Resource Allocation and Scheduling in Heterogeneous Cloud Environments

Gunho Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-78
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-78.html

May 10, 2012

Resource Allocation and Scheduling in Heterogeneous Cloud Environments

by

Gunho Lee

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Ion Stoica
Professor Ray R. Larson

Spring 2012

Resource Allocation and Scheduling in Heterogeneous Cloud Environments

by

Gunho Lee

Abstract

Resource Allocation and Scheduling in Heterogeneous Cloud Environments

by

Gunho Lee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

Recently, there has been a dramatic increase in the popularity of cloud computing systems that rent computing resources on-demand, bill on a pay-as-you-go basis, and multiplex many users on the same physical infrastructure. These cloud computing environments provide an illusion of infinite computing resources to cloud users so that they can increase or decrease their resource consumption rate according to the demands.

At the same time, the cloud environment poses a number of challenges. Two players in cloud computing environments, cloud providers and cloud users, pursue different goals; providers want to maximize revenue by achieving high resource utilization, while users want to minimize expenses while meeting their performance requirements. However, it is difficult to allocate resources in a mutually optimal way due to the lack of information sharing between them. Moreover, ever-increasing heterogeneity and variability of the environment poses even harder challenges for both parties.

In this thesis, we address "the cloud resource management problem", which is to allocate and schedule computing resources in a way that providers achieve high resource utilization and users meet their applications' performance requirements with minimum expenditure.

We approach the problem from various aspects, using MapReduce as our target application. From provider's perspective, we propose a topology-aware resource placement solution to overcome the lack of information sharing between providers and users. From user's point of view, we present a resource allocation scheme to maintain a pool of leased resources in a cost-effective way and a progress share-based job scheduling algorithm that achieves high performance and fairness simultaneously

in a heterogeneous cloud environment. To deal with variability in resource capacity and application performance in the Cloud, we develop a method to predict the job completion time distribution that is applicable to making sophisticated trade-off decisions in resource allocation and scheduling. Our evaluation shows that these methods can improve efficiency and effectiveness of cloud computing systems.

To Sooin and Wonjoong

Contents

List of Figures

List of Tables

Acknowledgements

I was fortunate enough to meet wonderful people and receive a great deal of support from them while pursuing my Ph.D. at Berkeley. Without them, it would not be possible to finish my dissertation.

First of all, I would like to thank my advisor, Professor Randy H. Katz. He not only taught me "how to science" and gave me insightful advice on my research, but also showed me what a great mentor looks like. He has been always supportive, cheerful, and energetic.

I am also grateful to my dissertation committee members, Professor Ion Stoica and Professor Ray R. Larson, and my qualifying exam chair Professor Anthony D. Joseph, for their valuable comments on my proposal and thesis.

My research would never have been fruitful without mentors in the industry. Niraj Tolia and Parthasarathy Ranganathan at HP Labs, Byung-Gon Chun at Intel Labs, Rune Dahl, Michael Abd-El-Malek, and John Wilkes at Google, all helped me to explore interesting problems and guided me to conduct meaningful research.

Discussions with fellow students and researchers in the RAD Lab and the AMP Lab, especially Yanpei Chen, Ariel Rabkin, Andy Konwinski, Matei Zaharia, Ganesh Ananthanarayanan, Rodrigo Fonseca, George Porter, Michael Armbrust, Junda Liu, Charles Reiss, Sara Alspaugh, David Zats, Ali Ghosi, and Rean Griffith, and fellow Korean CS students including Jaein Jeong, Daekyeong Moon, Chang-Seo Park, Yunsup Lee, Jaeyoung Choi, Sangjin Han, and Wontae Choi, were stimulating and entertaining. Behind the scenes, our staff Kattt Atchley, Sean McMahon, and Jon Kuroda made everything happen. I can't thank them enough.

I am deeply grateful to my family. My father and mother taught me important things in my life and always supported me. My grandfather, sister, all other family members, and my grandmother in heaven, have been behind me all the time. My son was born when I was in my second year. As he grew up, I did too. Thanks Wonjoong for making me a better person.

Last and most importantly, I would like to thank my beloved wife Sooin. I would not have been able to go through this without her support. Together, we have made a great journey so far. I am sure that it will be even greater from now on.

Chapter 1

Introduction

1.1 Motivation

Recently, there has been a dramatic increase in the popularity of cloud computing systems (e.g., Amazon's EC2 [4] and Rackspace Cloud [74]) that rent computing resources on-demand, bill on a pay-as-you-go basis, and multiplex many users on the same physical infrastructure. These cloud computing environments provide an illusion of infinite computing resources to cloud users so that they can increase or decrease their resource consumption rate according to the demands.

In cloud computing environments, there are two players: cloud providers and cloud users. On one hand, providers hold massive computing resources in their large datacenters and rent resources out to users on a per-usage basis. On the other hand, there are users who have applications with fluctuating loads and lease resources from providers to run their applications. In most cases, the interaction between providers and users occur as shown in Figure 1.1. First, a user sends a request for resources to a provider. When the provider receives the request, it looks for resources to satisfy the request and assigns the resources to the requesting user, typically as a form of virtual machines(VMs). Then the user uses the assigned resources to run applications and pays for the resources that are used. When the user is done with the resources, they are returned to the provider.

One interesting aspect of the cloud computing environment is that these players are often different parties with their own interests. Typically, the goal of providers is to generate as much revenue as possible with minimum investment. To that end, they might want to squeeze their computing resources; for example, by hosting as many VMs as possible on each machine. In other words, providers want to maximize

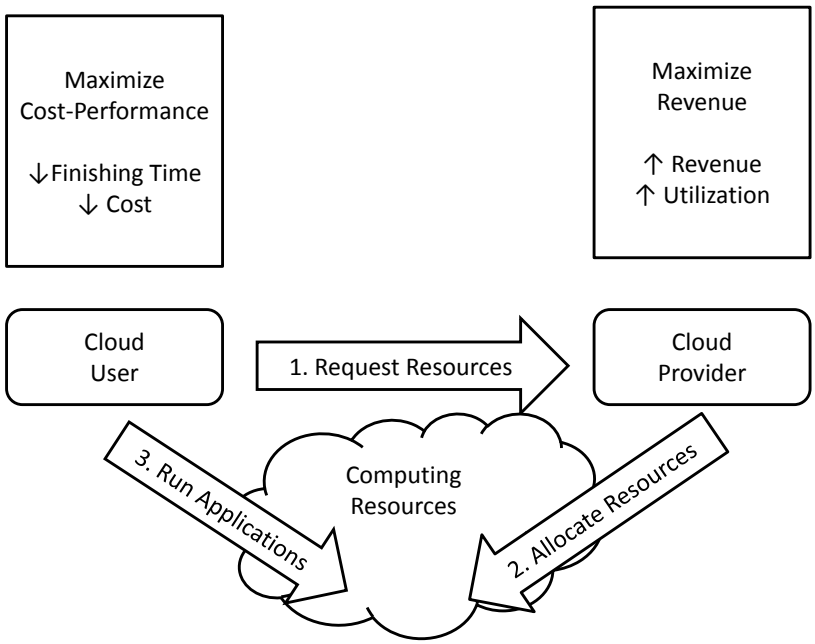

Figure 1.1. Cloud Usage Scenario

resource utilization. However, placing too many VMs on a single machine will cause VMs to interfere with each other and result in degraded and/or unpredictable performance which, in turn, frustrates the users. Thus, the providers may evict existing VMs or reject resource requests to maintain service quality, but it could make the environment even more unpredictable. On the other hand, users want their jobs done at minimal expense or, in other words, they seek to maximize their cost performance. This involves having proper resources that suit the workload characteristics of users' applications and utilize resources effectively. They also have to take unpredictable resources into account when they request resources and schedule applications.

However, these two parties do not want to share information with each other, which makes optimal resource allocation more difficult. For example, providers do not want to expose how many and what kind of machines they have and how they are connected because such information is critical to their business. Similarly, users will not reveal the details of their workload, including source codes and data sets to others, including providers. Therefore, users cannot articulate their resource requests so that the assigned resources are optimal for the applications, because they do not know exactly what is available. Likewise, providers cannot allocate resources in a manner

most suitable to users' applications, since there is no information about their workload patterns. It is also difficult for providers to multiplex their resources effectively by assigning resources to applications with complementary resource usage pattern to minimize interference (e.g., hosting a CPU-intensive workload and an I/O-intensive workload on a machine, rather than two I/O-intensive workloads).

In addition, datacenters (and the resource pool of cloud providers) will be increasingly heterogeneous and consist of various generations of equipment as the technology advances. It is likely that we will see more heterogeneity as the cloud services continue to expand and newer technology continues to emerge. For example, processors with more cores and greater cache memory are continuously being introduced to the market. In addition, energy-efficient processors, such as Atom [3] or ARM [2], which have significantly different characteristics from traditional server processors, begin to see their places in datacenters [22]. Moreover, new technologies are coming to other subsystems as well; solid state drive(SSD) [16], phase change random access memory(PCRAM) [14], and Memristor [9] to memory and storage subsystems; newer network architecture, such as B-Cube [47] and D-Cell [46]; and various accelerators, including GPUs. Cloud providers will need to adopt these new technologies to keep their services up-to-date, but it increases the degree of heterogeneity. Thus, it is necessary to exploit the ever-increasing heterogeneity for both providers and users to achieve their goals: maximizing resource utilization and cost-performance.

Moreover, because multiple users share heterogeneous computing resources without much visibility to each other or to the infrastructure, the cloud environment is highly unpredictable. Even though many cloud providers such as Amazon try to isolate different users' activity to provide a certain level of performance guarantee, there is still a high chance of interfering each other due to resource sharing and contention. Sometimes, cloud providers even voluntarily offer more unpredictable resource containers at a lower cost. For example, Amazon's EC2 offers spot instances for which users make a bid that is much lower than the price of regular instances. If the load to EC2 surges and the price of spot instances spikes higher than the bid, these instances can be evicted anytime. Another example is Micro instances, in which increased CPU capacity is provided in short bursts when additional cycles are available, but there is no guarantee. Even in regular instances, Amazon's EC2 offers different platforms and varying levels of performance within the same instance type [23]. All of these factors add up to the unpredictability of the resource capacity and the application performance.

Thus, we must take these properties of the cloud environment into account to

make cloud services and cloud-oriented applications efficient. By efficient, we mean appropriate resources are allocated at a right time to a right application, so that applications can utilize the resources effectively. In other words, we want to minimize the amount of resources for an application to maintain a desirable level of service quality, or maximize throughput (or minimize job completion time) of an application.

1.2 Challenges in the Cloud

However, we argue that many cloud services and cloud-oriented applications are not efficient enough for the following reasons: 1) lack of information sharing, 2) assumption of homogeneous environments and 3) unpredictability of the environments.

1.2.1 Information Sharing

Because providers and users are often different parties that have their own interests, they usually do not share workload characteristics and detailed resource states.

Typically, providers offer a few types of resource containers with vague rather than precise specifications. For example, Amazon's EC2 uses EC2 Computing Unit (ECU) to describe the CPU capacity of a instance type. One ECU is defined as the equivalent CPU capacity of 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, but it does not provide a strict performance guarantee. The specification of network I/O performance is even more ambiguous; it is only specified as being low, medium, high, and very high, without any numerical metric.

Likewise, the network topology and hierarchy of the machines is not disclosed. Even though Amazon allows users to choose the physical location of VMs by specifying Regions (i.e., geographical location) and Availability Zones (i.e., one Availability Zone is separated from other Availability Zones in the same Region), users are not informed of the location and the connectivity of their VMs in a datacenter.

As a result, users are only able to make resource requests that are based on guesses, which often go wrong. For example, the amount of resources required to run a certain application may be different depending on the micro-architecture of processors. Specifically, Opteron adopts high-speed interprocessor interconnects and built-in memory controller, while Xeon uses shared memory bus with off-chip processor-memory interconnect [45]. This incurs a significant difference in memory latency and bandwidth between machines that are equipped with processors of different micro-

architectures, even if their CPU capacities are rated the same. If the application is optimized for one micro-architecture, it may require many more machines with the other micro-architecture to match the desirable performance. But such a detailed information is usually not made available to users before they actually receive resources.

Similarly, users do not provide the information of their applications and workload that are executed in the VMs. Thus, providers place VMs in an arbitrary fashion, which leads to ineffective resource utilization and degraded performance. For example, if the application involves massive inter-machine communication, it is better to allocate machines in the same rack to the application in order to reduce costly inter-rack communication. On the other hand, if the application delivers contents over the Internet, scattering the allocated resources over many racks will be desirable in order to avoid bottleneck. However, providers are not informed about such details to make better resource allocation decisions.

1.2.2 Heterogeneous Environment

Many cloud applications largely assume a homogeneous environment. For example, Hadoop [6] assumes that all nodes participating in the cluster have the same processing power. Hadoop is an open source implementation of MapReduce, a framework for big data analysis that runs on a cluster of nodes. A Hadoop job is consists of a number of tasks that run on nodes concurrently. When Hadoop schedules a task of a job, it assumes that it takes about the same time to process a task regardless of where it runs. It considers network connectivity by giving preference to tasks that access local data over these access remote data, but does not consider the difference of computing capability of nodes. However, in a heterogeneous environment, some tasks run faster on a particular node than others. For example, if a task is able to exploit accelerators such as GPUs, it will be much faster to run on nodes equipped with GPUs than on regular nodes without such accelerators. To take a full advantage of available hardware, cloud-oriented applications must be heterogeneous-aware.

In addition, it is not straight forward to guarantee fairness among multiple jobs in heterogeneous environments. As described in the previous example, the value of a particular node depends on the job of which task is assigned to the node. Hence, simply adding up the number of nodes or relative speed (e.g., clock speed) that are used by a job does not account for the job's share. For example, a node with GPUs will have a significantly greater value to the jobs that can utilize it than other kinds

of nodes while it is as good as any other nodes if a job is not capable of using GPU. Thus, such differences in the value of nodes depend on the job. we call this *job affinity*. It must be considered when we want a job scheduler to ensure that each job receives its fair share of processing resources.

It is worth noting that some schedulers consider the memory capacity of nodes. For example, the Capacity scheduler [7] of Hadoop does not schedule a task that requires large memory on a node that has less memory than required. The LATE algorithm [98] also improved the existing scheduler by correctly identifying the stragglers in heterogeneous environments. However, they are more focused on preventing failure due to heterogeneity, rather than exploiting it.

1.2.3 Unpredictability

As described in Section 1.1, the cloud environment is highly variable and unpredictable. To increase resource utilization, providers try to oversubscribe as many users to a shared infrastructure. This results in resource contention and interference. Other factors that contribute to unpredictability of the environment include heterogeneity within the same instance type and administrative action (e.g., eviction) to maintain the service level. These make it extremely difficult to predict the performance variability and track down its causes.

The problem is that performance prediction plays an essential role in resource allocation and job scheduling. For example, if a user has a job that needs to be finished within a certain deadline, an adequate amount of computing resources must be allocated. To determine whether or not a certain amount of resources are "adequate", the user needs to predict the completion time of the job with the resources. However, given the variability of the environment, the prediction would be hardly accurate. Hence, we want to estimate not only the completion time, but also the confidence level of the prediction. In other words, we want to quantize the chance of finishing the job within a certain time frame, for example.

1.3 Problem Statement and Contribution

In the Cloud, computing resources need to be allocated and scheduled in a way that providers achieve high resource utilization and users meet their applications' performance requirements with minimum expenditure. We shall call this "the cloud resource management problem".

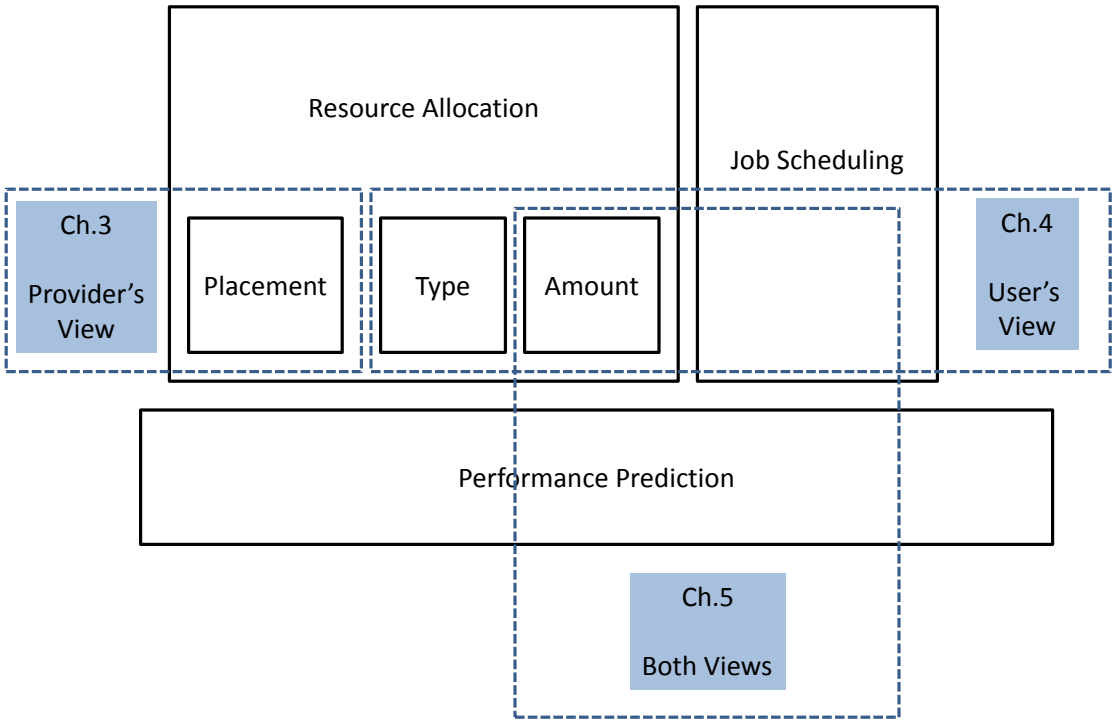

Figure 1.2. Elements of Resource Allocation and Job Scheduling

However, as described in the previous section, there are challenges that make it difficult to solve the problem effectively. This thesis addresses the challenges in the Cloud caused by lack of information sharing, assumption of homogeneous environments, and unpredictability of the environments.

In addressing these challenges, we use MapReduce as our target application. Arguably, MapReduce is one of the most important classes of applications that currently run in the Cloud. In addition, its structure is relatively well defined, which enables us to build an execution model and perform a simulation. Details of MapReduce is described in Section 2.1.2.

Thus, the problem is narrowed down to making MapReduce efficient in dynamic and heterogeneous cloud environments. Depending on the context, either the job completion time or the cost to lease the resources is used to measure the efficiency. To tackle the problem, we divide the problem into a few elements by looking at it from various aspects.

