Fig. 6.7(a). We immediately notice

that all three quantities seem to converge to their stable values after about 25 iterations (with small subsequent variations). However, we consider inadequate the behavior where $\sigma_{f(x)}$ drops to negligible values before the 5th iteration. Also, AMSD in many of the shown trajectories decreases significantly below its stable value of approximately 10^{-2}. We explain this effect by the model overfitting: from the GPR's perspective, a regression with a very small variance is produced. This happens because with fewer than 5 points the measurements often align well and the mean and the confidence interval bounds are close to each other. Even with more points, it is still possible to obtain random partitions containing no outliers or particularly skewed measurements, which can results in non-optimal AL decisions.

To eliminate the overfitting issue, we restrict the search space for the hyperparameter σ_n in GPR. Specifically, we increase the lowest bound for the values which can be considered in gradient ascent. In contrast with Fig. 6.7(a), which shows properties of GPR with $\sigma_n \geq 10^{-8}$, we generate 10 new trajectories with this limit increased to 10^{-1} and show them in Fig. 6.7(b). The new trajectories do not demonstrate the aforementioned downsides. In this setting, AMSD becomes an important practical measure: when it converges (i.e. the average does not change significantly with additional AL iterations), AL can be terminated. The plots confirm that at that point RMSE will also converge to its stable value, and subsequent experiments may be considered excessive.

While the described fixed limit for σ_n results in the desired behavior in our experiments, we believe that a more general solution should involve a limit that dynamically adjusts. For instance, we expect that the restriction: $\sigma_n \geq 1/\sqrt{(N)}$, where N is the iteration counter, is a viable choice. In future work, we will investigate such limits and their implications.

AL which selects points with the highest SD is the approach which we refer to as Variance Reduction. Below we discuss our experience with an alternative cost-aware AL approach. In the cost-aware algorithm, rather than minimizing the variance we attempt to minimize the variance/cost ratio. While it may not be entirely clear how to define the cost in many other application domains, in performance analysis and computational science, certainly, longer experiments are more "expensive". Therefore, Runtime and Total Energy, the same quantities we are predicting, can be

interpreted as cost approximations, as well as different measures of the resource utilization.

Since we work primarily with the log-transformed responses, the proposed variance/cost ratio takes the form:

$$x^* = \underset{x \in pool}{\arg\max} \left(\sigma_{f(x)} - \mu_{f(x)} \right), \tag{6.14}$$

where $\mu_{f(x)}$ and $\sigma_{f(x)}$ are the mean and the SD of the predictive distribution at point x. This selection criterion defines the AL algorithm which we refer to as Cost Efficiency. We expect that this algorithm will be less aggressive about SD minimization and lean toward smaller experiments rather than larger ones where such choice is appropriate.

We integrated the described algorithm into our prototype and performed the initial analysis. We ran both Variance Reduction and Cost Efficiency on 50 random partitions of the Performance subset shown in Fig. 6.6(b). Fig. 6.8(a) illustrates different trajectories of individual learning sequences. As expected, RMSE for Cost Efficiency does not converge as quickly as for Reduction Variance, neither does AMSD (we again used $\sigma_n \geq 10^{-1}$), but both quantities converge after approximately the same number of iterations. However, the cost for these algorithms, shown in Fig. 6.8(b), confirms that Cost Efficiency can be the algorithm of choice in cost-sensitive and cost-limited studies. The tradeoff curves shown in the same figure are meant to act as a practical tool for selecting one algorithm over the other. In other words, it can help an experimenter who relies on AL suggestions to see how aggressive the available selection algorithms can minimize the error for a given cost. What we infer from the intersecting curves is that Cost Efficiency initially selects several points that on average do not efficiently reduce the error. In the longer term, by learning from a larger set of smaller experiments as opposed to a smaller set of larger experiments, Cost Efficiency becomes the superior algorithm (lower cost and lower error), at least for a subrange of cost values. In the shown example, the curves intersect at the value of the cumulative cost of $C = 1626$ (total compute time in seconds * number of cores). After that point, Cost Efficiency outperforms Variance Reduction, where the relative difference between the two algorithms for any given cost reaches up to 38%. White it is difficult to see it on the log-scaled plot, this advantage is

also significant at many other points: at $2 * C$, $3 * C$, $5 * C$, and $10 * C$ it is 25%, 21%, 16%, and 13%, respectively. Eventually, the curves meet at the maximum cost, the point where all available experiments are used to create GPR models.

6.6 Summary

This chapter has shown how Active Learning can be effectively applied to regression problems in performance analysis. By combining AL and GPR, we have created a method that allows us to get high-confidence predictions across a large input space without the need for a static, and possibly inefficient, experiment design. This means that performance regression against expensive computations or with larger parameter sets, become more feasible.

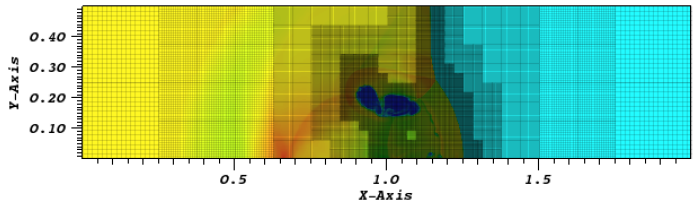
Chapter 7

Applying Active Learning to Adaptive Mesh Refinement Simulations

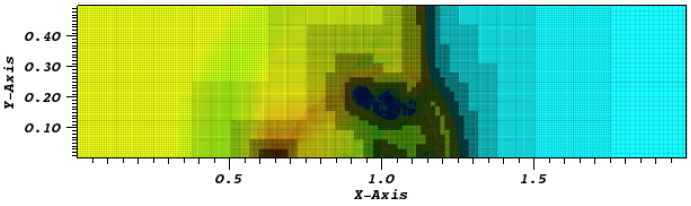
Adaptive Mesh Refinement (AMR) which was first introduced in the 1980s, is a popular technique in many areas of modern science and engineering. AMR refers to the methods which increase the resolution of simulation in the regions of studied domains where it is most needed. It maintains computational grids, so-called meshes, coarse in the majority of a given domain, if possible, and refines only selected regions by covering them with finer meshes. AMR is designed to significantly reduce computational and storage requirements comparing to the methods that obtain solutions on uniform high-resolution meshes.

In AMR computations, it is difficult to predict how much refinement will be applied in a particular simulation. User-defined criteria for refinement at the mesh point level may result in short, medium, or long, including extremely long, simulations for the entire studied domain. Even experienced users are likely to struggle with choosing the sufficient amount of computational resources for experiments with such widely varying costs. As shown in Fig. 7.1, allowing one more level of refinement in shock-bubble simulations with AMR reveals fine features of the studying phenomenon at the expense of a 3.8x increase in the simulation time. It is certain that this growth factor will be different when we add yet another refinement level, but it is impossible to predict its magnitude without doing model fitting and extrapolation. Moreover, when we vary one of many physical parameters – in shock-bubble interactions: viscosity, size and density of the bubble, change of pressure at the shock front, among others – we also observe drastic, unpredictable changes in performance characteristics.

Figure 7.1: Visualization of a 2D shock-bubble interaction simulated using ForestClaw package. Switching from (a) to (b) allows to resolve finer features but increases the simulation wall clock time by the factor of 3.8.



(a) AMR simulation with up to 5 levels of mesh refinement.



(b) AMR simulation with up to 6 levels of mesh refinement.