1.3.1 Elements of Our Problem

Figure 1.2 depicts the elements of our work. In a cloud environment, computing resources are allocated when a user (or a user application) makes a request. The type and amount of resources to allocate are determined according to the user request and availability of the resource, and the user application (or its processes) is placed somewhere in the provider's datacenter. Once the resources are allocated, a scheduler of the user application is responsible for assigning its sub-tasks to particular resources. In making these decisions, performance prediction plays an important, cross-cutting role. We describe each of these separate problems in the following subsections.

1.3.1.1 Resource Allocation

Resource allocation involves deciding what, how many, where, and when to make the resource available to the user. Typically, users decide the type and amount of the resource containers to request, then providers place the requested resource containers onto nodes in their datacenters. To run the application efficiently, the type of resource container need to be well matched to the workload characteristics, and the amount should be sufficient to meet the constraints (e.g., job completion time deadline). In an elastic environment like the Cloud where users can request or return resources dynamically, it is also important to consider when to make such adjustments. Related works in resource allocation are listed in Section 2.2.1.

1.3.1.2 Job Scheduling

Once the resource containers are given to the user, the application makes a scheduling decision. In many cases, the application consists of multiple jobs to which the allocated resources are given. For example, an MapReduce cluster in the cloud may run many jobs concurrently. The job scheduler is responsible for assigning preferred resources to a particular job so that the overall computing resources are utilized effectively. The application also has to make sure each job is given adequate amount of resources, or its fair share. Such a scheduling decision becomes more complex if the environment is heterogeneous. Among a large amount of researches on job scheduling, some closely related works in heterogeneous environments and for data-intensive frameworks like MapReduce are surveyed in Section 2.2.2.

1.3.1.3 Performance Prediction

In making a resource allocation decision, performance prediction plays an important role. We use performance prediction to estimate the completion time of a job with given resources, and to determine whether the job will be finished by the deadline with the given amount of resources. If it is not likely to meet the deadline, the user may want to request more resources. It is also used to estimate the outcome of a particular placement instance. By comparing the predicted completion time of various placement instances, the provider can pick the best placement. If the prediction is not accurate, we will end up either violating the deadline or having redundant resources. Variability in the computing environment makes it harder to predict performance accurately. Various methods to predict the execution time are summarized in Section 2.2.3.

1.3.2 Our Contributions

We attempt to solve the cloud resource management problem from various aspects. First, we look at the resource placement problem from the providers' perspective. Secondly, from the users' viewpoint, we address resource allocation and job scheduling issues. Lastly, we tackle performance prediction; that is, the basis of decision-making for both providers and users.

Thus, our contribution is three-fold. The first contribution of this work is a resource placement framework that is application- and resource-aware. We will show that providers can allocate resources more effectively by having users specify a few parameters of their workload without exposing greater detail. The experiment results show that our topology-aware resource allocation scheme can reduce completion time by up to 59% when compared to simple allocation policies.

The second contribution is a heterogeneity-aware resource allocation and job scheduling scheme for MapReduce workloads. We propose an overhauled scheduler that considers heterogeneity while keeping the simplicity of the current scheduler. In case studies, our scheme could cut the cost to maintain the cluster about 28%, and improved the performance of a job that can utilize GPU by 30% without penalizing other jobs.

The third contribution is a method with which to predict the performance of a job in a form of completion time distribution rather than a single point value. In addition, we describe how such a prediction method can be applied to determine the

amount of resources to be allocated in unpredictable cloud environments. It enables us to allocate resources to meet the deadline with a certain level of confidence, which is difficult to achieve with a point value prediction.

1.4 Thesis Organization

The rest of thesis is organized as follows.

In Chapter 2, we provide background information on cloud computing environments and MapReduce, our target application class. We also identify previous works that are related to our work.

In Chapter 3, we look at the resource placement problem in cloud computing environments. Even though the placement of VMs can significantly impact application performance, most cloud providers usually allocate resources arbitrarily as they are not aware of the hosted application's requirements. To address this problem, we present an architecture with which to guide allocation decisions taken by the providers.

In Chapter 4, we consider resource allocation and job scheduling in heterogeneous environments. In such an environment, it is important to schedule a job on its preferred resources in order to achieve high performance. In addition, it is also crucial to provide fairness among multiple jobs. To that end, we suggest an architecture to allocate resources to a MapReduce cluster in the cloud and propose a metric of share in a heterogeneous cluster to realize a scheduling scheme that achieves high performance and fairness.

Chapter 5 describes a method to predict the completion time *distribution* with stochastic values, rather than point values. In addition, we will describe how to apply such a prediction method to resource allocation in unpredictable cloud environments.

We summarize our contributions and present future works in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we take a closer look at our target environment, the Cloud, and the target application, MapReduce. We also investigate related works regarding resource allocation, scheduling, and performance prediction problem. Finally, we enumerate hardware and software infrastructure used in our work.

2.1 Background

2.1.1 Cloud Computing

The cloud computing environment refers to the hardware and systems software in the datacenters that provide computing resources as services [24]. Below is service models of cloud computing defined by National Institute of Science and Technology (NIST) [67]. Note that they used the term "consumer" instead of "user".

> *Software as a Service (SaaS).* The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.
>
> *Platform as a Service (PaaS).* The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and

tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

Throughout this thesis, we refer to IaaS by "the cloud environments" or "the Cloud" unless otherwise specified. SaaS and PaaS are also important service models of cloud computing, but we note that they are out of our focus in this thesis.

The last few years have seen a dramatic growth in the availability and demand for "cloud" systems, or IaaS, best exemplified by Amazon's EC2. Amazon's EC2 [4] is one of the most widely used cloud services. It offers various virtual machine (VM) types with different capacities. Users rent a number of VM instances to run their applications and pay by the hour for active instances. Rackspace [74] and Joyent [56] offer similar services. Because these services are publicly available, they are often called "public" clouds.

In contrast, "private" cloud refers to the infrastructure operated solely for a single organization. For example, Eycalyptus [71] is a software platform for the implementation of private cloud computing on computer clusters. It exports a user-facing interface that is compatible with Amazon's EC2. Its resource allocation policy is modularized and extensible, and currently supports two simple policies; Greedy and Round-robin.

The cloud users rent compute cycles, storage, and bandwidth with small minimum billing units (an hour or less for compute and per-MB for storage and bandwidth) and almost-instant provisioning latency (minutes or seconds). These systems contrast with traditional colocation centers (colos), where equipment leases span months and provisioning resources can take days or longer.

By using container-based mechanisms such as virtual machines (VMs), most cloud systems are able to securely multiplex many customers on the same physical infras-

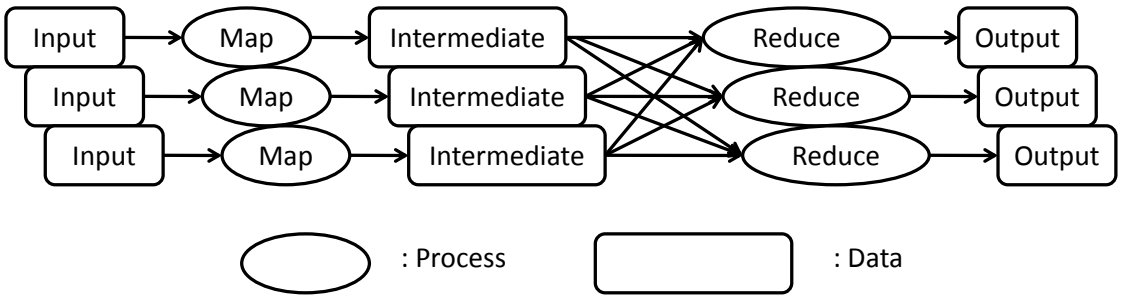

Figure 2.1. MapReduce Workflow

tructure to deliver a cost-effective service. At the same time, to host a large number of customers simultaneously, a cloud system tends to be large. It consists of servers spread across many racks, rooms, and even data centers. This has allowed cloud systems to provide economies of scale that were previously available only to the largest enterprises.

The ability to scale an application or service instantly, and then release resources when finished, has become a powerful incentive for cloud users. This feature is being used by companies ranging from startups and small and medium-sized businesses to larger enterprises (e.g., *The New York Times* newspaper [13]; Eli Lilly, the pharmaceutical firm [15]; and the NASDAQ stock exchange [10]). The convenience and overall cost-effectiveness of these systems seem to have outweighed any tradeoffs for customers that do not require a fully utilized cluster around the clock.

2.1.2 MapReduce

MapReduce [35, 93] is a software framework that supports data-intensive computation on large clusters developed by Google. It is best suited for embarrassingly parallel and data-intensive tasks. It is designed to read large amount of data stored in a distributed file system such as Google File System (GFS) [39], process the data in parallel, aggregate and store the results back to the distributed file system. Figure 2.1 briefly describes the MapReduce workflow. Typically, the input data is stored in a distributed file system, divided into a number of splits. Each split is loaded and processed by a number of map tasks, which in turn generate corresponding intermediate data that is grouped by keys. Then the intermediate data is shuffled (i.e., sent to corresponding reduce tasks according to the key) and processed by reduce tasks. Each of the reduce tasks is responsible for a range of key space and produces the final output, which is stored back to the distributed file system.

Key ID	**Value**
$[K_1, V_1]$	$[line_0, \text{'coast to coast'}]$
$[K_2, V_2]$	['coast', 1] ['to', 1] ['coast', 1]
$[K_2, < V_{2_1}, V_{2_2}, \cdots >]$	['coast', $< 1, 1 >$] ['to', $< 1 >$]
$[K_3, V_3]$	['coast', 2] ['to', 1]

Table 2.1. MapReduce Word Count Example

At its core, a MapReduce program centers on two functions: a *map* function and a *reduce* function. The framework converts the input data into key and value pairs ($[K_1, V_1]$) that the map function then translates into new output pairs ($[K_2, V_2]$). The framework then groups all values for a particular key together ($[K_2, < V_{2_1}, V_{2_2}, \cdots >]$) and uses the reduce function to translate this group into a new output pair ($[K_3, V_3]$). Example values for these key-value pairs with a MapReduce Word Count program and the text "coast to coast" can be found in Table 2.1. In reality, a MapReduce program can be composed of multiple map and reduce phases with the reduce output from one phase serving as the map input for the next.

In a MapReduce system, there is a single master node that manages the whole cluster. A number of slave nodes subscribe to the master to join the cluster. Each slave have one or more slots depending on the computing capacity (e.g., memory and CPU), and each slot hosts a map or reduce task at a time. Note that slaves are sometimes called "worker", especially when it has only one slot.

The master node accepts MapReduce jobs written by users, and responsible to schedule the job on the cluster. When the scheduler finds an available slot, it assigns one of the pending tasks to the slot. When it choose a task to assign, especially for map tasks, priority is given to tasks that access data split stored on the local machine over tasks that access remote data. When a task is found running significantly slower than it should, the scheduler speculatively run a copy of the task, or backup task, hoping the backup task finishes faster than the original one. Similarly, the scheduler

re-execute tasks that are terminated or of which output data is lost due to node failure.

Hadoop [6] is an open source implementation of the MapReduce framework. Hadoop includes the Hadoop Distributed File System (HDFS), which is also a clone of GFS. A body of research have been conducted based on Hadoop because it is publicly available and widely used in open source community. For example, researchers have made a number of improvements in the Hadoop scheduler, including complex proportional [81], queue [99] and flow-based [51] scheduling systems.

In the following chapters, we describe greater details of MapReduce and Hadoop that are related to each chapter. In Chapter 3, we study how to characterize a MapReduce job to simulate the execution. The scheduling process of MapReduce framework is investigated in depth in Chapter 4 to make it effective in heterogeneous environments. In Chapter 5, we take a closer look at each phase of a MapReduce job to capture the variability in execution time.

2.2 Related Works

2.2.1 Resource Allocation

A vast amount of research has been conducted on resource allocation or job placement, especially in the Grid/high-performance computing community. Systems such as Condor [34] and Globus Toolkit [40] have been used to share computing resources across organizations. In Globus, customers describe their required resources through a resource specification language that is based on a predefined schema of the resource database. The task of mapping specifications to actual resources is performed by a resource co-allocator, which is responsible for coordinating the allocation and management of resources at multiple sites. Many other works are similar, basically matching the specification to resources. However, user-specified resource requirements do not guarantee the optimal resource allocation, as there might be no available resources match to the specification, or the specification itself might even be wrong.

There are also a few studies on communication-aware job placement. However, these have concentrated predominantly on the overheads of WAN communication between Grid sites [37, 62, 84] and have generally assumed that the network availability within a site is homogeneous [66]. More closely related is the work by Santos et al. [82] on combining administrator policies or preferences with resource allocation decisions,

but it looked only at coarse-grained network usage and used simulation-based evaluation to examine allocation decisions, without considering application performance.

Besides the job placement, determining the number of machines to be allocated to each application is an important problem in an elastic environment such as the Cloud. Market-based [94, 60] and SLA-based [97] approaches have been proposed to consider application performance or business value. Verma et al. [91] used an analytical model to estimate the upper and lower bounds of the MapReduce completion time, and they used the result to determine the right size of resources to meet the service level objective (SLO).

As will be presented in the following chapters, we use a lightweight MapReduce simulator as a part of our solution to the resource allocation problem. Similarly, Wang et al. [92] also built a Hadoop simulator to predict completion time on different network topologies. However, the main objective of their study is to examine how a physical MapReduce cluster should be built. It is built using an external network simulator, ns-2 [12], to simulate the data transfers and communication among the nodes. It makes a per-job simulation on the order of minutes, which is too slow to use in determining an optimized resource allocation. In a similar vein, Kambatla et al. [58] attempted to determine the optimal values for MapReduce's configuration parameters (e.g., number of map and reduce slots) using the resource consumption signatures of similar applications. StarFish [48] also set up optimal Hadoop configuration parameters based on profiles and what-if simulations.

2.2.2 Scheduling in a Heterogeneous Environment

In many systems, a heterogeneous environment is preferable to one that is homogeneous [17, 18]. However, it provides better performance only for particular systems and workloads [41]. Even if the workload itself is more suitable to a heterogeneous environment, the system's scheduling algorithm should exploit heterogeneity well to benefit from it. Otherwise, it will lead to undesirable outcomes.

The process of scheduling parallel tasks determines the order of task execution and the processor to which each task is assigned. Typically, an optimal schedule is achieved by minimizing the completion time of the last task. Finding the optimal schedule has long been known as an NP-complete problem in both homogeneous and heterogeneous environments [50]. Therefore, many heuristics have been proposed [64, 29, 86, 27, 63] to find a feasible solution within a reasonable time. Some heuristics are known to have a bound on the deviation from the optimum [55]. However, most

of those studies have concentrated on processing power and neglected other resources such as network bandwidth.

In contrast, data-intensive computing systems, such as Hadoop [6] and Dryad [52], schedule tasks in favor of data-locality while assuming homogeneity in machines and tasks. If a machine or a task turns out to be slower than the others, it is treated as faulty and handled by speculative task re-execution.

Zaharia et al. [98] pointed out that this mechanism does not work well in a heterogeneous environment and modified the speculative execution routine to mitigate the problem. Their algorithm, which is called LATE, speculatively executes the task that has the longest approximate time to end rather than tasks that have been running longer.

Ananthanarayanan et al. [19] improved this further by adopting early re-execution of outliers and network-aware placement of tasks. SkewReduce [59] took a different approach; it tries to partition input data properly to minimize deviation in the task execution time. However, these studies mainly try only to avoid the negative effect of heterogeneity, rather than exploit it for better performance.

Tian et al. [88] proposed a scheduler that overlaps CPU-bound and I/O-bound tasks to improve the overall utilization of the cluster [88]. They explored a limited form of workload heterogeneity.

Besides completion time, fairness is another important criterion for scheduling tasks if there are multiple jobs consisting of tasks to schedule. Fairness among jobs should be considered to keep any jobs from starving or being over penalized.

Simple methods, such as static-partitioning, may be used to achieve fairness, but they usually sacrifice the overall performance because no job can use more resources than assigned even if there are available resources that are not used by other jobs.

The delay scheduler [100] in Hadoop resolves the problem by letting a job without data-local tasks wait for a small amount of time to improve throughput while preserving fairness.

Quincy [51] is a flow-based fair-share scheduler for Dryad, which is a more generalized variant of MapReduce framework. It maps the scheduling problem to a graph in which edge weights and capacities represent data locality and fairness, and then it uses standard optimization solvers to find a schedule.

Sandholm and Lai [81] used user-assigned and regulated priorities to optimize

the MapReduce schedule by adjusting resource share dynamically and eliminating bottlenecks. However, these studies still did not take heterogeneity into account.

In summary, there have been a body of researches in data-intensive computing systems that take either heterogeneity or fairness into account, but not both. In this thesis, specifically in Chapter 4, we attempt to exploit heterogeneity and ensure fairness at the same time.

2.2.3 Performance Prediction

Another difficult issue in the scheduling problem is predicting the execution time of each task. Many studies have assumed that the execution time of a given task is a known quantity. However, in reality, it is not readily available [54]. In the literature, there are three major classes of solutions to the problem of execution time estimation problem: *code analysis* [76], *analytic benchmarking/code profiling* [96], and *statistical prediction* [53].

In *code analysis*, an execution time estimate is found through analysis of the source code of the task. It can provide an accurate estimation if done right, but it is typically limited to a specific code type or a limited class of architectures.

Analytic benchmarking uses benchmarks for a number of primitive code types and code profiling to determine the composition of a task. The benchmarking data and the code profiling data are then combined to produce an execution time estimate. Even though it lacks a proven mechanism for producing an execution time estimate from the data over a wide range of algorithms and architectures, and it is difficult to compensate for variations in the input data set, analytic benchmarking can determine the relative performance differences between machines.

The third class of execution time estimation algorithms, *statistical prediction algorithms*, make predictions using past observations. Statistical methods have the advantages that they are able to compensate for the parameters of the input data and do not need any direct knowledge of the internal design of the algorithm or the machine. For example, Ganapathi et al. [38] used statistical models to predict resource requirements for MapReduce jobs. However, this class of methods does not work well with new types of tasks without enough past observations.

Moreover, most of the methods provide a single point value prediction. In production environments like the Cloud, the performance of the underlying machines is highly variable and the single point prediction is not reliable enough. Some studies

have tried to generate multiple predictions [68, 91], but they are still point values and it is difficult to determine which point would be correct. In a single machine, Schopf and Berman [83] proposed using stochastic values to represent a range of possible behaviors.

In this thesis, we use a simulation-based method that utilizes a mix of analytic benchmarking and statistical prediction. Specifically, metric derived from micro-benchmarks and historical data is used to determine the parameters of the simulation that estimate the job completion time. In addition, in Chapter 5, we try to extend the method to generate a distribution of the job completion time, rather than a single point value.

2.3 Research Infrastructure

Throughout this thesis, we exploit a number of existing systems and previous works to implement our system and run experiments. In this subsection, the research infrastructure used in this thesis is described.