Fig. 7.2 shows the level of detail we can observe in simulations with high degree of refinement. Such simulations typically have high memory requirements and can take a significant amount of time even on supercomputers and computing clusters. Such libraries as MPI [47] and OpenMP [37] power many AMR libraries and packages by accelerating these computations via the distributed-memory and the shared-memory parallelism, respectively. However, as the number of machines simultaneously used for these computations increases, the cost of these computational experiments, typically expressed in node-hours or core-hours, also grows. For large real-world applications, the total cost of a desired series of AMR simulations may exceed even fairly large allocations on supercomputers and be impractical. In our modeling and the proposed design-of-experiment analysis, we demonstrate how the magnitude and the variability of performance characteristics can be modeled and describe the practical approach to selecting the most informative and cost-efficient simulations. Our approach is aimed to deprioritize overly expensive computations if the studied characteristics can be predicted with sufficient accuracy. This optimization is vital in applications with modest time-to-solution and total compute budget constraints.

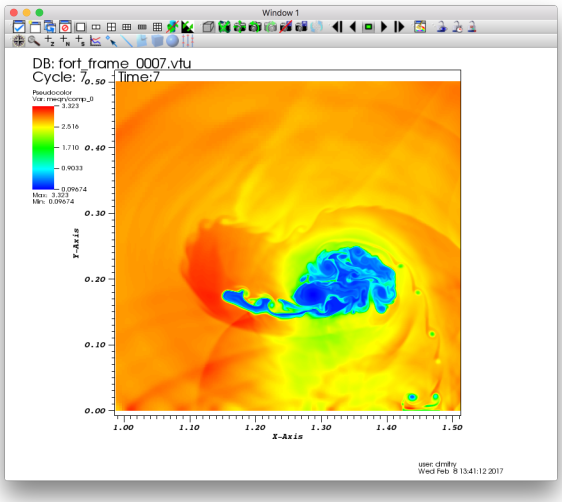


Figure 7.2: Visualization of the fine features of a shock-bubble interaction produced using VisIt. This simulation (selected parameters: mx = 16, maxlevel = 6) ran for 783.5 seconds on a single 24-core Intel Ivy Bridge node on Edison supercomputer at NERSC.

We extend the work described in the previous chapter and [40]. We use a combination of Gaussian Process Regression (GPR) [79] and Active Learning (AL) [89] as a method that allows us to obtain high-confidence predictions across a large input space without the need for a static, and most likely inefficient, experiment design. In this chapter, we describe how we apply this approach and the developed tools to a more challenging problem than the one studied previously (i.e. performance modeling for HPGMG benchmark [16]). We switch from GPR modeling with 2 parameters (we treated the problem size and the CPU frequency as input parameters) to the modeling with 5 parameters. The new input parameters, so-called features, include the following: the number of used processors, discretization box size, maximum level of refinement, bubble size, and bubble density (described in detail in Section 7.4). The last two are physical properties of the simulated shock-bubble interaction phenomenon, whereas the rest of them represent numerical and machine properties. In contrast with the modeling for a smooth function that we performed previously, we analyze the cost function that exhibits rapid, unpredictable growth along multiple dimensions in the aforementioned input space, and this growth occurs only at certain combinations of physical parameters, whereas the rest of the domain may appear relatively flat. With the cost

efficient AL algorithm presented in the previous chapter, we aim to perform sampling that is aware of the growth areas and cautiously selects candidate experiments near these areas. We expect the computational savings provided by this algorithm to be larger than the savings described in Chapter 6.

The remainder of this chapter is organized as follows: In Section 7.1, we summarize the related work. Section 7.2 presents the mathematical constructs used for GPR and AL, and Section 7.3 describes the practical aspects of our implementation of these constructs. Section 7.4 presents our evaluation of several AL algorithms running on a dataset with AMR simulation runtimes and costs. We investigate the AL "progress" and characterize the tradeoffs associated with the competing algorithms. We conclude the chapter in Section 7.5.

7.1 Background and Related Work

7.1.1 Adaptive Mesh Refinement Studies

Active Learning, described and evaluated in Chapter 6, appears to be an attractive way to acquire performance results and do necessary modeling for AMR simulations. There exist several approaches to AMR, including the block-structured and the tree-based AMR (e.g., see [1]), as well as many software tools for AMR, such as the libraries and packages listed at [30]. In our analysis, we focus on ForestClaw [24], the block-structured adaptive finite volume library for solving hyperbolic partial differential equations on mapped, logically Cartesian meshes. To achieve scalability, it uses p4est [26], the state-of-the-art library for grid management based on forests of quadtrees or octrees.

In [64], the authors describe Clawpack, an ecosystem of library and application code for solving nonlinear hyperbolic partial differential equations with high-resolution finite volume methods based on Riemann solvers and limiters, and also its components such as AMRClaw and ForestClaw. The performance and scalability of these and alternative solutions are investigated in many studies, including [24], [29], and [25]. To our knowledge, none of the existing studies considers running AMR simulations using Active Learning techniques.

7.1.2 Gaussian Processes

In addition to the studies referenced in Chapter 6, below we mention the relevant developments in the area of Gaussian Processes. In [52], the authors describe the Bayesian treed Gaussian processes where the treed partitioning algorithms help overcome the challenges associated with the stationarity of GPR (i.e. the fitting with the same covariance structure applied to the the entire input space) and its growing computational complexity. Similarly, [76] describes a method for accelerating GPR training by partitioning the training data in local regions and learning independent local Gaussian models inspired by the locally weighted projection regressions. The primary goal of the analysis in [23] is to provide insights into the optimal fitting for heteroskedastic datasets that exhibit input-dependent noise. In [74], the authors study the construction of the the models that are likely to yield sparse solutions and investigate fast GPR training algorithms. In all these studies, the computational cost is considered from the perspective of an individual fitting process. In contrast, we aim to improve the efficiency of AL processes where GPR is a building block rather than the end goal. Therefore, the aforementioned studies demonstrate the compatible optimizations and tools which will be beneficial in situations where our analysis tools are unable to process the studied datasets quickly enough. We will consider leveraging such tools in our future work.

7.2 Sequential Experimentation with Active Learning

To apply AL to the regression analysis for the AMR simulation cost, we use a process that produces estimates for both the mean and the standard deviation of the predictive distribution at each point in the selected input space. We use GPR, also referred to as kriging, for this process since it provides the desired information about the predictive posterior probability distribution. Below we summarize the mathematical constructs used in this modeling.

Let X and y be the design matrix (where the columns are the vectors of input values) and the vector of the response measurements, respectively. We aim to find the underlying function $f(x)$

which best fits the measurements with Gaussian noise:

$$y = f(X) + \mathcal{N}\left(0, \sigma_n^2\right), \tag{7.1}$$

where σ_n^2 – noise variance. For an arbitrary vector $x_\star$, the posterior distribution for individual predictions $f(x_\star)$ takes form of a multivariate Gaussian distribution:

$$p(f(x_\star)|x_\star, X, y) \sim \mathcal{N}\left(\mu_\star, \sigma^2 x_\star\right), \tag{7.2}$$

with

$$\mu_\star = k_\star^\mathsf{T} K_y^{-1} y, \sigma_\star^2 = k_{\star\star} - k_\star^\mathsf{T} K_y^{-1} k_\star, \text{ and } K_y = K + \sigma_n^2 I. \tag{7.3}$$

The covariance function $k(x_p, x_q)$ plays an important role in the parameter estimation in the following way. A covariance matrix:

$$[K]_{ij} = k(x_i, x_j), \text{ for all columns } x_i \text{ and } x_j \text{ in } X, \tag{7.4}$$

a covariance vector:

$$[k_\star]_i = k(x_\star, x_i), \text{ for all columns } x_i \text{ in } X, \tag{7.5}$$

and a scalar,

$$k_{\star\star} = k(x_\star, x_\star) \tag{7.6}$$

are calculated using the selected function $k(x_p, x_q)$.