Open Cirrus [1] is an open cloud-computing research testbed designed to support research into design, provisioning, and management of services. Particularly, we used the Open Cirrus testbed at HP Labs to evaluate our work in Chapter 3, because it allows us to access the underlying physical infrastructure. Amazon's EC2 [4] is one of the leading cloud computing services. It is used to run experiments in Chapter 4, as it offers virtual machine instances with GPUs. In Chapter 5, we used one of the Google's datacenters. It is not publicly available, but offers very large amounts of resources and is heavily utilized. Hence, it suits to our study on unpredictable environments.

MapReduce [35] is our target application in this thesis. MapReduce is a framework to support data intensive parallel jobs, and users can write their own MapReduce program. MapReduce was originally implemented by Google, but their implementation is not publicly available. Instead, the open source Hadoop [6] is widely used outside Google. In particular, we use Hadoop version 0.18.3 in Chapter 3 and version 0.20.2 in Chapter 4. In Chapter 5, we use Google's MapReduce implementation.

We use various MapReduce programs for the purpose of benchmarking in each chapter. The one common benchmark used throughout this dissertation is Sort. It is one of the representative benchmarks that is used in many MapReduce studies. It

is useful for benchmark purpose because it exercises most subsystems by generating enough amount of intermediate and final data.

In addition, we use a part of gridmix2 benchmark [5] in Chapter 4. Gridmix2 is a suite of MapReduce benchmarks that covers a wide range of workloads that include sorts, data scans, and "monster query". Benchmarks in Gridmix2 follow patterns that are observed in real-world workloads, so they are more suitable to emulate production jobs than simple Sort.

We also use several custom-made benchmarks. In Chapter 3 and 5, we use benchmarks that generate less intermediate data to see the difference from Sort that generates as large intermediate data as input data. In Chapter 4, we use a benchmark that utilizes GPU to investigate the case that the preference of a node is significantly different from job to job.

ParadisEO [31] is a meta-heuristic framework to implement genetic algorithms, and used to build our search engine in Chapter 3.

The Hadoop Fair Scheduler [8] is an implementation of the Delay scheduler [100]. We modified it to implement our progress share based scheduling algorithm in Chapter 4.

We begin our investigation into the cloud resource management problem by studying a resource placement problem from the providers' perspective in the next chapter.

Chapter 3

Topology-Aware Resource Placement

In this chapter, we first look at the initial placement problem in cloud computing environments. Even though the placement of VMs can significantly affect application performance, many cloud providers usually allocate resources using simple or random policies as they are not aware of the hosted application's requirements. To address this problem, we present an architecture to guide allocation decisions taken by the providers. Experimental results show that our topology-aware resource allocation scheme can reduce completion time by up to 59% when compared to simple allocation policies.

We begin by motivating the need for application- and topology-aware resource allocation scheme in Section 3.1, then we describes the architecture of the allocation engine in Section 3.2. We evaluate the architecture by comparing to application-independent allocation policies in Section 3.3 and summarize this chapter in Section 3.4.

3.1 Motivation

As presented in Chapter 2, Cloud-based Infrastructure-as-a-Service (IaaS) systems rent compute resources *on-demand*, bill on a pay-as-you-go basis, and multiplex many users on the same physical infrastructure. IaaS systems usually provide Virtual Machines (VMs) that are subsequently customized by the user.

Many applications run in the Cloud, especially data-intensive applications such as MapReduce, involve large amount of inter-machine communication. Hence, significant

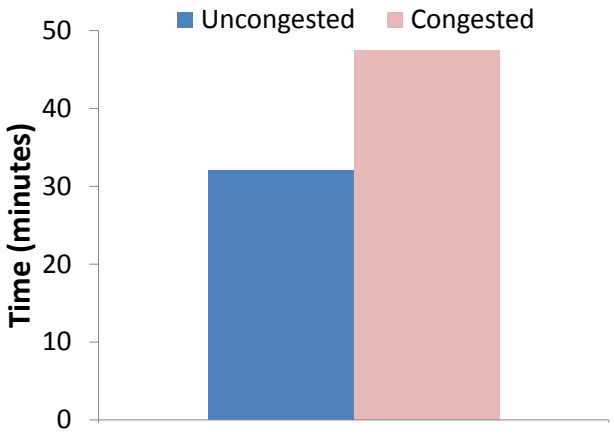

Figure 3.1. Difference in Sort Execution Times

efforts have been made to improve bisection bandwidth in datacenters [47, 44, 69]. Ananthanarayanan et al. [21] even suggest that the improvement in network technology will make the data location irrelevant. However, even though such improvements have been adopted in several datacenters [11], many data centers still have network over-subscription ratios between 5:1 and 10:1 between servers in a rack and the rack's uplink [26] and up to 240:1 when compared to the network core [43]. In addition, background traffic from other co-located workloads only exacerbates the problem. Therefore, Thus, the placement of these VMs can significantly impact application performance.

As an example, Figure 3.1 shows the dramatic difference that an incorrect placement makes. For this experiment, we performed a distributed sort with 90 GB of data and 18 nodes. The nodes were equally divided into two clusters with full bisection bandwidth available within each cluster. As seen in the figure, allocating the clusters in a way that inter-cluster traffic flows through a congested part of the network (limited to 500 Mbit/s) increases the benchmark completion time by almost 50% when compared to an allocation where there is no constraint on network traffic.

We might use simple heuristics such as a best-fit or greedy allocation algorithm to place VMs in the same rack and avoid network congestion. However, this requires the job to be small enough to fit in a single rack. Further more, in any datacenter that allocates resources dynamically, the datacenter will, over time, encounter scenarios similar to memory fragmentation where each rack will have only small or moderate-sized "holes" that could accommodate incoming workloads, which will limit

the effectiveness of simple heuristics. Finally, heuristic-based solutions often perform poorly due to the non-obvious relationship between resource allocation and application performance [65].

Thus, we believe that both the workload's resource usage characteristics and the topology of the IaaS need to be carefully considered to determine an optimized allocation policy. Since IaaS providers today are unaware of the hosted application's requirements, they allocate resources independently of an application's requirements. Similarly, as IaaS users do not have fine-grained visibility into or control over the underlying IaaS infrastructure, they can only rely on application-level optimization. While coarse-grained resource selection (e.g., restricting task execution to a particular data center) can be used, it is insufficient for optimizing performance.

Further, even though application-level optimization techniques [51, 99] can be used, they only alleviate the problem within an existing resource allocation. For example, when dealing with communication-intensive workloads, allocating VMs without considering network topology reduces performance by requiring inter-VM traffic to traverse bottlenecked network paths. It is therefore critical to optimize the initial resource allocation that could be responsible for the majority of performance anomalies.

One possible alternative might require users to explicitly specify their resource requirements, or "hints," to guide resource allocation. For example, users may specify a desired topology of machines they are renting. However, if based on incomplete information, hints can be incorrect or, depending on IaaS resource availability, impossible to satisfy. A successful solution therefore requires the IaaS to derive the information necessary for optimization with minimal or no user input. Further, when compared to application-independent allocation policies, it should improve performance with low latency and high confidence.

To address this issue, we propose an architecture that adopts a "*what if*" methodology. Our solution gathers information without explicit user input, and uses the information to forecast the performance of any particular resource allocation. Our prototype for Topology-Aware Resource Allocation (TARA) is composed of a prediction engine that uses a lightweight simulator to estimate the performance of a given resource allocation and a genetic algorithm to find an optimized solution in the large search space.

To check the feasibility of our approach, we evaluate it in a particular context. Specifically, we use a couple of Hadoop MapReduce applications that are sensitive

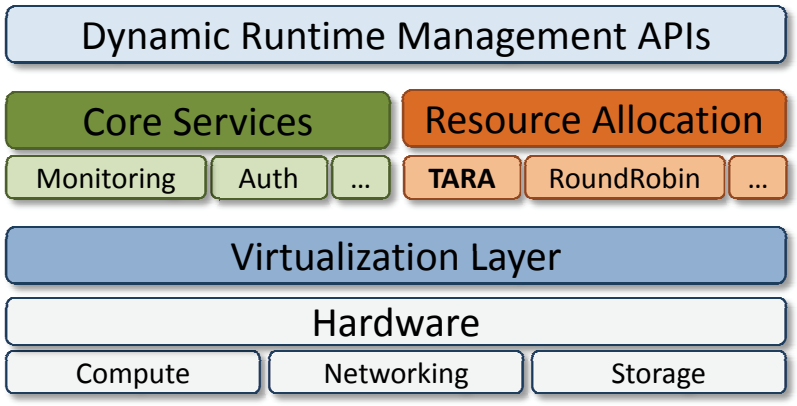

Figure 3.2. TARA's Integration into the IaaS Stack

to the network topology underlying their allocated resources. HP Labs Open Cirrus cluster [1] is used to emulate bottlenecked network topology and run the applications. More details are presented in Section 3.3.1 and 3.3.2.

Section 3.2 describes TARA's architecture. Our prototype-based evaluation in Section 3.3 demonstrates the accuracy and scalability of the prediction engine and TARA's effectiveness when compared to application-independent resource allocation policies. Then we conclude in Sections 3.4.

3.2 Architecture

Figure 3.2 shows a high-level overview of how TARA integrates into a typical IaaS stack. As shown in the figure, TARA is one of the possible resource allocation policies. Other policies may include simple ones such as "round-robin" that fills available resources up in order, or "random" that picks resources in arbitrary fashion. In other words, TARA is a part of the resource allocation subsystem. The difference between TARA and other simple policies is that TARA tries to come up with the optimal resource allocation by estimating the outcomes of a number of possible allocations while others simply follow pre-determined heuristics. As TARA requires application and topology-specific information to optimize resource allocation, it interacts with many other IaaS subsystems including the virtualization layer, monitoring system, and the runtime APIs.

Figure 3.3 describes the composition and the work flow of TARA. To make a resource allocation decision, TARA retrieves various inputs; what is the resource allocation optimized for (objective function, described in Section 3.2.1.1), what does

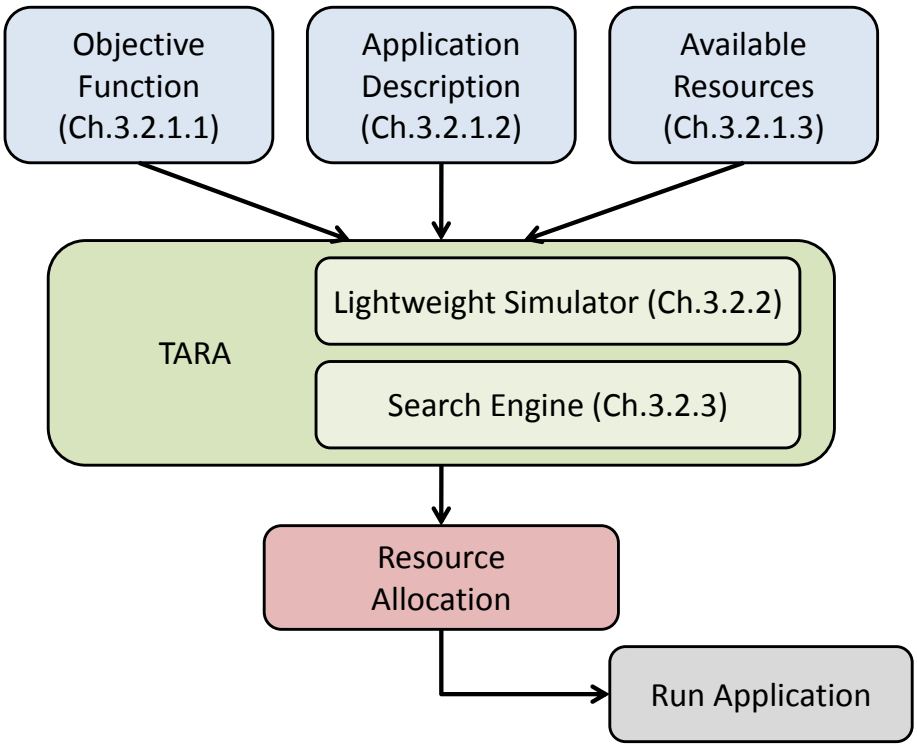

Figure 3.3. TARA's Prediction Engine Architecture

the application look like and how many resources it requests (application description, in Section 3.2.1.2), and what kinds of resources are avaialble and how they are connected (available resources, in Section 3.2.1.3). Based on these inputs, TARA search for the optimal placement of resources allocated to the application.

TARA is composed of two major components: a lightweight simulator-based prediction engine and a fast genetic algorithm-based search engine. The prediction engine, described in Section 3.2.2, is the entity to estimate the metric for the objective function of a particular resource placement. The search engine iterates through the possible subsets of available resources (each distinct subset is called a *candidate*) and have them evaluated by the prediction engine. The candidate that maximizes (or minimizes) the objective function is considered optimal, and used to make an actual resource allocation.

Because the prediction engine need to evaluate a lot of candidates, it must be fast enough to make a final resource allocation decision in a reasonably short period of time. At the same time, it need to be accurate enough to make comparison between different candidates. Hence, we developed a lightweight simulator that balances speed and accuracy.

Even with a lightweight prediction engine, exhaustively iterating through all possible candidates is infeasible due to the scale of the Cloud. We have therefore developed a genetic algorithm-based search technique, described in Section 3.2.3, that allows TARA to guide the prediction engine through the search space intelligently.

3.2.1 TARA Inputs

Basically, TARA requires three categories of inputs to make a resource placement decision; the objective function that the placement is optimized for, application-specific information that indictes the application behavior, and the description of the available resources in the data center. We describe these three inputs in greater detail below, and show how they are obtained.

3.2.1.1 Objective Function

The objective function defines the metric for which TARA should optimize. For example, given the increasing cost and scarcity of power in the datacenter [89], an objective function might measure the increase in power usage due to a particular allocation. Alternate objective functions could measure performance in terms of throughput or latency or even combine these into a single performance/Watt metric. Our prototype's objective function uses MapReduce job completion time as the optimization metric because it directly maps to the monetary cost of executing the job on an IaaS system. The output value for the objective function is calculated using the MapReduce simulator described in Section 3.2.2.

Note that a candidate allocation might be a good fit for a given objective function but a bad fit for another. For example, an application distributed across a number of racks might obtain the best performance but could cause the greatest increase in power usage. Thus, while it is possible to define a multi-objective function, it requires the function to either express different metrics in the same unit (e.g., monetary cost) or provide a prioritized ranking of the metrics.

3.2.1.2 Application Description

The application description is used by the prediction engine to determine application behavior once resources are allocated. It consists of three parts: 1) the framework type that identifies the framework model to use, 2) workload-specific parameters that

describe the particular application's resource usage and 3) a request for resources including the number of VMs, storage, etc.

The prediction engine uses a model-based approach to predict the behavior of the given application on the selected framework. Specifically, a model takes the framework parameters (e.g., input size) and the description of allocated resources (e.g., the capacity and topology of machines) as an input, and outputs a predicted value for the objective function (e.g., job completion time). For example, we can think of a black box model that is tuned by an execution history. By taking the framework parameters and the allocated resources as a feature vector, we can train a statistical model that predicts the completion time of a given job. Another approach is to run a simulation using given parameters. We take this approach to model the MapReduce framework. It is described with greater detail in Section 3.2.2.

As each framework behaves differently, TARA-based systems require a model for the framework being optimized. Currently, our prototype only supports the Hadoop-based MapReduce framework. However, other framework types can be also supported by either adding additional models or providing support for including user-defined models. If the framework type is not supported or the information is absent, the IaaS will fall back to an application-independent allocation scheme such as a Round Robin or Random placement algorithm.

After the correct model is identified based on the framework type, TARA needs additional runtime-specific information to predict performance (as defined by the objective function). This information is further divided into two groups: framework-specific configuration parameters and job or application-specific resource requirements.

Framework-specific parameters define the configuration of the application-level environment within which the job executes. Examples include the number of threads available in a web server or the number of map and reduce slots configured in the MapReduce framework. This information can usually be automatically derived from configuration files and, in particular, from Hadoop's cluster-specific `conf/hadoop-site.xml` file. As described earlier, tuning these parameters is an orthogonal optimization, but is important as the framework configuration can also significantly impact performance. In this work, we assume that systems such as RS-Maximizer [58] have already fine-tuned the framework's runtime system for the IaaS hardware and these optimized parameters can be used by TARA.

It is harder to extract job-specific resource requirements as they are dependent on a number of factors including the code being executed, the input data set, etc.

While determining this information for arbitrary frameworks is outside the scope of this work, log analysis [80, 85] has proven to be a powerful tool in characterizing an application's runtime resource requirements. For our prototype, we analyzed Hadoop logs to derive information on the job using our own scripts. These logs are usually gathered from a previous application run. Note that Rumen [79] similarly extracts job data from historical logs, but it was released August 2010, which is after the work in this chapter is done.

If a new application is introduced, if the application changes, or if the data set has changed significantly, TARA runs the application on a small subset of data and generates the required logs. This approach can also be followed if the user or system suspects that the specific context of the application has changed (e.g., search for a common keyword vs. an unusual one).

The individual metrics needed for the application-specific resource requirements and model include selectivity (input/output ratio) during the map phase, CPU cycles required per input record for both map and reduce tasks, CPU overhead per task, and the communication requirement for the average task. These metrics and their use are described in greater detail in Section 3.2.2. While these metrics are collected when enough capacity is available, the performance prediction described in Section 3.2.2 accounts for other applications that will be co-located with the new workload.

3.2.1.3 IaaS Information on Available Resources

The final input required by TARA is a resource snapshot of the IaaS data center. This includes information derived from both the virtualization layer and the core monitoring service shown in Figure 3.2. The information gathered ranges from a list of available servers, current load and available capacity on individual servers, and the processing power of virtual CPUs to data center topology and a recent measurement of available bandwidth on each network link.

For the prototype, TARA would obtain topology information from the IaaS monitoring service which in turn can use SNMP data gathered from the data center's switches. Alternatively, if switch support is unavailable but end-nodes can be controlled, tools such as pathChirp [77] can also estimate available link bandwidth.

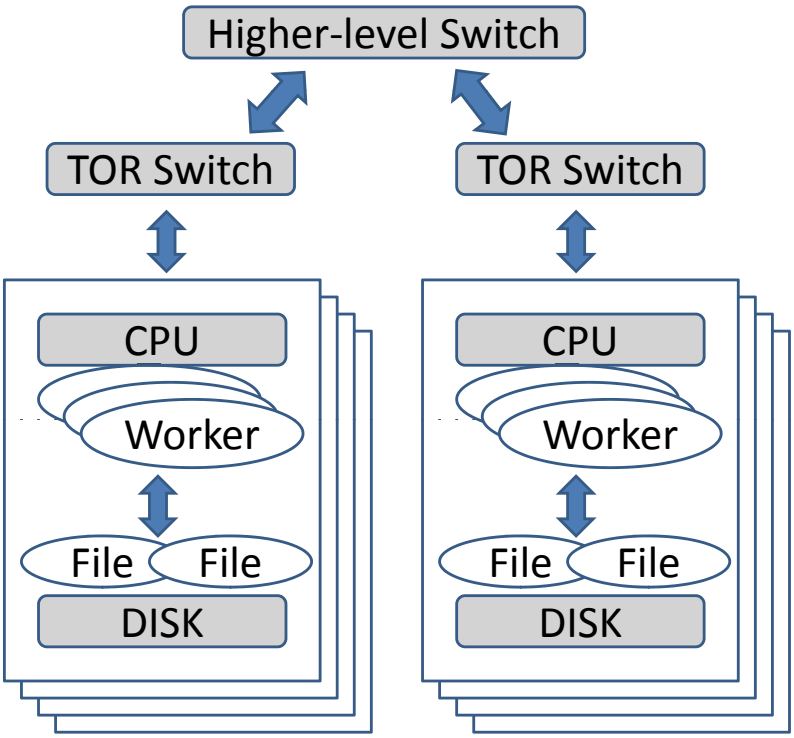

Figure 3.4. Simulation components

3.2.2 Prediction Engine

The prediction engine's role in TARA is to map a candidate resource allocation to a score that measures the "fitness" of a candidate with respect to a given objective function. This fitness score also captures relative behavior of different candidates and allows TARA to compare and rank them. As our prototype supports the Hadoop-based MapReduce framework, we will describe how the prediction engine for MapReduce applications is built.

As the relationship between a set of selected servers, available network bandwidth, and MapReduce performance is not straightforward, it is difficult to come up with a simple model that can be used to quickly evaluate the "fitness" of candidate resource allocations. We therefore adopt a simulation-based approach where the MapReduce framework is simulated, in C++, to predict job completion time, the objective function's metric. For other frameworks, less computationally expensive solutions based on queuing theory might also be available [87].

As there is no benefit of using TARA's prediction-based approach if finding an optimized resource allocation takes an inordinate amount of time, our design is heavily driven by the need to be fast and lightweight. Given that the main objective of

the simulator is to score candidates for comparison, and it is more important to maintain the ordering of good assignments than obtaining accurate completion time predictions, we focus on ensuring the relative performance trend between different candidates. Therefore, unlike other MapReduce simulators [70, 92] that focus on accurately representing MapReduce behavior but often take longer than executing the MapReduce job on real hardware, we decided to trade absolute accuracy for speed.

To enable fast scoring for each candidate, we use a simplified execution model instead of attempting a full-system simulation. For example, we use a stream-based approach to simulate network traffic. While this is not as accurate as a packet-level simulation, our results shows that it is sufficient to compare aggregate performance. Similarly, we use a simple disk model instead of a DiskSim-based approach [30].

Like other simulators [92], we do not model non-deterministic behavior such as speculative execution. While adding these non-deterministic features would improve accuracy, the evaluation of the simulator, presented in Section 3.3.4.4, shows that prediction accuracy is high enough. Hence, we choose not to incur the overhead needed to support these scenarios.

As input, the simulator uses the application description, IaaS resource information, and an allocation candidate, provided by the search algorithm described in Section 3.2.3. The simulator models the application by following the control flow of the Hadoop MapReduce framework. Specifically, our prototype models Hadoop 0.18.3. Based on the framework-specific configuration, the simulator creates a number of workers that will "execute" tasks. As shown in Figure 3.4, these workers, which host map and reduce tasks, need to read input data from the local or remote HDFS nodes, execute the user-defined map and reduce functions on the input, and then either store the intermediate output on local disk or write the final output back to HDFS.

For every map or reduce task, the simulator allocate CPU cycles that are proportional to the input size instead of performing the actual computation. We assume that the algorithm used in each task has linear time complexity, but it can be extended to super- or sub-linear algorithms. It also account for Hadoop's initialization overhead, if any, for each new task. Finally, if the selected server or network links has other workloads present, the simulator also account for the resources already in use by utilizing the information described in Section 3.2.1.3. We use an averaged value of resource usage including network traffic. For data-intensive workloads, our experiments and an independent analysis of a production Dryad cluster [20] have

shown that this to be a reasonable assumption for short jobs. Significantly, Ananthanarayanan et al. [20] have also shown this assumption to hold for long-running tasks due to an averaging effect that arises from the distributed nature of the input data. Alternatively, bursty behavior can be included in the model by a tighter integration with the application-level frameworks or eliminated by shaping network [75] or disk traffic [57].

Each map task consumes a fraction of the input data and generate intermediate output. The size of intermediate output is determined by the selectivity or input/output ratio that is obtained from the job-specific information defined in Section 3.2.1.2. While the size of the intermediate output would vary depending on the contents of the input data, the simulator assumes that it is proportional to the size of input. Following the map step, each reducer then performs a network copy of the intermediate output generated by the map tasks. The maximum parallel copies per reduce and the number of concurrent streams per node allowed is also defined in the framework-specific portion of the application description.

It should be noted that our TARA prototype optimizes MapReduce jobs individually, i.e., it optimizes the incoming workload based on the currently executing workloads in the IaaS system. If faced with multiple simultaneous requests, the current prototype would need to serialize them to prevent inaccurate predictions. However, as shown in Section 3.3.4.1, TARA should not introduce a significant latency when launching multiple MapReduce applications.

3.2.3 Search Algorithm

In any large IaaS system, a request for r VMs will have a large number of possible resource allocation candidates. If n servers are available to host at most one VM, the total number of possible combinations is $\binom{n}{r}$. Given that $n \gg r$, exhaustively searching through all possible candidates for an optimal solution is not feasible in a computationally short period of time.

To help in efficiently identifying an approximate solution, we decided to use a genetic algorithm (GA) [42] to generate possible candidates for the prediction engine to evaluate. Genetic algorithms are a search technique inspired by evolutionary biology for finding solutions to optimization and search problems. In GA, candidates are represented as genes and they evolve toward better solutions. A typical genetic algorithm requires a genetic representation of the candidate and a fitness function to evaluate it.

In comparison to other search techniques, we found that GA is a good match for this problem as it is easy and natural to map resource allocation in a data center to GA's genetic representation, and apply operations during the evolution process. Our results, shown in Section 3.3, also show that this approach outperforms various greedy and heuristic-based algorithms.

To represent each possible candidate, we use a bit string where the string length is equal to n, the number of servers available to host a single VM. For each bit in the string, a value of 1 represents the physical server being selected for hosting a VM and a 0 represents the server being excluded. We assigned one bit to each physical server as our current prototype never allocates more than one VM from the same allocation request to the same physical server. TARA however does allow VMs belonging to other arbitrary applications to be simultaneously executing on the same physical nodes. If we want to allow a server to host multiple VMs from the same allocation request, it is easy to extend the candidate representation. We can consider n to be the number of available VM slots instead of servers and assigning multiple bits to a physical server.

To evaluate a candidate, the prediction engine described above is used. Once we have the genetic representation and the fitness function, GA initializes a population of candidates and then goes through the evolution process of reproduction and selection until it terminates. Detail of each of these steps are described below.

Initialization: The initial population, from which evolution begins, is created by randomly generating a large number of candidates. We picked 100 after sensitivity analysis. If it is smaller, the result tends to converge to a local optimum. If it is larger, it takes longer to find a good solution. The initial population may optionally also contain candidates generated by heuristics to improve the quality of final solution. Each candidate has the number of selected servers (i.e., the number of 1s in the string) fixed to the total number of VMs requested by the job.

Reproduction: In this step, mutation, swap, or crossover operations are applied at random to the candidate population to create offspring, i.e., the next generation of candidates. The mutation operation exchanges two single bits in a string while the swap operation swaps two substrings. In other words, the mutation operation exchanges one server with another while the swap operation swaps more than one server in a selected rack with servers in the other racks. Both mutation and swap perform modifications within a single candidate. In contrast, the crossover operation is used to combine portions of different candidates into a new offspring. After the crossover operation is applied, there might be a different number of servers selected

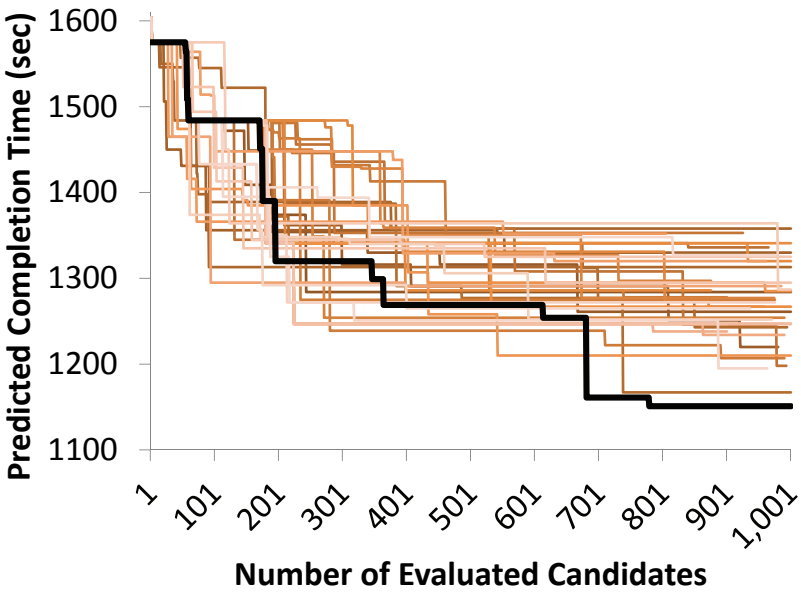

Figure 3.5. Sensitivity of Genetic Algorithm

than requested by the job. If this occurs, the search algorithm will randomly flip bits in the candidate representation to match the number of selected servers to the requested value.

Selection: For each successive GA iteration, or generation, the prediction engine described in Section 3.2.2 is used to evaluate the fitness of each candidate. Once each candidate has been evaluated, a stochastic process is used to select a majority of the "fitter" candidates along with a small percentage of "weak" candidates to maintain population diversity. In our prototype, we ensure that every new generation has at least 10% new offspring.

Termination: As with all search techniques, deciding when to terminate the search process can be done using a variety of policies. In TARA, the search algorithm terminates and returns the best candidate found when it reaches a tunable time limit (60 seconds in the current prototype). This scenario might represent a case when an optimal solution was not found but, as shown later in Section 3.3, the selected candidate was significantly better than those generated by alternate allocation schemes.

As mentioned above, we used 100 candidates for the population size and ensured 10% new offspring in each iteration. A sensitivity analysis, shown in Figure 3.5, was performed to select these values. Each line in the graph represents a different pair of values picked for both population size and offspring rate. The population size and

new offspring rate ranged from 10–500 and 10–100% respectively. While time is not illustrated on this graph, GA with a larger population performs less iterations as it evaluates a larger number of candidates in each step. Similarly, the performance of GA with a smaller population or larger offspring rate plateaus after a short time. The highlighted black line illustrates our chosen values and represents a balanced selection that can find optimized candidates in a short period of time.

Our search algorithm, like the simulator, was written in C++ and used the ParadisEO [31] meta-heuristic framework as the base for the GA. Results in Sections 3.3.4.1 and 3.3.4.4 demonstrate that the both the simulator and the search algorithm met their goals and were lightweight, scalable, and accurate.

3.3 Evaluation

We used two benchmarks to evaluate TARA. Sort, the first benchmark, is synthetic while the Analytics benchmark closely resembles the requirements of a real-world website. The benchmarks, described in greater detail in Section 3.3.1, are followed by details on the experimental setup and methodology in Sections 3.3.2 and 3.3.3. Finally, we present the results from evaluating the TARA prototype in Section 3.3.4.

3.3.1 Benchmarks

3.3.1.1 Sort

Sort is a synthetic benchmark and is included with the Hadoop distribution. This benchmark allows us to generate variable-sized data sets and use Hadoop's distributed framework to sort the generated data. Even though this is a synthetic benchmark, sorting is generally considered to be an integral part of a number of applications including data mining, super-computing, and web indexing. The sort benchmark has also been used by both Yahoo [72] and Google [35] to evaluate their MapReduce frameworks and it is considered to be a good measure of the performance characteristics of the underlying platform. Hence, Sort is used for the benchmarking purpose not only in this chapter, but also throughout this thesis. For the results presented below, we generated 160 GB (2 GB/node) of random data as input for this benchmark.

3.3.1.2 Analytics

MapReduce is being increasingly used for analytics ranging from ad-hoc data processing and business intelligence applications to almost-real time data analysis, search, and trend tracking. As an example of this usage scenario, we converted the MapReduce-based backend system that powers the `trendingtopics.org` website into an Analytics benchmark. The benchmark uses data gathered from Wikipedia access logs to track trends and is similar in function to websites that track emerging themes or "memes" on the Internet. Unlike Sort, this Analytics benchmark is used only in this chapter because there are other benchmarks that are more suitable to highlight the point of each of the following chapters. For example, in Chapter 4, we use a benchmark that utilizes GPUs to contrast the difference in performance, for which Analytics benchmark is not suitable.

For the experiment, we had a dataset that captures hourly Wikipedia article traffic logs gathered from Wikipedia's squid proxy. Our dataset, also used by `trendingtopics.org`, covers the time period from May 1st, 2009 to Dec 31st, 2009 and represents ∼1.12 TB of uncompressed data. In the interest of time, we ran our experiments with a smaller subset (438 GB) of the dataset that covered three months.

The benchmark, like the website, is split into two different MapReduce phases. In the first phase, MapReduce is used to preprocess the data by eliminating non-article or incorrect records, decode special characters used in accessed URLs, and convert the access counts for articles from an hourly interval to a aggregated daily interval. The next stage takes this preprocessed data, maps the daily aggregations by article name and then merges this data in the reduce stage to generate, for each article, a time series that represents the access frequency over the time period captured by the dataset.

3.3.2 Experimental Setup

We used the HP Labs Open Cirrus cluster [1] to evaluate TARA. The testbed used was a subset of the larger cluster and was composed of 111 machines with a single-socket quad-core Intel 3.0 GHz Xeon X3370 processor, 8 GB of RAM, a Gigabit Ethernet port, and four 750 GB disks.

While these machines were distributed over 4 physical racks, the cluster was over-provisioned significantly to support a wide variety of experiments. Each rack contained a maximum of 32 machines and two 48-port top-of-rack switches with a

10 GigE uplink each. Further, both switches were in active use and each switch only connected to half the machines in each rack. Therefore, the cluster was representative of a 8 rack setup with an effective bandwidth oversubscription ratio of 1.6:1. As this is anomalous when compared to commonly observed 5:1 or 10:1 ratios [26], we perfomed rate-limiting all systems on the OpenCirrus testbed at the switch-level to model realistic data centers.

All machines in the test bed ran the Ubuntu 9.04 Linux distribution with Xen 3.4.1 [25] as the virtualization layer. All VMs were configured with access to 4 GB of memory and 4 Virtual CPUs (VCPUs), where each VCPU corresponded to a physical CPU. For our MapReduce framework, we used Cloudera's Hadoop 0.18.3 distribution with an HDFS replication factor of 3 and Sun's Java 1.6.0.

3.3.3 Methodology

3.3.3.1 Topology Creation

Even though we have a moderately large test cluster, it is still considerably smaller than the size of most IaaS systems and, left unmodified, would not have allowed us to adequately test the scalability and performance of TARA's allocation system. To solve this problem, we adopted a ModelNet-like approach [90] where we created virtual topologies and used them as the input to TARA and the different resource allocation policies described below. Based on the resource allocation decision taken, we then used Linux's traffic control and routing mechanisms to create the underlying network topology.

While we experimented with a number of different topologies, we only present the results from a representative topology here. The virtual topology used consisted of 20 racks with each rack containing 40 servers available to host a VM. To model different background workloads and rack-utilization levels, we restricted each rack's uplink bandwidth. To cover a broad range, the restrictions uniformly ranged from 800 Mbit/s to 200 Mbit/s with a step of 100 Mbit/s. The oversubscription ratio is higher than typical data center setup due to the bandwidth limitation of servers used as virtual switches, but it is still in acceptable range considering background traffic in data centers. Creating this virtual topology required us to dedicate 31 nodes as virtual switches and therefore allowed us to run applications that spanned 80 nodes.

3.3.3.2 Resource Allocation Policies

We used five different allocation policies for each of the two benchmarks. They included:

- **RR-R**: This policy allocates VMs in a round-robin (RR) manner across racks (-R).
- **RR-S**: This policy allocates VMs in a round-robin (RR) manner across servers (-S) with a preference for selecting all available servers in a rack before moving on to the next rack. This is the default policy used by Eucalyptus [71] and, based on work by Ristenpart et al. [78], we also believe that it is closest to what is used by Amazon's EC2 for a single job. To positively bias RR-S results, we also enabled this policy to select racks with the highest available bandwidth first.
- **H-1**: This policy is a hybrid (H-1) that combines RR-S and RR-R with a preference for selecting servers in the rack with the greatest available bandwidth but will only select a maximum of 20 servers per rack before moving on to the next best rack.
- **H-2**: This hybrid (H-2) policy is similar to H-1 but only selects a maximum of 10 servers per rack before moving on to the next best rack.
- **TARA**: This policy uses TARA's prediction-engine with a genetic algorithm search to identify the best resource allocation among all available machines.

As stated earlier, each physical node never contains more than one benchmark VM for all of the above allocation policies. Further, input data for all benchmarks is only copied into the VMs after they are launched but before the benchmarks are executed. While the time to copy data into VMs in an IaaS system also impacts overall performance, we do not include it in the below results as, modulo minor differences, it is a fixed cost paid by all experimental configurations.

3.3.4 Results

There are three main goals of our evaluation. First, in Section 3.3.4.1, we quantify the scalability and performance of TARA's prediction engine. Second, in Sections 3.3.4.2 and 3.3.4.3, we quantify application performance using a TARA-based resource allocation vs. the other resource allocation policies described in Section 3.3.3.2. Finally, in Section 3.3.4.4, we confirm that the performance trends predicted by

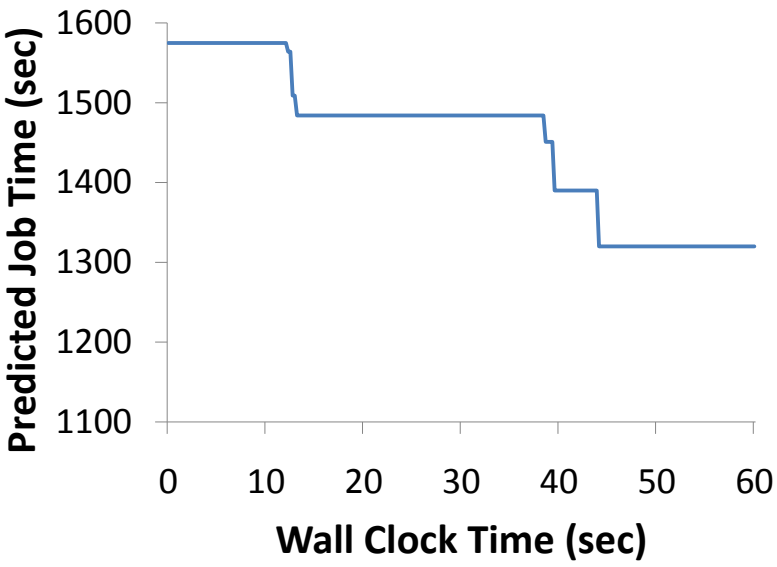

Figure 3.6. Search Algorithm Efficiency

TARA's simulation-based approach matches those observed by running the application on real hardware.