Our choice of the covariance function is motivated by [78] and [79], where the authors describe the squared exponential (also referred to as the radial basis function or RBF) as a common solution:

$$k(x_p, x_q) = \sigma_f^2 \mathsf{exp}\left(-\frac{|x_p - x_q|^2}{2l^2}\right), \tag{7.7}$$

where the length scale l, the amplitude σ_f^2, and the noise level σ_n^2 are so-called hyperparameters. The hyperparameters need to be fitted, i.e. given appropriate values such that the resulting regression best matches the given dataset (X, y). As an alternative to RBF, the authors in [52] and [86] consider the Matérn covariance function family and argue about its advantages, including the

controllable smoothness. In our future work, we plan to reproduce the evaluation analysis described in Section 7.4 with the Matérn covariance functions.

According to the Bayesian inference with the marginal likelihood, the following quantity, referred to as the log marginal likelihood(LML),

$$\log p(y|X, l, \sigma_f^2, \sigma_n^2) = -\frac{1}{2}\left(y^{\mathsf{T}} K_y^{-1} y + \log|K_y|\right) + \mathsf{C}, \tag{7.8}$$

is minimized with respect to the selected hyperparameters:

$$(l, \sigma_f^2, \sigma_n^2) = \underset{l,\sigma_f^2,\sigma_n^2}{\arg\min}(\log p(y|X, l, \sigma_f^2, \sigma_n^2)). \tag{7.9}$$

After solving this optimization problem and fitting the hyperparameters, we can use the constructed GPR model to obtain the mean and the standard diviation of predictions for every input point x. According to AL algorithms, we then use these quantities to choose x for which the standard deviation of predictions is the highest, with or without adjusting for the corresponding cost.

7.3 Implementation and Collected Data

Similar to the prototype implementation described in Chapter 6, we perform our AL analysis in Python using the tools for Machine Learning and data analysis from the package called scikit-learn [14]. Specifically, we use the code for Gaussian Processes [15] from the latest stable version 0.18.1 (in comparison to 0.17 and older versions, this version provides the completely revised Gaussian Process module and supports kernel engineering, among other features).

To drive the developed AL system, we partition the dataset, which we describe in detail further in the current section, into 3 subsets: Initial (for initial regression training), Active (for one-at-a-time experiment selection with AL), and Test (for prediction quality analysis). For each run of AL, we use random partitioning with selected ratios between these sets. Thus, we choose the 1%:50%:49% ratios for the Initial, Active, and Test subsets, respectively, which yield the following absolute numbers of experiments: 11, 595, and 583. 11 initial experiments correspond to realistic scenarios where an application is first evaluated using a small batch of runs to verify the correctness

and gauge the magnitude and the variance of performance characteristics. The Active and Test sets in our analysis split the remaining experiments roughly with the 1:1 ratio.

After fitting the model on the initial set, our AL system repeats the following cycle: suggest a point for the next experiment, obtain the desired measurement for that experiment, and retrain the model with this measurement This process guides the sequential exploration of the input space in an unpredictable yet efficient manner. Currently, the developed prototype runs in the "offline" mode and consults a database of precomputed performance samples. In future work, we will investigate the "online" mode, the target use case where the modeling runs alongside the job scheduler and helps suggest computations in real time.

In order to study the efficiency of the developed AL algorithms, we ran a large collection of shock-bubble situations with different parameters using the ForestClaw package. We ran over 1190 computational jobs on the supercomputer called Edison [75] at NERSC (the US National Energy Research Computing Center). These jobs were organized into batches of up to 100 jobs (in order to satisfy the queueing limit) and scheduled to run on Edison's compute nodes (two-socket 12-core Intel "Ivy Bridge" processors running at 2.4 GHz and interconnected with the Cray Aries network with the Dragonfly topology running at 23.7 TB/s global bandwidth) using the supercomputer's scheduler. After completion, we collected all relevant information, including ForestClaw output, error logs, and scheduler accounting information, and transferred it from the supercomputer to a local workstation in order to analyze it using the developed AL evaluation prototype.

Table 7.1 provides details of the collected dataset. It is worth mentioning that the intervals in which the responses and features change are relatively large, allowing us to explore the behavior of the application performance across a domain where the growth along some of the dimensions is significant. For instance, the computational cost, expressed in node-hours, of the most expensive experiment exceeds the cost of the least expensive experiment by the factor of 6.2e3. It is also worth noting that these 1190 simulations represent a subset of the total 1920 possible combinations of the values selected for these 5 features. Among the analyzed 1190 simulations, 590 jobs uniquely represent their parameter combinations, whereas the remaining 600 jobs provide 2 to 3 repetitive

Table 7.1: The Parameters of the Collected Dataset with 1190 shock-bubble simulations.

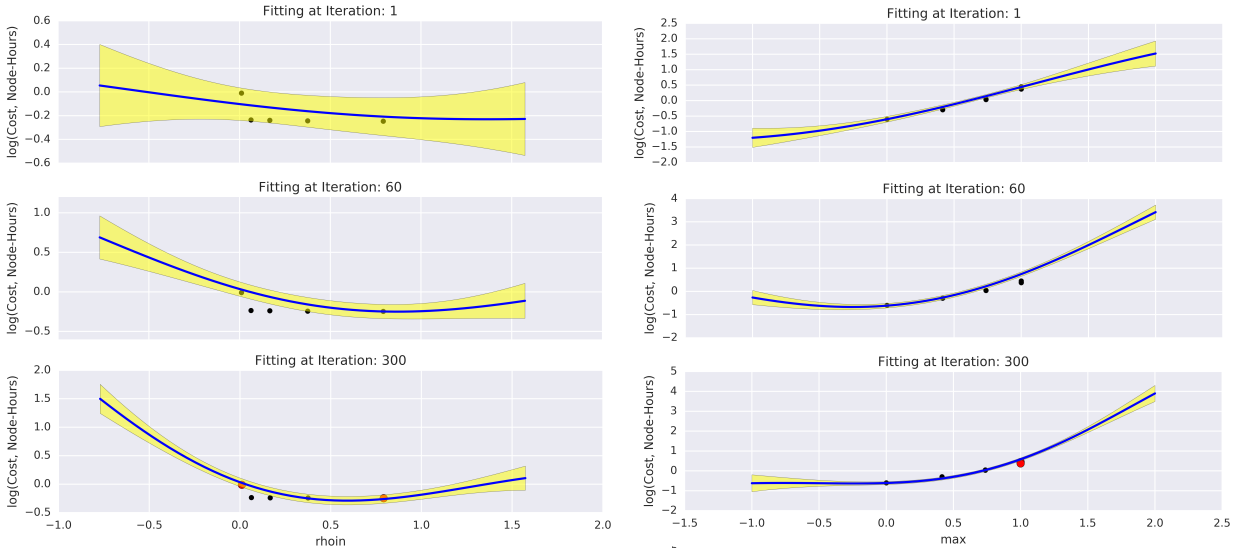
	min	median	mean	max
Feature: *p*, # of nodes	4	8	11.8	32
Feature: *mx*, box size	8	16	17.1	32
Feature: *max_level*, max refinement level	3	5	4.55	6
Feature: *r0*, bubble size	0.2 0	0.3 0	0.34	0.50
Feature: *rhoin*, bubble density	0.02	0.10	0.17	0.50
Response: *wall clock time*, seconds	1.77	39.90	137.61	4262.73
Response: *cost*, node-hours	0.0019	0.0947	0.4564	11.8527

measurements of performance for 292 combinations, capturing the machine performance variability. Therefore, 882 studied combinations amount up to 46% of the total 1920 combinations, which we expect to provide sufficient data to adequately model and predict performance within the studied domain. To collect this dataset, we used over 30K core-hours from our existing allocation at NERSC. So far, we used only Edison nodes, but we can apply the same analysis to the Knights Landing [93] machines (e.g., on the Cori supercomputer at NERSC) to demonstrate the cost-efficient exploration of the new hardware architecture in our future work.

7.4 Evaluation

We showed 1D and 2D GPR modeling in Chapter 6. Below we demonstrate how the GPR models for the 5-feature input space represent the data points that lay along several selected cross sections. Fig. 7.3 depicts how the fitting improves as the AL process progresses and more data becomes available to train the models. These cross sections are generated by fixing values for all features except for the selected ones, rhoin and max_level, and plugging ranges of numbers into the GPR-based predictors trained by the specific AL iteration counts. The black dots show all available data in the selected cross sections, whereas the red ones depict the experiments that have been included in the GPR fitting by the specific iterations labeled in the subfigure titles. From these plots, we can draw the following conclusion: for both features, 60 iterations provide enough data for GPRs to be trained on points that are not in these cross sections but lay nearby in the input

Figure 7.3: Gaussian Process Regression fittings that improve as the Active Learning progresses. With more iterations, more data becomes available for fitting GPR hyperparameters.



(a) Varying scaled bubble density.

(b) Varying scaled maximum level of refinement.

space and produce adequate models. The following iterations improve the curves but only slightly. Additionally, not only the predictive means (shown with blue lines) adjust to take the expected shapes, but also the confidence intervals (shown with yellow filled areas) shrink substantially as the iteration number increases. These experiments use the greedy AL selection algorithm which we refer to as Variance Reduction. In addition to studying individual AL "trajectories", we analyze batches of random partitions. The aggregate quantities such as the average prediction error and the average cumulative cost of experiments provide insights into how the AL process behaves independent of the initial state (i.e. the partition properties) and allows us to draw statistical conclusions described in the remainder of this section.

In this batch mode, we look at three different algorithms for guiding experiment selection: the aforementioned Variance Reduction algorithm, which seeks to run experiments that are expected to purely reduce prediction uncertainty, the Cost Efficiency algorithm – an alternative approach that weights experiments by the expected cost (in terms of node-hours) in order to learn the

most information using the least amount of compute cycles, and Random algorithm, which we treat as the baseline, reference implementation (without the intelligent AL selection, sequential experiments are likely to be selected in a random, suboptimal manner). A human experimenter will rely on his intuitions and most likely outperform this Random selection in terms of the amount of useful information obtained from sampling in simple modeling problems. However, when a fairly complex function is studied in many dimensions, such intuitions provide diminishing returns, and human-driven sampling efforts inevitably degrade to random. For the Cost Efficiency algorithm, we consider several variations where the degree with which the expected cost influences the candidate selection in this algorithm can be controlled using a tunable parameter which we refer to as cost weight. The candidate selection in this algorithm can be defined as:

$$x^* = \underset{x \in pool}{\arg\max} \left(\sigma_{f(x)} - C * \mu_{f(x)} \right), \tag{7.10}$$

where the cost weight C was set to 1.0 in the evaluation described in Chapter 6. Considering that such cost "penalty" appeared to be too strong in the previous work, we consider multiple values between 0.0625 and 1.0 in the current analysis.

We compare these algorithms using the output of our AL system produced for 50 repetitions for the selected algorithms, where each repetition represents a random partitioning of the dataset. Fig. 7.4 illustrates the progress accomplished by each algorithm on reducing the uncertainty and the errors associated with GPR predictions. The filled areas depict the interquartile ranges (IQRs) for progress characteristics quantifying AL convergence, whereas the thick lines depict the medians for the selected metrics (for heavily skewed data such as the shown distributions, the median is viewed as a more representative measure of central tendency than the mean). We use the same metrics as in the evaluation presented in Chapter 6: the standard deviation of the prediction distribution at selected AL candidates, the arithmetic mean of the standard deviation calculated across all points in the Active set, RMSE – the root mean squared error for all points in the Test set, and cumulative cost of learned experiments. These metrics are shown in Fig. 7.4 in the order from top to bottom. Additionally, the tradeoff curves shown in Fig. 7.5 are meant to act as a practical tool for selecting

one algorithm over the others. What we infer from these curves is that Cost Efficiency initially demonstrates the efficiency that is similar to the level shown by other algorithms. After some initial training, by learning from a larger set of smaller experiments as opposed to a smaller set of larger experiments, Cost Efficiency becomes the superior algorithm (lower error for the same cost) for the majority of the AL process. This conclusion is the same as the one we arrived at when described the modeling for the HPGMG-FE benchmark in the previous chapter.

In studying the described dataset and the AL algorithms, we have observed that performing anisotropic GPR modeling, which refers to the process that allows the GPR hyperparameters to change differently along different dimensions of the input space, is a desirable improvement. In contrast with the isotropic modeling, which treats the length scale l hyperparameter as a scalar (i.e. the same value for every dimension), the notion of proximity of points in the studied input space should be different along different dimensions as the selected features use the ranges of values that significantly differ in magnitude: as shown in Table 7.1, for instance, [4,32] for p is much larger than [0.02, 0.50] for $rhoin$. However, in our experiments with anisotropic modeling, we observed that treating l as a 5-component vector instead of a scalar results in a significant increase of the fitting cost. In Fig. 7.6 (a), we show how much time such fittings require in a sample AL process. All shown measurements add up to roughly 1174 seconds. In order to reduce these modeling costs, we can scale all features to the unit cube, which is a common practice, and perform an isotropic fitting. The reduced computational costs associated with the process of fitting scalar l values (along with the amplitude and the noise level) is shown in Fig. 7.6 (b). This reduction in the number of hyperparameters leads to the reduction of the total computational time down to 447 seconds (approx. 62% reduction). This optimization certainly needs to be considered and used along with other fitting optimizations, for instance, varying the number of retries in the fit optimizer, when AL needs to repeatedly run on large datasets.

7.5 Summary

The main contributions of the analysis presented in the current chapter include:

- We obtain additional evidence that supports our argument for using Active Learning in combination with Gaussian Process Regressions to optimize sequences of computer experiments. We evaluate the developed analysis techniques in the context of learning and predicting the computational experiments for Adaptive Mesh Refinement simulations.

- We show how in the studied 5-feature input space the Cost Efficient algorithm for Active Learning can outperform its alternatives and provide significant reductions of prediction errors for a wide range of given computational budgets.

- We discuss how we can reduce the cost associated with the GPR modeling by switching from the anisotropic fitting to the isotropic while preserving its predictive power.