3.3.4.1 Prediction Engine Microbenchmarks

As mentioned in Section 3.2.2, one of the goals behind the prediction engine's design was to be lightweight. We therefore used two different benchmarks to individually quantify the scalability of both the simulator and the search algorithm.

First, we timed how long the simulator took to predict job completion time for the sort benchmark using a given resource allocation of 80 VMs. We repeated this experiment 20 times and discovered that, on average, the simulator took 0.96 seconds to predict completion time with very little variation ($\sigma = 0.019$) between runs.

Second, we examined the efficiency of the search algorithm in finding an optimized resource placement for the same benchmark and topology. The results, seen in Figure 3.6, show that the algorithm quickly discovers better allocations than the initial population (around 12 seconds) but probably has not discovered the optimal candidate at the end of the GA's 60 second deadline. Based on this behavior, one could increase the deadline for longer jobs. However, we should note that TARA's prediction engine is currently unoptimized. The scoring of resource allocation candidates in each iteration is currently done on a single server. Even though the search algorithm

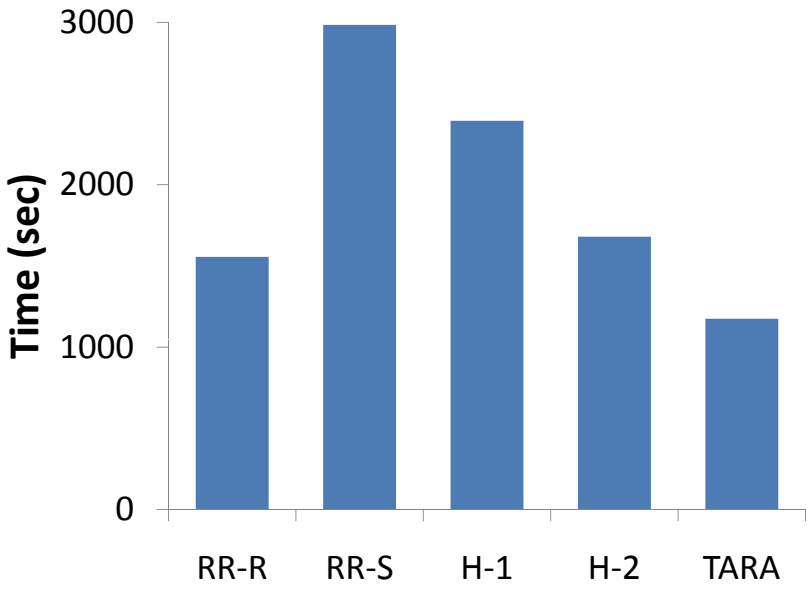

Figure 3.7. Sort Benchmark Results

is multi-threaded, it is limited by the number of CPUs present on the machine. Given the prediction engine's support for MPI, we expect to reduce the prediction overhead by using more machines and an island model [61] to evaluate a larger number of the candidates in parallel. This will allow us to further reduce the prediction time and simultaneously improve the quality of the search results.

3.3.4.2 Sort Benchmark Results

The results from the sort benchmark, described in Section 3.3.1.1, can be seen in Figure 3.7 and clearly illustrate the performance gains that can been obtained by using TARA. Among the application-independent resource allocation policies, we see that RR-R is the best application-independent policy and is followed by H-2. H-1 performs significantly worse than either of these two with RR-S showing the worst performance of the group. The behavior of RR-R is expected because it distributes the workload equally over all racks and is therefore less likely to encounter a network bottleneck. In contrast, RR-S will quickly overwhelm the rack's uplink bandwidth when a large number of VMs are selected in a single rack. The performance of the hybrid heuristics falls in between RR-R and RR-S as they represent a tradeoff between the two extremes.

Finally, once TARA is introduced, its application and topology-aware resource allocation policy performs considerably better than the application-independent poli-

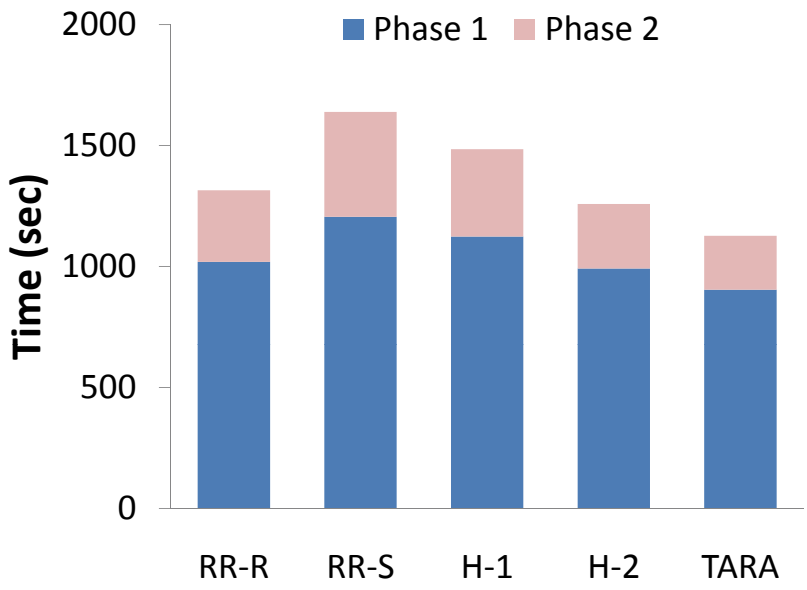

Figure 3.8. Analytics Benchmark Results

cies. When compared to RR-R and H-2, it improves performance by 25% and 28% respectively. The improvements are even more pronounced when compared to H-1 and RR-S with gains of 49% and 59% respectively.

It is also important to note that the presented results do not conclusively prove the effectiveness of one application-independent allocation policy over another. For example, while RR-R always performs better than RR-S, experiments where the job required fewer than 40 VMs showed the inverse. The reason for the change was that the entire workload fit within a single rack and therefore was not constrained by the top-of-rack switch's uplink bandwidth. This behavior helps to emphasize that heuristic-based policies can perform poorly in different scenarios. However, in all cases, TARA always exhibited better performance than any of the application-independent policies with improvements similar to the results described above.

3.3.4.3 Analytics Benchmark Results

The results from the analytics benchmark, described in Section 3.3.1.2, can be seen in Figure 3.8. As this benchmark consisted of two different phases, the results are presented as a stacked bar graph. For Phase 1, TARA improves the applications performance by 1% when compared to the H-2 allocation policy and by as much as 35% when compared to the RR-S policy. Similarly, TARA improves Phase 2's performance by 28% when compared to H-1 and by up to 57% when compared to RR-S.

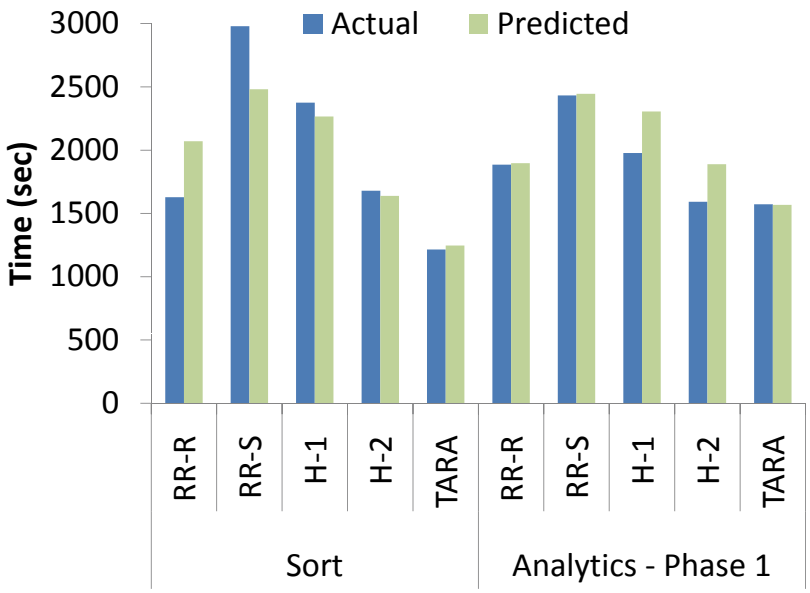

Figure 3.9. Predicted vs. Actual Results

As Phase 1 was more data and communication-intensive, we had therefore expected to see a greater performance improvement for this portion of the benchmark. An analysis of the runtime logs seems to indicate that this was an artifact of the smaller-than-expected input files used in the first phase (each file only captured an hour-long trace). The small input files lead to a large number of map tasks being created which, in turn, lead to a larger number of overlapping and pipelined intermediate data fetches by reducers. Therefore, the network transfer phase wasn't as intensive as predicted by the size of the input dataset. However, once Phase 1 reduced the input data to file sizes more typically seen in MapReduce setups, Phase 2 showed an increased benefit from using TARA. We believe that using larger file sizes for Phase 1 would have similarly improved the observed performance gains. Finally, even though Phase 2's overall runtime is shorter than Phase 1, TARA's overall performance gain, for the entire benchmark, still ranges from 8% to 41% when compared to H-2 and RR-S respectively.

3.3.4.4 Prediction Engine Trends Comparison

Finally, in this section, we compare the prediction engine's output for the results presented in Sections 3.3.4.3 and 3.3.4.2 to the results obtained from one run on real hardware. Figure 3.9 presents the results of this comparison for the Sort benchmark and Phase 1 of the Analytics benchmark. Phase 2 of of the Analytics exhibited the same trends.

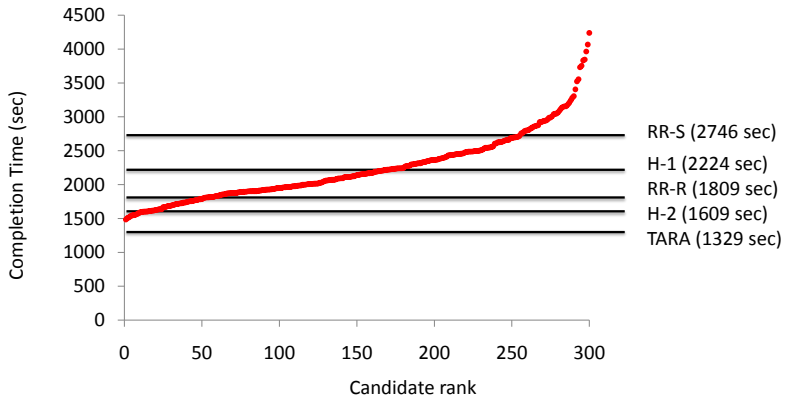

Figure 3.10. Completion Time of Random Candidates and Policies

	Mean	Median	Worst Random	Best Random
GA Improvement	38%	38%	69%	10%

Table 3.1. GA. vs. 300 Random Candidates

As can be seen, the simulation-based predicted completion doesn't always track the results observed on real hardware. The absolute difference ranges from 2–27% for the Sort benchmark and from 0.3–19% for Phase 1 of the Analytics benchmark. However, as mentioned previously in Section 3.2.2, the goal of the lightweight simulator was not to be an accurate predictor of completion time but instead to correctly identify performance differences between different resource allocations. As seen in the figure, the trends in terms of the predicted differences between different scenarios accurately captures the differences seen on real hardware.

Finally, we also wanted to compare TARA's GA-based approach to simply picking random candidates and then using TARA's prediction engine to evaluate them. For this experiment, we randomly picked the same number of candidates (∼300) as evaluated by the GA and compared them to TARA and other polices for the Sort experiment. Figure 3.10 shows the estimated completion time of randomly picked candidates as dots, and result of various policies as lines. It reveals that simple policies could do worse than just picking a resource allocation among a few random candidates. Table 3.1 gives more detailed comparison of the result of GA-based TARA to random candidates. It shows that TARA is still 10% better than the best random candidate. TARA's improvement would be even greater when a job can be clustered within a subset of racks. In contrast, a random allocation would spread machines across all racks and would have an increased chance of encountering a bot-

tleneck link. The overall results presented in this section show that TARA's use of the GA-based search makes it resilient to changes in assumptions about the underlying network topology. This allows it to perform better than both heuristics and randomly chosen candidates.

3.4 Chapter Summary and Discussion

Cloud-based Infrastructure-as-a-Service models are gaining in popularity. However, the potentially huge variations in performance due to the application-unaware resource allocation in these environments is likely to pose a key challenge for their increased adoption. In this chapter, we proposed and evaluated a topology-aware resource allocation solution that addresses this problem.

Our approach derives application-specific information with little manual input (retaining the simplicity of interfaces that have made cloud computing popular) and finds an optimized allocation with low latency and high confidence. In this work, we focused on MapReduce-based data-intensive workloads and build a solution based on a lightweight MapReduce simulator and a genetic-algorithm based search optimization to guide resource allocation. We have developed a prototype of our architecture and demonstrated the benefits of our architecture on a cluster with 80 nodes on a sort and analytics-based benchmark. Our results show that TARA can reduce completion time by up to 59% when compared to simple allocation policies.

In this chapter, we focus on the resource placement problem, from provider's point of view. Overall, as "cloud-based" platforms see wider adoption, future IaaS systems will increasingly need better resource management architectures. We believe that approaches like ours that address this problem without a significant increase in interface complexity and with low overhead will be a key component of future systems. Now, in the next chapter, we will change our position to user's view and consider how the application should use once the resources are allocated.

Chapter 4

Job Scheduling in Heterogeneous Environments

In this chapter, we will consider resource allocation and job scheduling in the Cloud, from the user's or the application's point of view. We assume that the physical locations of VMs are determined by the provider, as studied in Chapter 3. Given that, it is the user's responsibility to determine when to request how many resources of which type, and how to schedule the user's application on the allocated resources.

Particularly, we will focus on heterogeneous environments. In a homogeneous environment where all the machines have the same computing capacity, there is very few things to consider in job scheduling; we only need to consider data locality and network connectivity when making a scheduling decision. To see if multiple jobs receive their fair share of resources, it is enough to count the number of machines assigned to each job. The same applies when the user make a resource request; he or she only need to specify the number machines he want.

However, in a heterogeneous environment, it is important to schedule a job on its preferred resources to achieve high performance. In addition, it is not straight forward to provide fairness among jobs when there are multiple jobs. Moreover, the capacity of different machine types need to be considered when a user make a resource request. To address these issues, we propose an architecture to allocate resources to a MapReduce cluster in the Cloud, and propose a metric of share in a heterogeneous cluster to realize a scheduling scheme that achieves high performance and fairness.

This chapter is organized as follows. We first describe the need for heterogeneous-aware resource allocation and scheduling scheme in Section 4.1. Then we present a resource allocation strategy for a MapReduce cluster in the Cloud in Section 4.2, and a scheduling scheme in heterogeneous and shared environments in Section 4.3. We

illustrate the benefits of our approach with case studies in Section 4.4 and summarize the chapter in Section 4.5.

4.1 Motivation

When we maintain a cluster in the Cloud to serve MapReduce jobs, we want to minimize the cost of running the cluster by keeping the cluster size as small as possible. At the same time, the cluster should be big enough to meet the performance requirements and the job deadlines. As the resource demands for MapReduce jobs fluctuate over time, it is desirable to dynamically adjust the cluster size, which is one of the important features of cloud computing [24]. However, the solution to the problem is not straightforward when the environment is heterogeneous, because we need to determine not only the size of the cluster, but also the type of machines participating in the cluster.

MapReduce workloads have heterogeneous resource demands because some workloads may be CPU-intensive whereas others are I/O-intensive. Some of them might be able to use special hardware like GPUs to achieve dramatic performance gains [73]. It is also likely that the computing environment is heterogeneous. The Cloud consists of generations of servers with different capacities and performance levels; therefore, various configurations of machines will be available. For example, some machines are more suitable to store large amount of data whereas others run faster computations. As the performance of a job depends on where it runs, we need to track *job affinity* (i.e., performance relationship between jobs and machine types) and determine which type of machine offers the most suitable cost-performance trade-off for a job.

Accounting for heterogeneity properties in the Cloud becomes more difficult when the cluster is shared among multiple jobs, because the most suitable type of machine depends on the job. Hence, the MapReduce scheduler should be aware of *job affinity* to make appropriate scheduling decisions. In addition, it is not clear how to define "fair-share" if we want the scheduler to ensure "fairness" among jobs.

In this chapter, we will consider resource allocation and job scheduling problems associated with a MapReduce cluster in the Cloud. Given the fluctuating resource demands of MapReduce workloads, we must scale the cluster according to demands. To that end, we propose a resource allocation strategy that (1) divides machines into two pools - *core* nodes and *accelerator* nodes - and (2) dynamically adjusts the size of each pool to reduce cost or improve utilization.

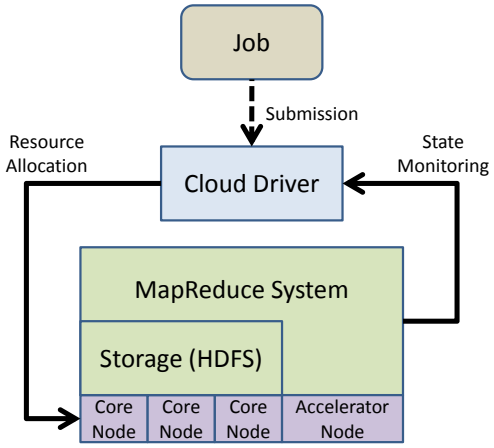

Figure 4.1. Data Analytic Cloud Architecture

In addition, to provide good performance while guaranteeing fairness in shared heterogeneous clusters, we propose *progress share* as a fairness metric to redefine the share of a job in a heterogeneous environment. We will also show how to use it in a MapReduce scheduler.

To present use cases, we use a Hadoop cluster to run MapReduce jobs and Amazon EC2 as the cloud environment. However, the idea of exploiting heterogeneity to improve efficiency and fairness may be applied to other systems running in other environments.

4.2 Resource Allocation

In this section, we will examine the resource allocation strategy to make a MapReduce cluster in the Cloud more efficient. By "efficient" here, we mean minimizing the cost to maintain the cluster while meeting the performance requirements. In other words, we want the cluster big enough to process jobs without violating deadlines, but not so big that we can introduce unnecessary expense for redundant machines. To that end, we propose an architecture with which to run an elastic MapReduce cluster in the Cloud as seen in Figure 4.1.

In this architecture, participating nodes are grouped into one of two pools: (1) long-living *core* nodes to host both data and computations, and (2) *accelerator* nodes

that are added to the cluster temporarily when additional computing power is needed. A MapReduce system (e.g., Hadoop MapReduce) runs on nodes in both pools whereas a storage system (e.g., HDFS) is deployed only on core nodes. This approach is similar to the previous work of Chohan et al. [33], in which spot instances (cheaper but may be terminated without notice, as described in Chapter 2) of Amazon EC2 are added to processing nodes to speed up Hadoop jobs. The *cloud driver* manages nodes allocated to the MapReduce cluster and decides when to add/remove what type of nodes to/from which pool, and how many.