- As shown in Fig. 7.5, Random AL sampling works well: its prediction errors are comparable to the optimal error levels provided by the Cost Efficiency algorithms with small cost weights (0.25, 0.125, and 0.0625). Random sampling is likely to perform well due to the fact that it is allowed to select points in the expensive areas unlike the rest of the sampling algorithms. These points provide valuable information from the modeling perspective, whereas other algorithms, specifically Cost Efficiency, may treat high-cost measurements as rare outliers and never sample enough in the nearby areas. In the dataset that we study in this chapter, the number of expensive experiments is rather small, and therefore such experiments are easily under-sampled. In our future work, we plan to repeat the described evaluation on the subsets of this dataset with much higher ratios of expensive experiments. In such scenarios, AL with the Cost Efficient algorithm will not be able to focus purely on the cheap experiments, the ones that also provide minimal information for improving the modeling, and will be forced to navigate around fairly expensive experiments in an optimal way. In this synthetic analysis, we aim to gain additional valuable insights into the performance of AL algorithms and better understand the tradeoffs that we will likely observe when we proceed to running Active Learning for "online" performance analysis in future work.

Figure 7.4: Comparing Active Learning algorithms: Variance Reduction, Random, and several variations of Cost Efficiency with different cost weights.

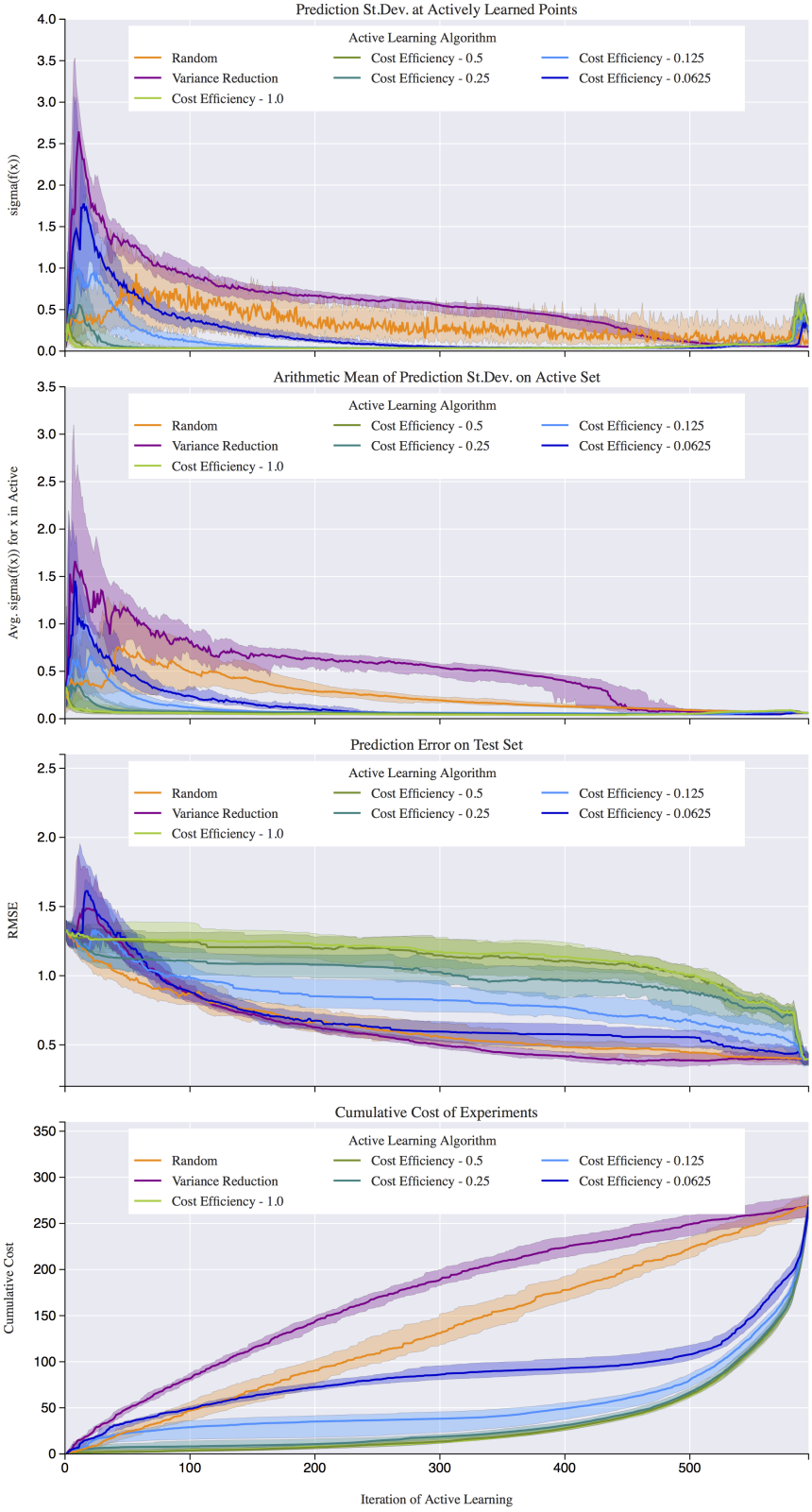


Figure 7.5: Cost-Error tradeoff curves for the developed Active Learning algorithms.

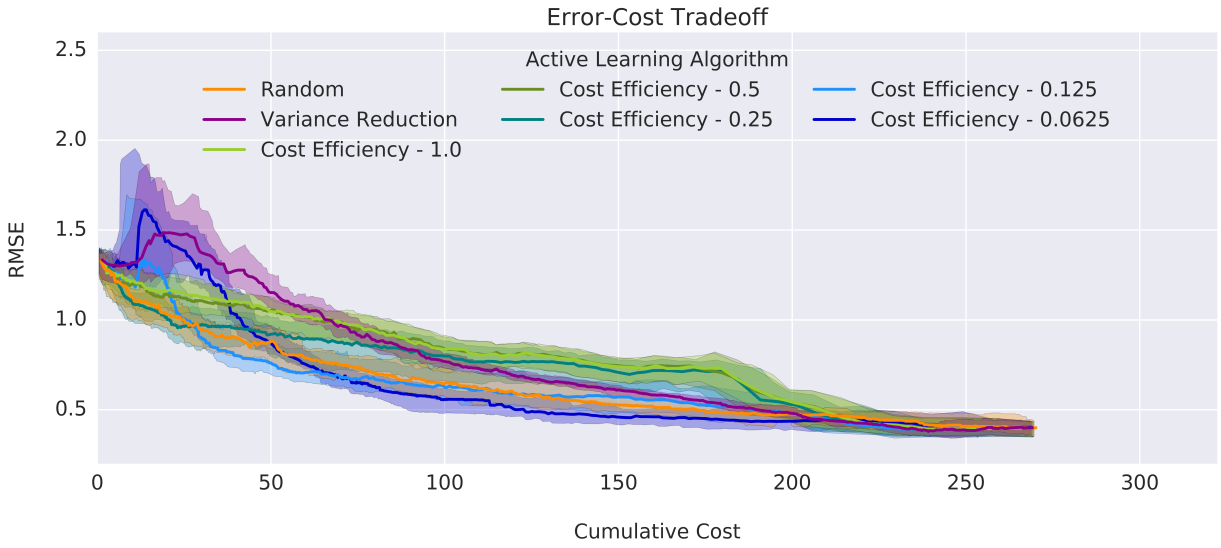
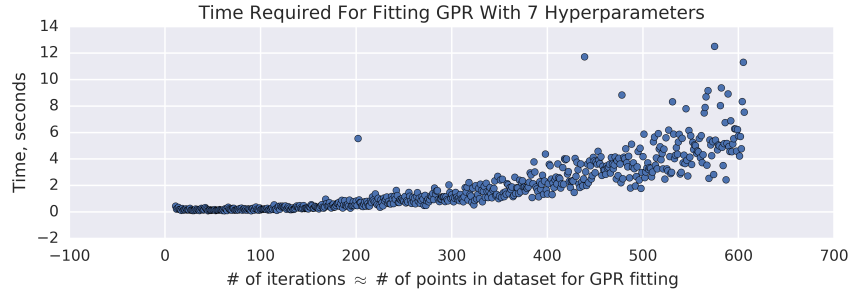
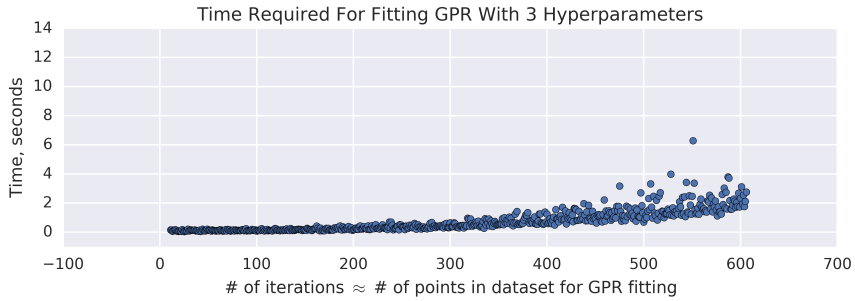


Figure 7.6: Evaluating computational complexity of GPR fitting.



(a) Anisotropic fitting.



(b) Isotropic fitting.

Chapter 8

Conclusions and Future Work

8.1 Key Contributions

To address the challenges observed in research and experimentation that involve High Performance and Cloud Computing systems, we design and evaluate several system architecture and experimentation approaches as summarized below.

- **On rebalancing in multi-cloud environments:** We design a multi-cloud environment where virtual machines are distributed across multiple cloud infrastructures and work collaboratively to process High Throughput Computing workloads. We evaluate several rebalancing policies that guide these deployments towards desired configurations while having minimal negative impacts on the workloads.

- **On enabling balanced elasticity between computing environments:** We describe our analysis of interactions between portions of a shared cyberinfrastructure managed by diverse computing frameworks and present a novel architecture for resource sharing and optimization of resource usage. Using the developed discrete even simulator, we collect information that allows us to make reliable conclusions about the efficiency of the elasticity policies that can be employed to manage these interactions on a prototype implementation and on production deployments.

- **On designing a highly available configuration management systems:** We design an architecture for a highly available configuration management system that runs simulta-

neously on a local system and in a remote cloud. We deploy a prototype that uses Chef, a popular modern configuration management system, where we aim to preserve the way this system is used and create no unnecessary adoption barriers.

- **On using configuration management systems to configure computing environments:** While continuing our analysis on configuration management, we demonstrate how configuration management systems can help orchestrate software environments deployed on resources in such multi-site cyberinfrastuctures as the CloudLab testbed. We show how Chef addresses customization issues encountered in the development and maintenance of complex software stacks designed for running on multiple hardware platforms.

- **On using Active Learning for performance analysis:** Proceeding beyond the provisioning and configuration of computing environments, we focus on the efficient performance analysis and benchmarking in such environments. We demonstrate how Active Learning can be used to empirically model and acquire new performance measurements. By combining Active Learning and Gaussian Process Regression, we create an efficient method that provides high-confidence predictions across large input spaces without the need for static, and possibly inefficient, experiment designs. This means that performance regressions against expensive computations or with larger parameter sets, become more feasible. Additionally, we apply Active Learning analysis to Adaptive Mesh Refinement simulations. By finding the Gaussian Process Regression fittings for a highly variable cost function in a 5-feature input space, we analyze a much more challenging problem than the one studies in our initial work described in Chapter 6. We show that the Cost Efficient algorithm for Active Learning in most cases also significantly outperforms its alternatives with respect to the prediction errors associated with a given compute budget. Tentatively, the material described in Chapter 7 will be used to prepare a paper submission for the IEEE Cluster 2017 conference.

8.2 Future Work

Chapters 3 and 7 document our most recent research and development efforts. Naturally, our primary future work ideas revolve around the possible improvements that can be applied to the technologies and methods used in elasticity managements and design of experiment using Active Learning.

In Chapter 3, we investigate interactions between portions of a shared cyberinfrastructure where elasticity polices can manage allocations and the resource sharing associated with such interactions. Experimenting with workload traces from various High Performance Computing centers and characterizing the behavior of the developed policies under the corresponding workloads constitute one of the directions in our future research. We will also consider the development of policies that take into account user-requested job wall clock times. However, we are skeptical of the accuracy of such estimates; e.g., for the workload from the Peregrine supercomputer deployed at the the National Renewable Energy Laboratory that we analyzed in Chapter 3, these estimates demonstrate extremely low accuracy: for over 50% of the jobs, the requested wall clock exceeds the actual wall clock by the factor of 19.2. While the accurate prediction of job runtimes is considered a difficult problem, and, to the best of our knowledge, there is no common practical solution employed in HPC environments for this purpose, we imagine that assigning heavier weights to the most recent jobs for each user, application, queue, etc., and predicting with acceptable accuracy whether the remaining runtimes are likely to exceed the Grace Period or not is feasible. Another direction to further improve the most efficient policy among the studied ones it to introduce a job prioritization that changes after jobs exceed their requested wall clock times. Similar to the scheduler configurations often used at HPC centers where such jobs are terminated after they exceed the requested time, another variation of the parallel-aware policy can deprioritize such jobs, and therefore increase the probability of preemption for the nodes on which they run. Finally, we will continue to experiment with the prototype and the developed simulator in order to obtain additional insights and gain confidence in the fact that the elasticity mechanisms are reliable and

intelligent before releasing them to the community of users on CloudLab and related testbeds.

In Chapters 6 and 7, we evaluate Active Learning techniques. We performed the experimentation in these studies "offline": our Active Learning system consulted a database of precomputed performance samples. In contrast, the target use case is "online" where the next experiment must be scheduled. In future work, we plan to run some experiments in parallel at every iteration of Active Learning, which will add additional scheduling concerns and software engineering complexity and may indicate less greedy selection strategies but will lead to the reduction of the total learning time. Additionally, realistic simulations often involve continuous or near-continuous parameters, such that the active sets cannot be treated as finite. We expect that this could be handled by choosing the best option within a finite subset or, preferably, by using continuous optimization techniques. Gradient-based methods, which are available with GPR, would provide an important benefit for problems with high-dimensional parameter spaces, such as the configurations analyzed for Adaptive Mesh Refinement simulations in Chapter 7. We expect that the proposed techniques can also be generalized to perform cost-aware adaptive reduced order modeling for application-relevant response surfaces. In a separate study, we plan to rigorously investigate computational requirements of competing GPR methods and the incremental fitting techniques described in the literature, as well as the developed AL algorithms, and consider possible optimizations that will enable Active Learning on large datasets.

Bibliography

[1] Adaptive mesh refinement (amr): Numerical methods and tools. http://www.training.prace-ri.eu/uploads/tx_pracetmo/AMRIntroHNDSCi15.pdf. Accessed: 2016-10-18.