Users submit a job to the cloud driver with a few hints about the job characteristics, including memory requirement, ability to use special features like GPUs, and the deadline, if available. Many production jobs are routinely processed, so the cloud driver keeps the history of job executions to estimate the submission rate of these jobs and update the hints provided. It also monitors the storage system to estimate the incoming data rate. In this way, the cloud driver predicts the resource requirements to process jobs and store data.

The cloud driver is responsible for allocating resources to the cluster. The number of core nodes is determined primarily based on the required storage size. In addition, the cloud driver refers to the history to see if more nodes should be added to the core pool to accommodate the production jobs. When many production jobs with tight deadlines are anticipated or a large ad-hoc job is submitted, the cloud driver will add nodes to the accelerator pool temporarily to handle them rather than allocating too many core nodes that will be underutilized.

When adding nodes, the cloud driver also makes a decision on which resource container (e.g., virtual machine) to use. As an illustration, we examine the case when we use Amazon EC2 [4] for the Cloud. EC2 offers several types of instances. The specification and the cost of EC2 instances is listed in Table 4.1.

Instance Type	$/Hour	Disk(GB)	$/GB/mon	Core	CU	$/CU	Mem(GB)	GB/Core	I/O
m1.small	0.085	160	0.38	1	1	61.20	1.7	1.70	moderate
m1.large	0.340	850	0.29	2	4	61.20	7.5	3.75	high
m1.xlarge	0.680	1690	0.87	4	8	61.20	15	3.75	high
m2.xlarge	0.500	420	2.57	2	6.5	55.38	17.1	8.55	moderate
m2.2xlarge	1.000	850	0.80	4	13	55.38	34.2	8.55	high
m2.4xlarge	2.000	1690	2.56	8	26	55.38	68.4	8.55	high
c1.medium	0.170	350	0.35	2	5	24.48	1.7	0.85	moderate
c1.xlarge	0.680	1690	0.87	8	20	24.48	7	0.88	high
cc1.4xlarge	1.600	1690	0.68	8	33.5	34.39	23	2.88	very high
cg1.4xlarge	2.100	1690	0.89	8	33.5	45.13	23	2.88	very high

Table 4.1. EC2 Instances as of June 2011. CU represents Compute Unit.

If we consider only the cost of the storage, using m1.large instances is the cheapest. However, these instances also have the least computing power among available instance types, so we might need more nodes to accommodate the production jobs. In this case, using other instance types such as c1.medium can be more efficient in terms of the whole expense to store and process the data. Moreover, some jobs may run significantly faster on nodes of a particular instance type (e.g., cg1.4xlarge instances with GPUs). Hence, it is important to know the job/instance type relationship (*job affinity*) to find a good mix of different instances that minimize the cost to maintain the cluster.

We quantize *job affinity* using the relative speed of each instance type for a particular job, which we call *computing rate*(CR). $CR(i, j)$ is the computing rate of instance type i for job j. If $CR(i_1, j)$ is twice that of $CR(i_2, j)$, a task of job j runs twice as fast on i_1 than on i_2, or finishes in half the time. Note that the computing rate is defined within the same job, so it is not useful to compare the fitness of a particular instance type to different jobs. CR is determined by running the job on various instances during the calibration phase of a job execution, which is described in Section 4.3. By using the computing rate information and the cost of each instance type, the cloud driver can make a decision on which instance type to use.

4.3 Scheduling

Once the cloud driver allocates a set of resource units such as virtual machines, the MapReduce system uses the resources that are heterogeneous and shared among multiple jobs. In this section, we consider issues in job scheduling on a shared, heterogeneous cluster, to provide good performance while guaranteeing fairness.

Before presenting our scheduling algorithm, we first summarize terminologies used in this section in Table 4.2 and 4.3. Their relationship is depicted in Figure 4.2.

4.3.1 Share and Fairness

In a shared cluster, providing fairness is one of the most important features that a MapReduce cluster should support. There are many ways to define fairness, but one method might be having each job receive equal (or weighted) share of computing resources at any given moment. In that sense, the Hadoop Fair Scheduler [8] takes the number of slots assigned to a job as a metric of share, and it provides fairness by having each job assigned the same number of slots.

Job	A MapReduce program that processes a particular input data file.
Task	Unit of execution. Each task is responsible for a block of input data file. A task reads and processes a replica of its corresponding block.
File	Logical unit of data stored in a distributed file system. A file is consist of a number of blocks.
Block	Part of a file. The size of a block is typically fixed. A block is replicated by a number of replicas.
Replica	Replication of a block. Replica is actually stored in a distributed file system.
Rack	Physical enclosure of nodes. Nodes in a rack are connected to a top-of-rack switch. Inter-rack communication is more expensive than intra-rack communication.
Node	Physical server that stores replicas and hosts slots to execute tasks.
Slot	Logical unit of computing resources. A slot can execute a task at a time.

Table 4.2. Terminologies used in Chapter 4

Task Category	Location of the Block
Local task	The same machine
Rack-local task	Another machine in the same rack
Remote task	A machine in another rack

Table 4.3. Data Locality

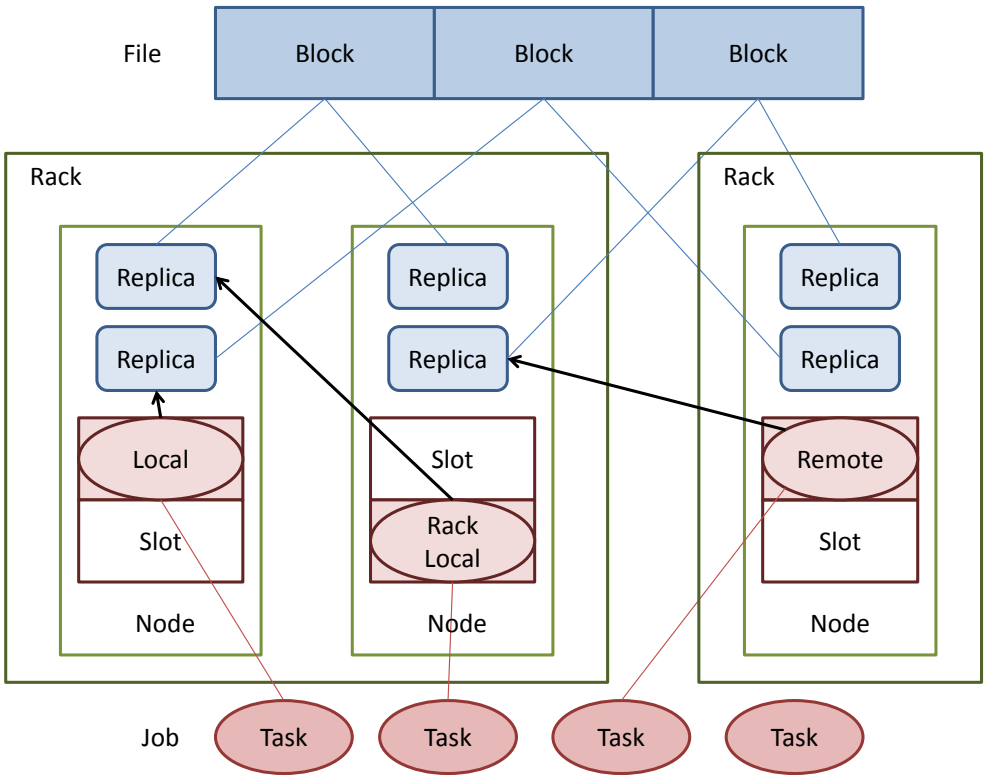

Figure 4.2. Terminologies used in Chapter 4

Algorithm 1 General Fair Scheduler

Require: when slot s on node n becomes available

```
1: sort jobs in increasing order of share
2: for j in jobs do
3:    choose the best task t of job j
4:    if t is good enough then
5:       launch t, then exit
6:    end if
7: end for
```

Algorithm 1 presents a general fair scheduling algorithm, which also describes how Hadoop schedulers operate. In most Hadoop schedulers, the "share" typically represents the number of running tasks. If a task is to read data stored in a particular node, the task is considered the best for the node. Launching the best task right away might not be the best strategy; if there is a chance to find a better place to run the task, some schedulers, such as Delay scheduler [100], allow the task to be skipped.

In a heterogeneous cluster, the number of slots might not be an appropriate metric of the share because all the slots are not the same. Moreover, even on the same slot, the computation speed varies depending on jobs. The performance variance on different resources is also important to consider to improve overall performance of the data analytics cluster; assigning a job to unpreferred slots will not only make the job run slow, but also may prevent other jobs that prefer the slot from utilizing it.

To realize fair and effective job scheduling in a shared and heterogeneous cluster, we introduce *progress share*(PS) that captures the contribution of each resource to the progress of a job. We also show how *progress share* can be used to choose a task to launch.

4.3.1.1 Progress Share

Conceptually, progress share refers to how much progress each job is making with assigned resources (or slots in Hadoop) compared to the case of running the job on the entire cluster without sharing. The progress of a job over time will become slower if the job receives less resources, as seen in Figure 4.3. As the progress share is equivalent to the ratio of progress slope, it is between 0 (no progress at all - no resources received) and 1 (maximum progress - all available resources received).

The computing rate (CR) is used to calculate the progress share of a job. Specifically, given that $CR(s, j)$ is the computing rate of slot s for job j, the progress share

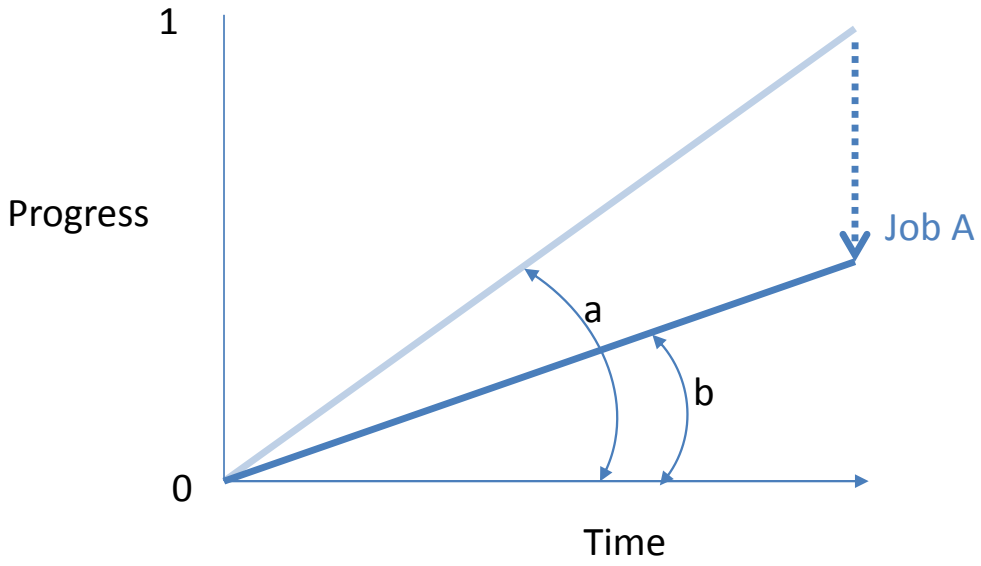

Figure 4.3. Progress Share

$PS(j)$ of job j is defined as follows:

$$PS(j) = \frac{\sum_{s' \in S_j} CR(s', j)}{\sum_{s'' \in S} CR(s'', j)} \tag{4.1}$$

where S_j is a set of slots running tasks of job j and S is the set of all slots. The sum of the progress share for all jobs indicates the effectiveness of the resource assignment. If it is below 1, there is an alternative assignment that makes the sum above or equal to 1.

For example, suppose that we have two jobs, A and B. If they run on a homogeneous cluster, each of them will receive half of the slots on the cluster and the progress will be half as well, as seen in Figure 4.4. As all slots are the same, both of the slot share and the progress share will be 0.5 for each job.

Now suppose that they run on a cluster that consists of two different types of nodes, $N1$ and $N2$, where there are two slots of $N1$ and four slots of $N2$. In addition, assume that a task of job A runs three times faster on a slot of $N1$ than $N2$, whereas job B runs on $N1$ as fast as on $N2$. Figure 4.5 illustrates how they will be scheduled if the cluster runs only one job. White and gray cells represent the slots of $N1$ and

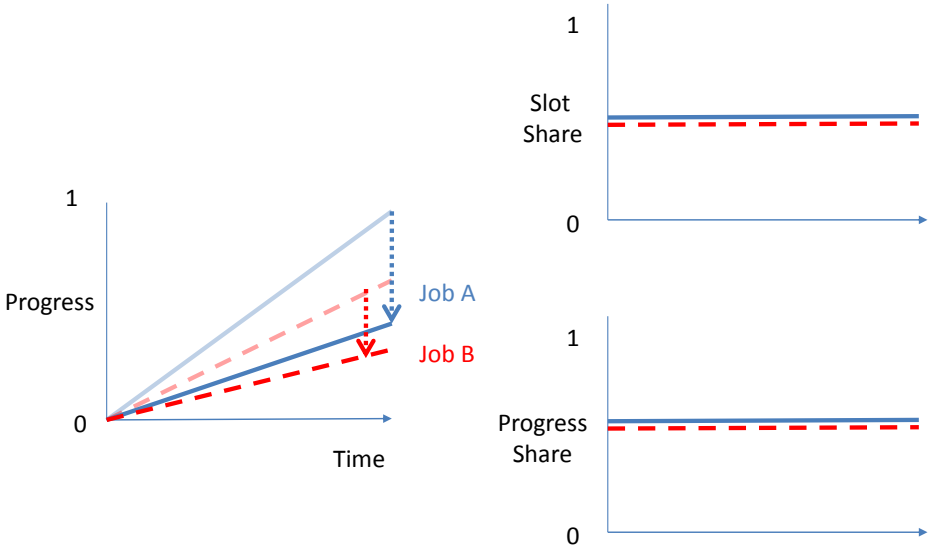

Figure 4.4. Homogeneous Cluster

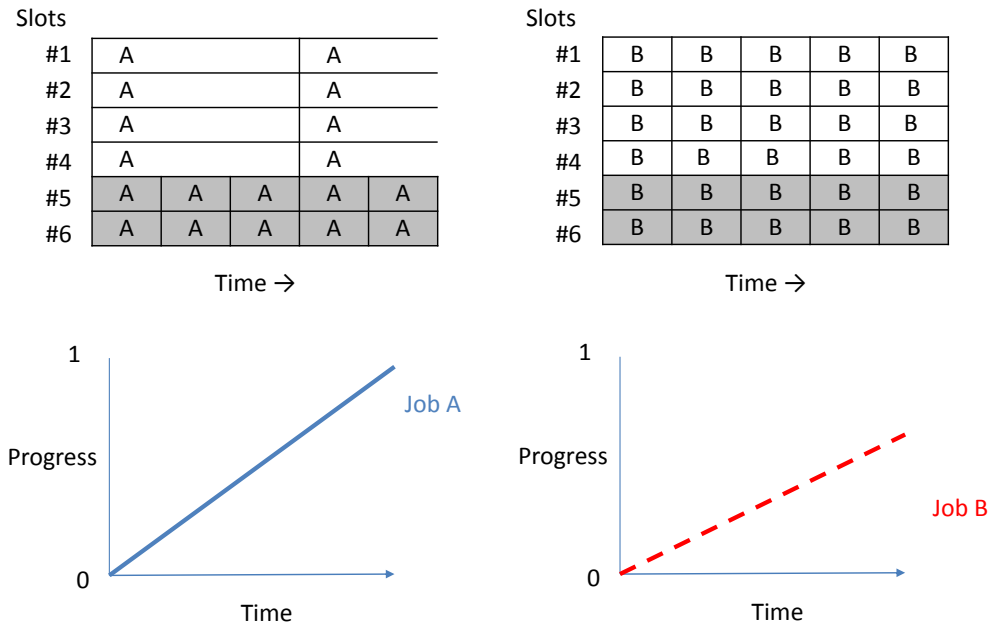

Figure 4.5. Heterogeneous Cluster

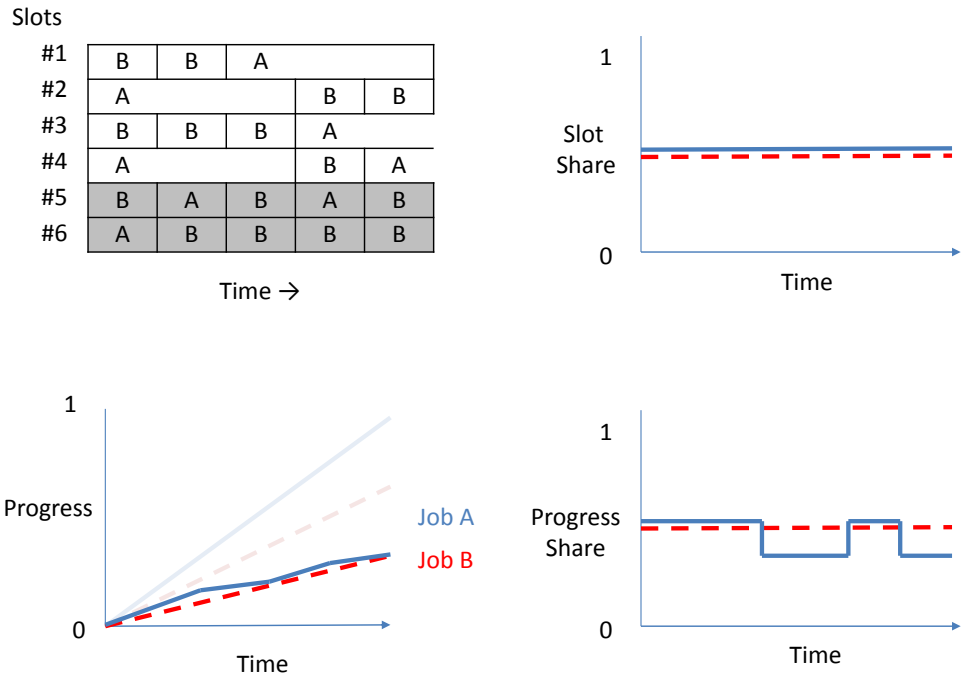

Figure 4.6. Scheduling based on slot share

$N2$, respectively. The letter in each cell indicates the job occupying the slot. The graph below shows the progress of each job over time.

Figure 4.6 is an example of scheduling when the number of assigned slots is used as a share metric. The graph at the bottom-left shows the progress of each job over time. It is drawn only to the point at which there are enough tasks to schedule. Note that the translucent lines describes the progress each job would make if they occupied the whole cluster without sharing. The graphs on the right side track the change in the slot share and the progress share of each job. Even though each job occupies the same number (three) of slots at all times, the progress share of job A often falls below its fair share (0.5) because many tasks of job A run on slots of $N2$, which is not suitable for the job. As a result, job A is making less than half progress compared to the job that occupies the whole cluster. Obviously, job A is under-served in this scenario even though the job received enough slot-share.