[2] Auto scaling. https://aws.amazon.com/autoscaling/. Accessed: 2016-10-12.

[3] Boto: A python interface to amazon web services. http://boto.readthedocs.io/en/latest/. Accessed: 2016-10-12.

[4] Chef. https://docs.chef.io/. Accessed: 2016-10-13.

[5] Chpc - research computing support for the university. https://www.chpc.utah.edu/. Accessed: 2016-10-18.

[6] The cloudlab manual - hardware. http://docs.cloudlab.us/hardware.htm. Accessed: 2017-04-18.

[7] Condor log analyzer. http://condorlog.cse.nd.edu. Accessed: 2016-10-12.

[8] Elastic compute cloud (ec2) cloud server and hosting at aws. https://aws.amazon.com/ec2/. Accessed: 2016-10-12.

[9] Elasticslice - client code repository. https://gitlab.flux.utah.edu/elasticslice/elasticslice. Accessed: 2016-10-18.

[10] Elasticslice - documentation. http://www.emulab.net/downloads/elasticslice/. Accessed: 2016-10-18.

[11] Futuresystems. https://portal.futuresystems.org/about. Accessed: 2016-10-12.

[12] Htcondor. http://research.cs.wisc.edu/htcondor/. Accessed: 2016-10-12.

[13] Nimbus. http://www.nimbusproject.org. Accessed: 2016-10-12.

[14] scikit-learn - machine learning in python. http://scikit-learn.org. Accessed: 2017-04-18.

[15] scikit-learn-0.18.dev0 - gaussian processes. http://scikit-learn.org/dev/modules/gaussian_process.html. Accessed: 2017-04-18.

[16] Mark Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Sam Williams. HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems. May 2014.

[17] Mark Adams, Jed Brown, John Shalf, Brian van Straalen, Erich Strohmaier, and Sam Williams. HPGMG: High-performance geometric multigrid.

[18] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. Communications of the ACM, 53(4):50–58, 2010.

[19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. 2009.

[20] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2016.

[21] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. Journal of computational Physics, 53(3):484–512, 1984.

[22] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. Computer Networks, 61(0):5–23, March 2014.

[23] Mickael Binois, Robert B Gramacy, and Michael Ludkovski. Practical heteroskedastic gaussian process modeling for large simulation experiments. arXiv preprint arXiv:1611.05902, 2016.

[24] Carsten Burstedde, Donna Calhoun, Kyle Mandli, and Andy R Terrel. Forestclaw: Hybrid forest-of-octrees amr for hyperbolic conservation laws. arXiv preprint arXiv:1308.1472, 2013.

[25] Carsten Burstedde and Johannes Holke. Coarse mesh partitioning for tree based amr. arXiv preprint arXiv:1611.02929, 2016.

[26] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. SIAM Journal on Scientific Computing, 33(3):1103–1133, 2011.

[27] Rajkumar Buyya. High performance cluster computing: Architecture and systems, volume i. Prentice Hall, Upper SaddleRiver, NJ, USA, 1:999, 1999.

[28] W. Cai, Y. Zhang, and J. Zhou. Maximizing expected model change for active learning in regression. In 2013 IEEE 13th International Conference on Data Mining, pages 51–60, Dec 2013.

[29] Alan C Calder, Bruce C Curtis, LJ Dursi, Bruce Fryxell, P MacNeice, K Olson, P Ricker, R Rosner, FX Timmes, HM Tufo, et al. High performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In Proceedings of the 2000 ACM/IEEE conference on Supercomputing, page 56. IEEE Computer Society, 2000.

[30] Donna Calhoun. Adaptive mesh refinement resources. http://math.boisestate.edu/ calhoun/www_personal/research/amr_software/. Accessed: 2016-10-18.

[31] Chef. An overview of chef. https://docs.chef.io/chef_overview.html. Accessed: 2016-08-01.

[32] Trieu C Chieu, Ajay Mohindra, Alexei A Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on, pages 281–286. IEEE, 2009.

[33] CloudLab. Cloudlab portal. https://cloudlab.us/. Accessed: 2016-08-01.

[34] CloudLab. Flexible, scientific infrastructure for research on the future of cloud computing. https://cloudlab.us/. Accessed: 2017-04-18.

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

[36] CycleComputing. Hgst buys 70, 000-core cloud hpc cluster, breaks record, returns it 8 hours later. https://cyclecomputing.com/hgst-buys-70000-core-cloud-hpc-cluster-breaks-record-8-hours/. Accessed: 2016-10-11.

[37] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering, 5(1):46–55, 1998.

[38] Dmitry Duplyakin. Jupyter notebooks for experimental evaluation. http://dmitry.duplyakin.org/p/elastic/. Accessed: 2017-04-18.

[39] Dmitry Duplyakin. Performance and power dataset for active learning. http://dmitry.duplyakin.org/al-performance-and-power-dataset.html. Accessed: 2017-04-18.

[40] Dmitry Duplyakin, Jed Brown, and Robert Ricci. Active learning in performance analysis. In Proceedings of the IEEE Cluster Conference, September 2016.

[41] Dmitry Duplyakin, Matthew Haney, and Henry Tufo. Architecting a persistent and reliable configuration management system. In Proceedings of the 6th Workshop on Scientific Cloud Computing, pages 11–16. ACM, 2015.

[42] Dmitry Duplyakin, Paul Marshall, Kate Keahey, Henry Tufo, and Ali Alzabarah. Rebalancing in a multi-cloud environment. In Proceedings of the 4th ACM workshop on Scientific cloud computing, pages 21–28. ACM, 2013.

[43] Dmitry Duplyakin and Robert Ricci. Introducing configuration management capabilities into cloudlab experiments. In IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2016.

[44] Dmitry Duplyakin and Robert Ricci. Introducing configuration management capabilities into CloudLab experiments. In Proceedings of the International Workshop on Computer and Networking Experimental Research Using Testbeds (CNERT), April 2016.

[45] Emulab. Network emulation testbed home. https://www.emulab.net/. Accessed: 2017-04-18.

[46] Dror G Feitelson. The supercomputer industry in light of the top500 data. Computing in science & engineering, 7(1):42–47, 2005.

[47] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[48] Ian Foster and Carl Kesselman. The Grid 2: Blueprint for a new computing infrastructure. Elsevier, 2003.

[49] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In 2008 Grid Computing Environments Workshop, pages 1–10. Ieee, 2008.

[50] Guilherme Galante, Luis Carlos Erpen De Bona, Antonio Roberto Mury, Bruno Schulze, and Rodrigo Rosa Righi. An analysis of public clouds elasticity in the execution of scientific applications: A survey. J. Grid Comput., 14(2):193–216, June 2016.

[51] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In Cloud Computing (CLOUD), 2011 IEEE International Conference on, pages 716–723, July 2011.

[52] Robert B Gramacy and Herbert KH Lee. Adaptive design of supercomputer experiments. Technical report, Dept of Applied Math & Statistics, 2006.

[53] HTCondor. Computing with htcondor. `http://research.cs.wisc.edu/htcondor/`. Accessed: 2017-04-18.

[54] Deng Huang, TT Allen, WI Notz, and RA Miller. Sequential kriging optimization using multiple-fidelity evaluations. Structural and Multidisciplinary Optimization, 32(5):369–382, 2006.