In contrast, Figure 4.7 shows an example of progress share-based scheduling. The graph at the bottom-left shows that both jobs A and B are making more progress than they might otherwise. This is because they received more than half progress share all the time, as seen in the graph at the bottom-right. Note that job A sometimes

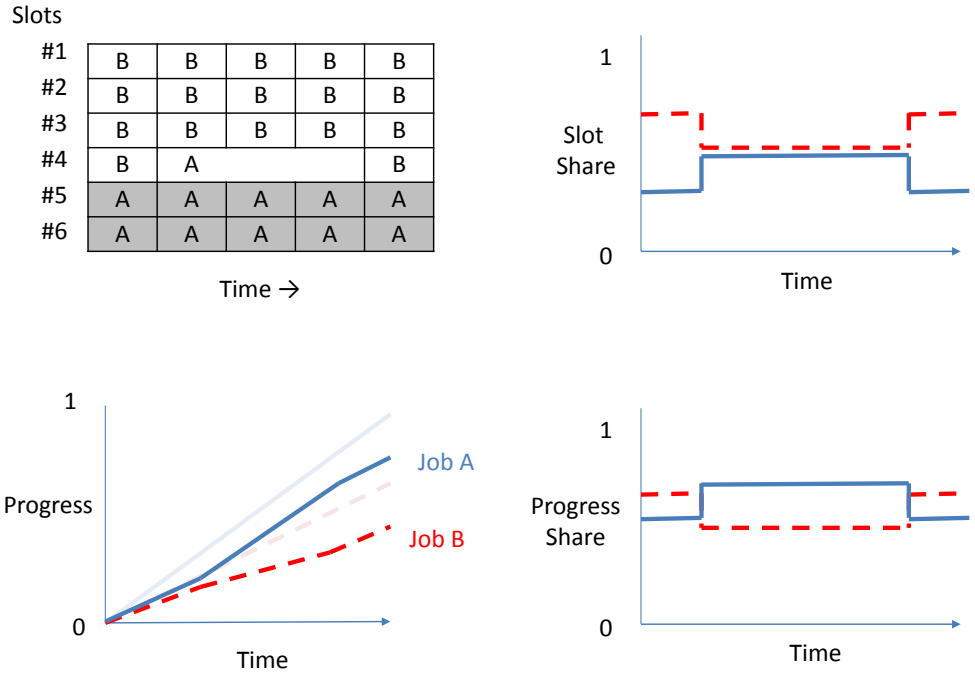

Figure 4.7. Scheduling based on progress share

received less than half slot-share. However, it does not make the job under-served as it received preferred slots. It shows that progress share-based scheduling not only guarantees that each job receives its fair share, but also reduces the job finishing time, thereby improving overall performance.

4.3.1.2 Task Selection

Once a job to be scheduled is selected using the progress share, the scheduler picks the best task of the job to run on the node, which typically has the minimum time to finish. As most Hadoop schedulers assume homogeneous environments, they usually pick a task that accesses the data stored close to the node. In other words, it picks a task in priority order of local, rack-local, and remote. This makes sense in homogeneous environments where it takes roughly the same time to process a task in the same job regardless of the node it runs, so only the time to transfer the data matters. However, in heterogeneous environments where the time to process a task varies depending on nodes, it is sometimes better to pick a remote task that runs faster on the node. Hence, we need to consider both the time required to read and the time needed to process the data, to pick the best task for the node.

Before illustrating our algorithm, we define a few variables. B_j is the input blocks of job j, $\bar{B}_j$ is the blocks not yet assigned in B_j, R_b is a set of replicas of block b, and $block(r)$ is the block that replica r represents.

Choosing a task is equivalent to choosing a block $b \in \bar{B}_j$, then finding a replica $r \in R_b$. By "the best task" we mean the task that runs the fastest on the node. Hence, the scheduler chooses the r that minimizes $T(s, j, r)$, which is the time to finish the task for job j and its block $block(r)$ on slot s. The function $T(s, j, r)$ can be represented as the sum of two sub functions, $Tr(s, r)$ and $Tp(s, j, block(r))$, where the former is the time to read replica r on slot s and the latter is the time to process block $block(r)$ with the task of job j on slot s.

While it is a difficult problem to predict $Tr(s, r)$ and $Tp(s, j, b)$ accurately, it is sufficient to have an approximate value for our purpose of picking a task within a job. Given that the block size is fixed, we can set $Tp(s, j, b)$ to be inversely proportional to $CP(s, j)$. For $Tr(s, r)$, we can simply use a step function that represents local, rack-local, and remote tasks, or use disk/network bandwidth and the block size to calculate it analytically.

4.3.1.3 Launch Decision

After choosing a task, the scheduler should decide whether to launch the task or to skip it. The chosen task is the best for the slot regarding the time to read and process the corresponding input data block among the available ones within the job. However, it can be much faster to run the task on another slot. In other words, there might be another slot $s' \in S$ and replica $r' \in R_b$ where $T(s', j, r') < T(s, j, r)$. In this case, the slot is not the most preferred for job j and block b, even if the block is the most preferred for the slot. Thus, it may make more sense to let a task of another job run on the slot and wait for better slots to become available. Hence, the scheduler needs to consider the following two factors before making a decision: First, how "good" is the slot for the chosen task? Second, how long do we delay scheduling a task for the job to find a better slot?

The first question is to assess the preference of the slot with regard to the given job and the chosen block/replica compared to the other slots. The higher the preference, the more the scheduler is willing to launch the task on the node. Existing schedulers favoring data locality use three levels of preference, in the order of local, rack-local and remote. This is based on the assumption that, for each job, T_p is stationary and T_r has three levels. However, the assumption does not hold in heterogeneous environments

where $T_p(s, j, b)$ differs from slot to slot. Hence, we quantize the preference of a slot to a given job and block with a value between 0 and 1 rather than discrete numbers. To that end, we introduce a function $Pref_{j,b}(s)$ to denote the preference of a slot s for job j and its block b (and the best replica, implicitly). The value is 1 for the most preferred slot and 0 for the least preferred one. We will describe the details of the function later in this section.

Once we have the preference value, we can answer the second question. Basically, the scheduler will wait longer before launching a task on a less preferred slot. Whenever a task of a job is picked to be scheduled, a delay threshold is determined based on the preference of the slot to the job. If the job has been delayed longer than the delay threshold, the slot will be taken (i.e., the task is launched on the slot). Otherwise, the scheduler proceeds to the next job.

Delay scheduler uses pre-defined thresholds for each of the three levels of preference. For example, delay scheduler launches local tasks immediately while it waits five or ten seconds before launching rack-local or remote tasks, respectively. In contrast, in our case, it is difficult to use pre-defined thresholds as the preference is a continuous value rather than a discrete level. Thus, we calculate the threshold in proportion to the preference value with a pre-defined maximum delay. In other words, the slot is taken if the delay made to the job j is less than $Pref_{j,b}(s) * max\_delay$. For example, if the preference value is 0.5, the task is launched only if the delay for the job so far is more than half of the maximum delay.

We put all our scheduling components together and present our heterogeneity-aware MapReduce scheduler in Algorithm 2.

Now the question is how to define the preference function $Pref_{j,b}(s)$. We could take a naïve approach in which the values are evenly distributed in the order of the time required to finish the task, $T(s', j, r')$ for each $s' \in S$ where j is given and $r' \in R_{block(r)}$. However, such an approach does not consider likelihood nor performance gain of finding better slots. For example, if the performance gap between the given slot and better ones is significant, it would be beneficial to wait more. Similarly, if there is a greater chance of finding better slots, the scheduler is more willing to give up the given slot. To take this into account, we define the preference function as follows.

$$Score_j(s, r) = \frac{1}{T(s, j, r)} \tag{4.2}$$

Algorithm 2 Heterogeneity-Aware Scheduler

Require: when slot s on node n becomes available

```
sort jobs in increasing order of progress share
for j in jobs do
    b = argmax_{b' ∈ B̄_j} (BS(s, j, b'))
    if (1 − Pref_{j,b}(s)) ≤ delay(j)/max_delay then
        launch a task for r
    end if
end for
```

$$BS_{j,b}(s) = \max_{r' \in R_b} Score_j(s, r') \tag{4.3}$$

$$Pref_{j,b}(s) = \frac{\sum_{\substack{s' \in S \\ BS_{j,b}(s) \geq BS_{j,b}(s')}} BS_{j,b}(s')}{\sum_{s'' \in S} BS_{j,b}(s'')} \tag{4.4}$$

All pairs of slots and replicas have their own score on each job, represented by $Score_j(s, r)$. $BS_j, b(s)$ (best score) is the highest score that slot s can mark for job j and block b. The preference of the given combination of job, block and slot, $Pref_{j,b}(s)$, is the sum of best scores of less preferred slots than s divided by the sum of best scores of all slots. As only less preferred slots contribute to the preference, the value will be higher if there are fewer chances of having better slots. Also, as a score is proportional to the performance of each combination, the slot will become less preferred if the task will run faster on another slot. In this way, each combination will have weighted preference that reflects the expected performance gain of waiting for better slots.

4.3.1.4 Scheduler

To calculate the *progress share* of each job, the MapReduce system should be aware of the per-slot computing rate (CR). To that end, each job goes through two phases: calibration and normal. When a job is submitted, it starts with the calibration phase. In this phase, at least one map task is launched on each type of node. By measuring the completion time of these tasks, the scheduler can determine the CR. Once the scheduler knows the CR, the job enters the normal phase.

During the normal phase, the scheduler works similar to the Hadoop fair scheduler.

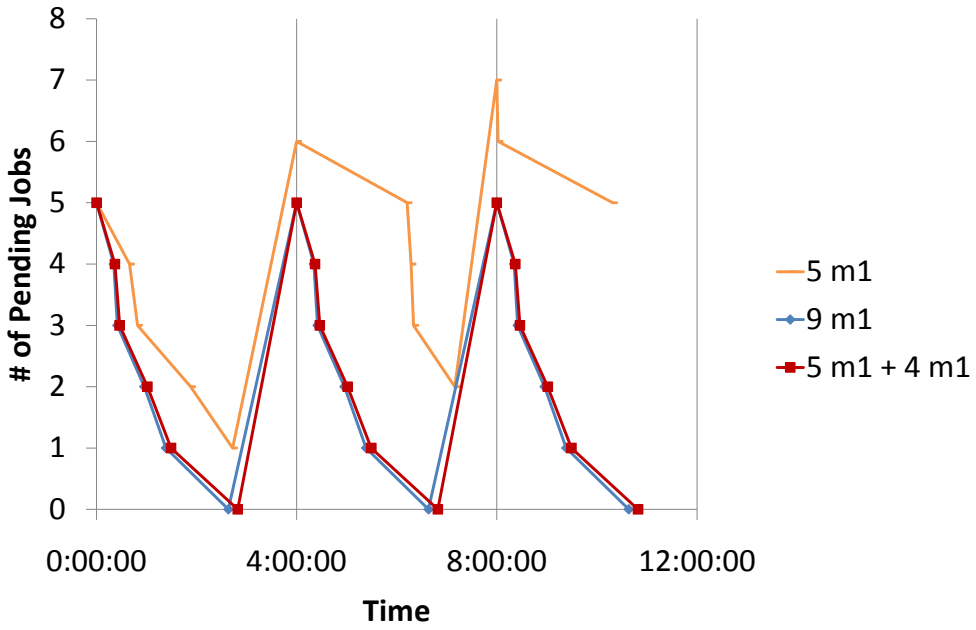

Figure 4.8. Number of pending jobs

When a slot becomes available, a job of which the share is less than its minimum or fair share is selected. However, if there is another job with a significantly higher computing rate on the slot, the scheduler chooses that job to improve overall performance. This is similar to the "delay scheduling" [100] mechanism in the Hadoop fair scheduler.

By using progress share, the scheduler can make an appropriate decision. As a result, the cluster is better utilized (i.e., the sum of the progress share ≥ 1). In addition, each job receives a fair amount of computing resources.

4.4 Case Study

In this section, we examine two case studies to illustrate the potential benefits of our system using Amazon EC2. The first case study is to see the cost-performance trade-offs that can be made in heterogeneous environments. The second one is to validate effectiveness of our progress share based scheduling algorithm.

4.4.1 Resource Allocation and Accelerator Nodes

As the first case study, we consider a situation in which a number of production jobs are issued periodically. To be more precise, We chose 5 jobs from the gridmix2 benchmark [5] as production jobs and have them arrive every 4 hours. To handle these jobs, we prepare clusters with various configurations and see the performance (i.e., the time to complete all jobs) and the cost associated with it.

Figure 4.8 shows the number of pending jobs in the queue over time. In the figure, "5 m1" means using 5 m1.large instances as core nodes. Similarly, "5 m1 + 4 m1" represents a cluster with 5 m1.large core nodes and 4 m1.large accelerator nodes. If we use 5 m1.large instances for core nodes, its capacity is overloaded by the production jobs; thus, the response time of a job keeps increasing over time. Therefore, we need to add more nodes to the core pool, or the accelerator pool. By expanding the core nodes to 9, we can make the cluster powerful enough to accommodate the production jobs. Yet another alternative is to have 4 m1.large accelerator nodes on top of 5 m1.large core nodes. This configuration performs a little less satisfactorily than a 9 m1.large core node configuration, as the accelerator nodes do not take part in the HDFS cluster and have no data-local tasks. However, it still performs reasonably well while saving on the cost because these accelerator nodes are not needed during the last one hour period before another round of production jobs come.

Figure 4.9 shows the number of pending jobs in the queue during one round of the production jobs on the clusters with various configurations. In the figure, "5 m1 + 4 c1" represents a cluster with 5 m1.large core nodes and 4 c1.medium accelerator nodes. As we can see, the jobs finished earlier as more accelerator nodes were added. For example, with three of the accelerator-rich configurations (5m1+8c1, 5m1+12c1, and 5m1+15c1), it took less than 2 hours while it took between 2 and 3 hours with other configurations. In other words, if the jobs have tighter deadlines, it is possible to meet the deadlines with additional expenditure.

Another thing to note is that accelerator nodes can be released after the jobs were done, so using more accelerators can cost less. For example, the cluster with 8 accelerator nodes of c1.medium instance (5m1+8c1) can finish all jobs less than 2 hours, we need to pay for only 2 hours' usage of accelerator nodes. On the other hand, it takes more than 2 hours with 6 c1.medium accelerators (5m1+6c1), which requires paying charges for 3 hours. Figure 4.10 shows the cost to maintain the cluster during one round of the production jobs (4 hours) and the relative speed of processing the

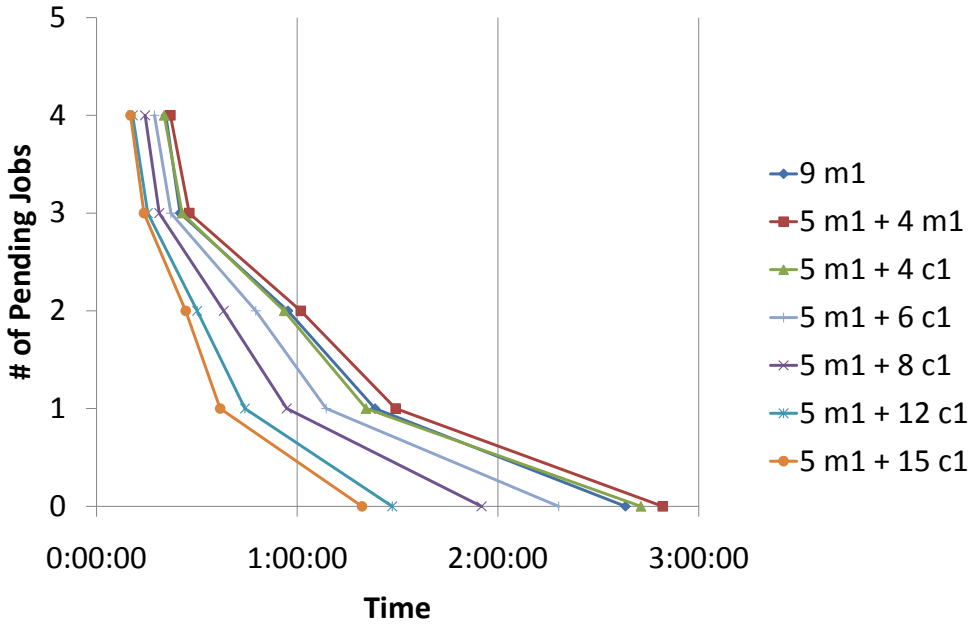

Figure 4.9. Number of pending jobs in one round

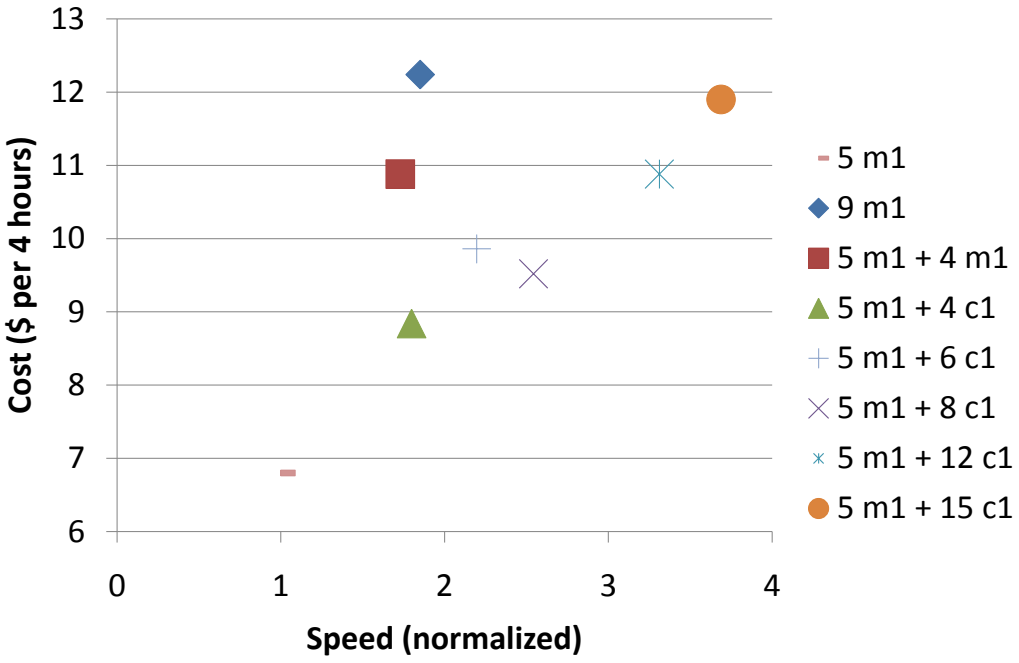

Figure 4.10. Cost and speed of various configuration

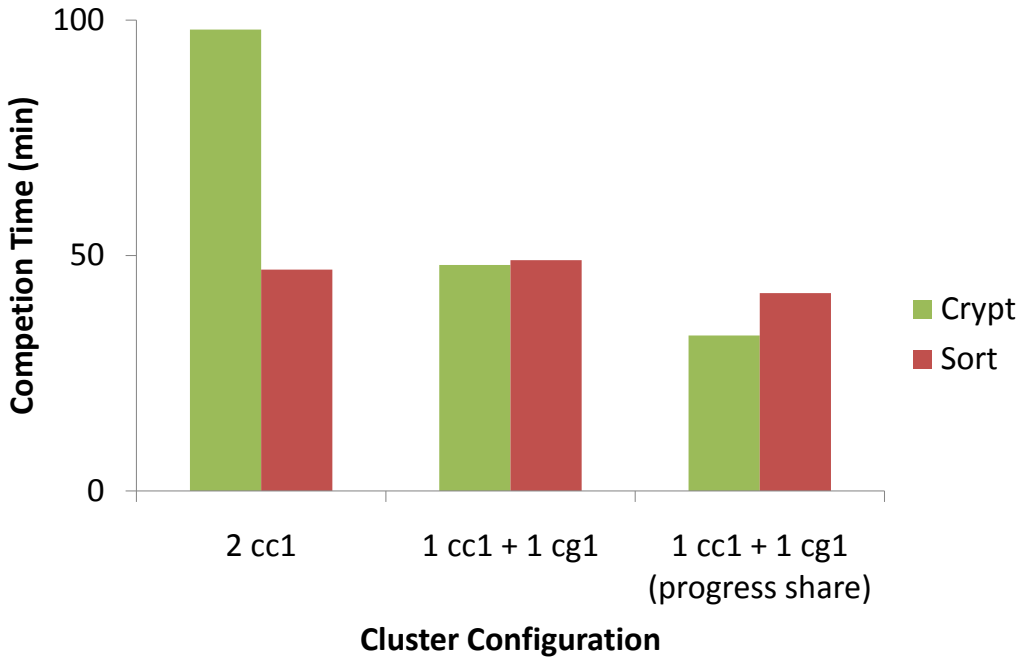

Figure 4.11. Accelerable jobs and the progress share

jobs. As expected, 5m1+6c1 costs more but performs worse than 5m1+8c1. Hence, there is no reason to choose 5m1+6c1 configuration over 5m1+8c1.