[55] Jinho Hwang, Sai Zeng, Frederick y Wu, and Timothy Wood. A component-based performance comparison of four hypervisors. In Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on, pages 269–276. IEEE, 2013.

[56] R. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, 1990.

[57] Kate Keahey, Patrick Armstrong, John Bresnahan, David LaBissoniere, and Pierre Riteau. Infrastructure outsourcing in multi-cloud environment. In Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit, pages 33–38. ACM, 2012.

[58] Kate Keahey, Renato Figueiredo, Jos Fortes, Tim Freeman, and Maurcio Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. Cloud computing and applications, 2008:825–830, 2008.

[59] Jay Lepreau et al. Protogeni. In 1st GENI Engineering Conference, 2007.

[60] Shuo Liu, Gang Quan, and Shangping Ren. On-line scheduling of real-time services for cloud computing. In Services (SERVICES-1), 2010 6th World Congress on, pages 459–464. IEEE, 2010.

[61] Alejandro Lucero. Simulation of batch scheduling using real production-ready software tools. Proceedings of the 5th IBERGRID, 2011.

[62] Dylan Machovec, Cihan Tunc, Nirmal Kumbhare, Bhavesh Khemka, Ali Akoglu, Salim Hariri, and Howard Jay Siegel. Value-based resource management in high-performance computing systems. In Proceedings of the ACM 7th Workshop on Scientific Cloud Computing, pages 19–26. ACM, 2016.

[63] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. Future Generation Computer Systems, 48:1 – 18, 2015. Special Section: Business and Industry Specific Cloud.

[64] Kyle T Mandli, Aron J Ahmadia, Marsha Berger, Donna Calhoun, David L George, Yiannis Hadjimichael, David I Ketcheson, Grady I Lemoine, and Randall J LeVeque. Clawpack: building an open source ecosystem for solving hyperbolic pdes. PeerJ Computer Science, 2:e68, 2016.

[65] 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, pages 43–52. IEEE Computer Society, 2010.

[66] 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.

[67] Paul Marshall, Kate Keahey, and Tim Freeman. Improving utilization of infrastructure clouds. In Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on, pages 205–214. IEEE, 2011.

[68] 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. IEEE, 2012.

[69] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW '12, pages 1085–1094, Washington, DC, USA, 2012. IEEE Computer Society.

[70] Paul Marshall, Henry Tufo, Kate Keahey, David LaBissoniere, and Matthew Woitaszek. A Large-Scale Elastic Environment for Scientific Computing, pages 112–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[71] Paul Marshall, Henry M Tufo, Kate Keahey, David La Bissoniere, and Matthew Woitaszek. Architecting a large-scale elastic environment-recontextualization and adaptive cloud services for scientific computing. In ICSOFT, pages 409–418, 2012.

[72] M. Mattess, C. Vecchiola, and R. Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, pages 180–188, Sept 2010.

[73] Rick McGeer, Mark Berman, Chip Elliott, and Robert Ricci, editors. The GENI Book. Springer International Publishing, 2016.

[74] Franziska Meier, Philipp Hennig, and Stefan Schaal. Incremental local gaussian regression. In Advances in Neural Information Processing Systems, pages 972–980, 2014.

[75] NERSC. Edison. http://www.nersc.gov/users/computational-systems/edison/. Accessed: 2017-04-18.

[76] Duy Nguyen-Tuong, Matthias Seeger, and Jan Peters. Model learning with local gaussian process regression. Advanced Robotics, 23(15):2015–2034, 2009.

[77] OpenStack. Open source software for creating private and public clouds. https://www.openstack.org/. Accessed: 2017-04-18.

[78] E. Pasolli and F. Melgani. Gaussian process regression within an active learning scheme. In Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE International, pages 3574–3577, July 2011.

[79] Carl Edward Rasmussen and Christopher K. I. Williams. Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press, 2005.

[80] HPC Energy Research. https://cscdata.nrel.gov/#/datasets/d332818f-ef57-4189-ba1d-beea291886eb. Accessed: 2017-04-18.

[81] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. USENIX ;login:, 39(6), December 2014.

[82] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A platform for repeatable research in computer science. ACM SIGOPS Operating Systems Review, 49(1):100–107, 2015.

[83] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A platform for repeatable research in computer science. ACM SIGOPS Operating Systems Review, 49(1), January 2015.

[84] Gonzalo Pedro Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Hpc system lifetime story: Workload characterization and evolutionary analyses on nersc systems. In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, pages 57–60. ACM, 2015.

[85] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, CLOUD '11, pages 500–507, Washington, DC, USA, 2011. IEEE Computer Society.

[86] Thomas J Santner, Brian J Williams, and William I Notz. The design and analysis of computer experiments. Springer Science & Business Media, 2013.

[87] SchedMD. Slurm workload manager. https://slurm.schedmd.com/. Accessed: 2017-04-18.

[88] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In Proceedings of the International Conference on Dependable Systems and Networks, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[89] Burr Settles. Active learning literature survey. Computer Sciences Technical Report, 1648, 2010.

[90] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In Proceedings of the 17th International Middleware Conference, page 1. ACM, 2016.

[91] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In Proceedings of the 2Nd ACM Symposium on Cloud Computing, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[92] J. L. L. Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In High Performance Computing and Simulation (HPCS), 2011 International Conference on, pages 1–7, July 2011.

[93] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In Hot Chips 27 Symposium (HCS), 2015 IEEE, pages 1–24. IEEE, 2015.

[94] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure, page 29. ACM, 2015.

[95] Chameleon Team. Chameleon cloud: A configurable experimental environment for large-scale cloud research. `https://www.chameleoncloud.org/`. Accessed: 2017-04-18.

[96] Yi Wei and M Brian Blake. Proactive virtualized resource management for service workflows in the cloud. Computing, 98(5):523–538, 2016.

[97] Patrick J. Whitcomb and Mark J. Anderson. RSM Simplified: Optimizing Processes Using Response Surface Methods for Design of Experiments. Productivity Press, 2004.

[98] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI). USENIX, December 2002.

[99] Hao Wu, Shangping Ren, Steven Timm, Gabriele Garzoglio, and Seo-Young Noh. Experimental study of bidding strategies for scientific workflows using aws spot instances. In Proceedings of 8th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS). ACM, 2015.

[100] Keiji Yamamoto, Atsuya Uno, Hitoshi Murai, Toshiyuki Tsukamoto, Fumiyoshi Shoji, Shuji Matsui, Ryuichi Sekizawa, Fumichika Sueyasu, Hiroshi Uchiyama, Mitsuo Okamoto, Nobuo

Ohgushi, Katsutoshi Takashina, Daisuke Wakabayashi, Yuki Taguchi, and Mitsuo Yokokawa. The k computer operations: Experiences and statistics. Procedia Computer Science, 29:576 – 585, 2014.

[101] J. Yang, T. Yu, L. R. Jian, J. Qiu, and Y. Li. An extreme automation framework for scaling cloud applications. IBM Journal of Research and Development, 55(6):8:1–8:12, Nov 2011.

[102] Katherine Yelick, Susan Coghlan, Brent Draney, Richard Shane Canon, et al. The magellan report on cloud computing for science. US Department of Energy, Washington DC, USA, Tech. Rep, 2011.

[103] Andrew J. Younge, Robert Henschel, James T. Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C. Fox. Analysis of virtualization technologies for high performance computing environments. In Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, CLOUD '11, pages 9–16, Washington, DC, USA, 2011. IEEE Computer Society.