Overall, by having accelerator nodes that can be released when the jobs are done, the cost to maintain the cluster is reduced by 11%, from $12.24 (9m1) to $10.88 (5m1+4m1). Moreover, by using c1.medium instances that have faster CPUs at lower prices than do m1.large (see Table 4.1), we can cut the cost further by 28%, to $8.84 (5m1+4c1).

4.4.2 Job Affinity and Progress Share

The second case is when there is significant job affinity. We used a GPU-accelerable dictionary-based cryptographic attack job (Crypto), and a sort job (Sort) for this case. We selected Crypto because it can exploit the GPUs whereas jobs in Gridmix2 cannot. It helps us to highlight the benefit of progress share-based scheduling algorithm. We prepared a small cluster consisting of two nodes and used the fair scheduler.

The first bars in Figure 4.11 show the completion time of each job when the two nodes are all cc1.4xlarge instances. For the second bars, we replaced a node with a cg1.4xlarge instance that is identical to cc1.4xlarge except for the GPUs with which

it is equipped. Even though there is no significant difference for Sort because it is not able to utilize GPU, we can observe a significant performance gain for Crypto, which runs twice faster. This suggests the benefit of having a heterogeneous cluster.

However, just having a heterogeneous cluster does not lead to optimal usage; the scheduler will assign tasks randomly without being aware of job affinity. The last bars show the results with a hardware-aware scheduler that adopts progress share. By prioritizing to the Crypto job to the nodes with GPUs, the job was completed much earlier, about 30% faster than the default scheduler.

It is worth noting that even the sort job finished earlier compared to other cases because the Sort receives more than half of slots at any moment. Assigning a fraction of a cg1.xlarge node to Crypto contributes a significant amount towards its progress share, which in turn makes the Sort job receive more resources to match its progress share to Crypto's. Even if Sort receives more slots than Crypto, we argue that it is more fair than giving the same number of slots to each job because Crypto already receives significantly preferred resources, which quickens its progress.

In this case study, we can see that the scheduling algorithm of the analytics engine plays an important role to improve performance while providing fairness.

4.5 Chapter Summary

The cloud environment is already heterogeneous, and will be more so. Current cloud providers offer heterogeneous resource containers (i.e., VM instances), and as the technology advances, the degree of heterogeneity in the Cloud will be ever increasing. Therefore, it is important for cloud-oriented applications to exploit the heterogeneity. However, many cloud applications assume a homogeneous environment. Even though some take heterogeneity into account, the efforts are usually limited to avoiding failure or performance degradation due to heterogeneity, rather than taking advantage of it.

In this chapter, we presented a system architecture to allocate resources to maintain a MapReduce cluster in a cost-effective manner. By considering various configuration possible in a heterogeneous environment, we could cut the cost of maintaining such a cluster by 28%. In addition, we proposed a scheduling algorithm that provides good performance and fairness simultaneously in a heterogeneous cluster. By adopting progress share as a share metric, we could improve the performance of a job that can utilize GPUs by 30% while ensuring fairness among multiple jobs.

Chapter 5

Performance Prediction in Unpredictable Environments

In previous chapters, we considered initial placement, resource allocation and job scheduling problems in cloud computing environments. Our solutions to those problems are based on models that produce a single point-value prediction (e.g., predicted completion time). However, the dynamic nature of cloud environments makes the performance highly variable and unpredictable. To address this issue, we propose a method to predict the completion time *distribution* with stochastic values, rather than point-values in this chapter. In addition, we will describe how to apply such a prediction method to resource allocation in unpredictable cloud environments.

This chapter is organized as follows. We begin by motivating the need for performance prediction based on stochastic values in Section 5.1. Then we describe the modeling method for MapReduce jobs in Section 5.2. We evaluate the effectiveness of our method in Section 5.3 and present use-cases of the method in resource allocation in Section 5.4. Then we summarize the chapter in Section 5.5.

5.1 Motivation

As seen in previous chapters, the performance prediction of MapReduce jobs is critical to find an optimal placement of virtual machines in cloud environments. It is also important to determine the adequate amount of resources allocated to finish the job in reasonable time. For example, if too few resources are allocated to a MapReduce job that has a certain deadline to finish, the job will not finish in time. Allocating all the available resources is not an option either, as other applications are running concurrently. To make a proper resource allocation decision for a MapReduce

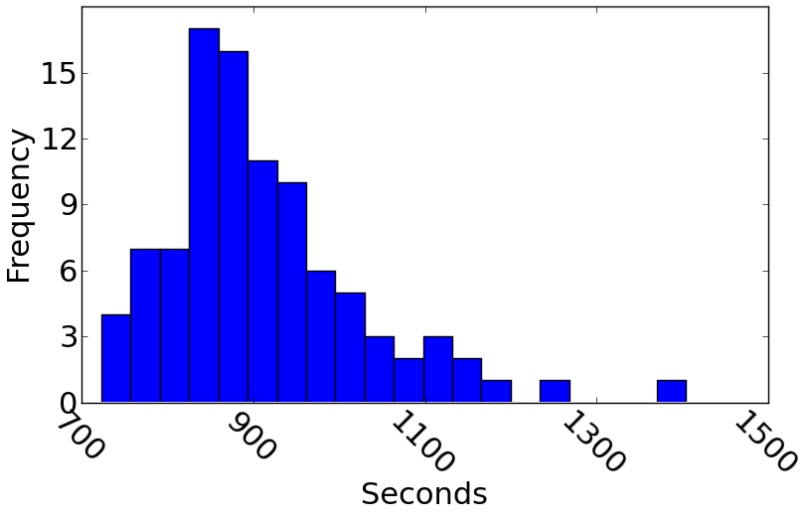

Figure 5.1. Completion Times of 500GB Sort with 100 Workers (100 runs)

job, it is necessary to predict the completion time of the job subject to the resource allocated.

However, the cloud computing environment poses a number of challenges for performance prediction. As depicted in Chapter 1, the impact of multiple applications sharing machines in a cloud makes it hard to predict application performance. The heterogeneity of hardware and resource preemption caused by over-subscription makes it even more unpredictable. MapReduce jobs are particularly susceptible to preemption because many batch jobs have lower priority than services such as web servers. The resulting dynamic and unpredictable natures of cloud environments lead to high variability in the completion time of MapReduce jobs. As a result, the completion time varies even with the same configuration. Figure 5.1 is a histogram of the completion times of 100 executions of a 500G Sort on 100 machines in one of Google's datacenters. As expected, the results are highly variable, ranging from 700 seconds to 1400 seconds.

Previously, there has been a body of effort to predict the completion time of MapReduce jobs, as described in Section 2.2.2 and 2.2.3. However, many of them provide a single point-value (e.g., average) as the predicted completion time of a job. Some of them use multiple values (e.g., best and worst) to predict the completion time, but they are still fixed values and do not indicate a probabilistic distribution of possible completion times.

To provide more useful prediction that accounts for the long-tail distribution of the completion time, we propose a method to predict the completion time distribution. Our prediction method uses stochastic parameters to model MapReduce execution and performs a Monte-Carlo simulation to produce a number of possible outcomes. The results are analyzed to determine the probability distribution of the completion time.

This method can be extended to handle different input sizes, and used in deadline-based worker allocation and online remaining time prediction. In the following sections, we will show that our model can effectively predict the completion time distribution and describe a couple of use cases in which the distribution prediction can be used.

5.2 Modeling a MapReduce job using stochastic values

5.2.1 MapReduce Execution Model

MapReduce [36] is a software framework introduced by Google to support distributed processing on large data sets on a cluster of computers. Basically a MapReduce job goes through the following three phases.

Map phase The map phase consists of a number of map tasks. Each task is responsible for reading a block (split) of the input file, applying a user-defined map function, and producing map intermediate data. Typically the number of map tasks is much larger than the number of workers.

Shuffle phase The shuffle phase consists of a number of shuffle tasks. Each shuffle task is responsible for reading and sorting a subset of map intermediate data hashed on map output keys to that shuffle task and producing shuffle intermediate data. The map and shuffle phases can overlap because shuffle tasks can start reading map intermediate data as soon as any map task finishes.

Reduce phase The reduce phase consists of a number of reduce tasks. Each reduce task has a corresponding shuffle task. A reduce task reads the corresponding shuffle intermediate data, applies user-defined reduce function and produces the final results. Typically the number of shuffle/reduce tasks is set to be a little lower than the number of workers, but sometimes it is larger.

Therefore, if we are able to determine the duration and the scheduling order of

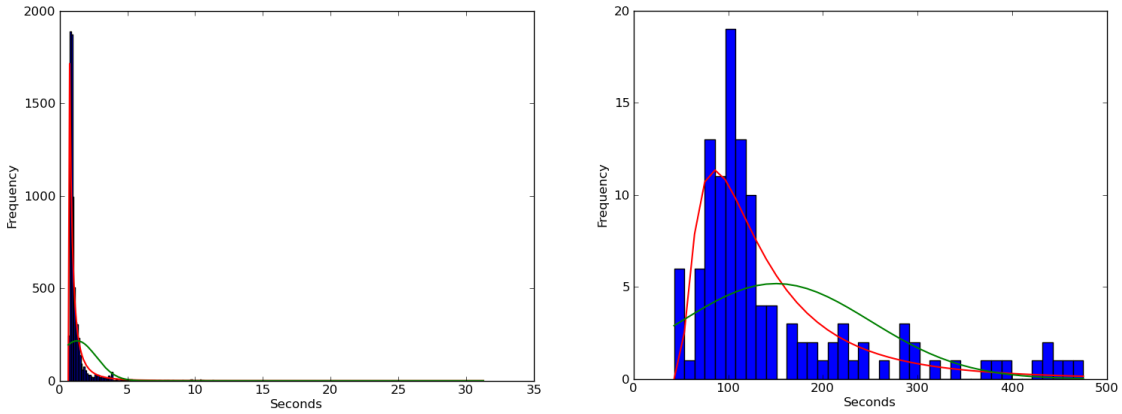

Figure 5.2. Histogram of Map and Reduce Task Durations

each task, we can simulate the execution of the job and estimate the completion time. However, due to the non-determinism in the scheduling and variability of worker performance (i.e., a worker may be co-hosted with other workloads on the same machine), the duration of each task is not static. In other words, they behave differently from execution to execution, even if they apply the same function to the same data on the same set of machines. Hence, it is virtually impossible to simulate the exact execution behavior of a job and come up with an accurate prediction. This motivates us to use stochastic values to parameterize the characteristics (i.e., the duration of each task) of a MapReduce job instead of point values.

To capture the non-determinism of the MapReduce job execution, we first need to model the task durations of each phase, and one of the simplest ways to describe a stochastic value is using a normal distribution. However, observation has shown that the task duration of each phase is actually a long-tail distribution, as shown in Figure 5.2. The green lines represent the fitted normal distribution, while red lines describe the fitted log-normal distribution. We can see that the log-normal distribution fits better than the normal distribution. Because the normal distribution is symmetrical and unsuitable to model long-tails, we use the log-normal distribution to represent the task duration. Among many distributions that can describe a long-tail distribution, we chose the log-normal distribution because it fits the actual distribution better and is easy to manipulate. For example, the Pareto distribution is another long-tail distribution that is widely used. However, it has the maximum frequency at the minimum value, which is different from actual observation. In addition, if we assume that the variables follow the log-normal distribution, it is easy to derive

parameters to describe the distribution (μ and σ). Equation 5.1 shows how they are calculated from the mean and variance of variables.

$$\begin{aligned} \mu &= ln(E[X]) - \frac{1}{2}ln(1 + \frac{Var[X]}{E[X]^2}), \\ \sigma^2 &= ln(1 + \frac{Var[X]}{E[X]^2}). \end{aligned} \tag{5.1}$$

Once we parameterize the durations of each phase using the log-normal distribution, we can estimate the probabilistic distribution of the expected completion time of a job by performing a Monte-Carlo simulation as follows:

1. Generate the duration of each task in a map/shuffle/reduce phase randomly based on the task duration distribution.
2. Schedule the tasks on a pre-defined number of workers using a greedy scheduling algorithm.
3. Find the estimated completion time, which is the last reduce task of the job
4. Repeat 1-3 multiple times to produce a number of outcomes.
5. Analyze the outcomes to get the probability distribution of the completion time.

5.2.2 MapReduce Execution Model - Take Two

However, simply following the simulation steps described above will not estimate the completion time very well in real cloud environment for a few reasons:

Worker arrival time Not all workers become available at the beginning. The worker arrival time is a function of scheduling delays, package distribution delays, invocation delays, and the time required to register with the master.

Idle in shuffle phase The shuffle phase can start as soon as the map phase begins. As shuffle tasks read intermediate data that is generated by map tasks, the first wave of shuffle tasks spends a fair amount of time in an idle state waiting for intermediate data to become available. This introduces a significant skew in the shuffle task duration distribution.

To capture the worker arrival time, we added it to our MapReduce model and simulation step. As shown in Figure 5.3, the worker arrival time follows the log-normal

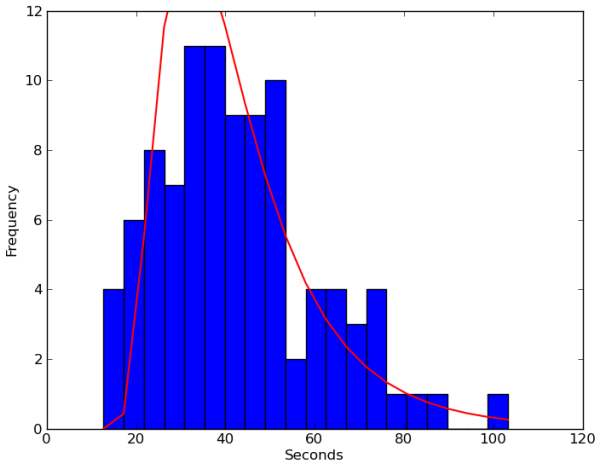

Figure 5.3. Histogram of Worker Arrival Time

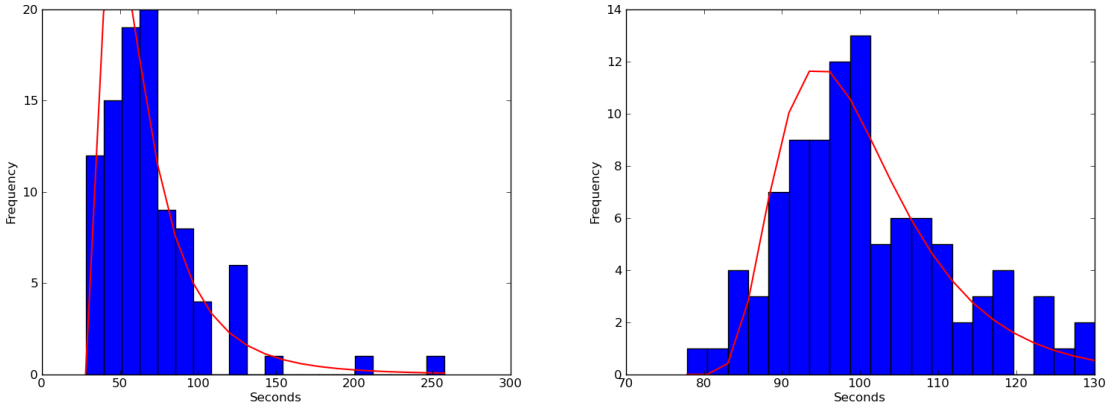

Figure 5.4. Histogram of Shuffle Duration of First and Second Wave

distribution. Hence, we use the same method to model the distribution of worker arrival time as other task durations. In the simulation step, we sample the worker arrival time from the distribution and have workers available when the simulation time passes the arrival time of each worker rather than assuming that all the workers are ready from the beginning.

To address the skew in shuffle durations, we divide the shuffle phase into two waves.

First wave The shuffle tasks in the first wave start before the map phase finishes. Shuffle tasks in the first wave can start as soon as the map phase begins, but they may end up waiting for map intermediate data to become available. Hence, the shuffle

durations of the first wave are longer than the time required to read and process the data.

Second wave The rest of the shuffle tasks occur in the second wave. They start after the map phase finishes and all the map intermediate data become available. It is likely that the shuffle duration of the second wave will be shorter than that of the first wave because no delay is involved. However, if the shuffles lag behind the maps (i.e., the map throughput is greater than the shuffle throughput), there will be no significant difference. It is also worth noting that, if the number of reduce workers is larger than the number of shuffle tasks, there will be no shuffle task in the second wave.

Now we describe the complete steps of our MapReduce execution simulation as shown in the pseudo-code below.

```
w = the number of workers
m = the number of map tasks
r = the number of shuffle/reduce tasks

W = w samples for worker arrival times
M = m samples for map durations
for i = 1 to m                      # for all map tasks
  find worker j where W[j] = min (W) # find an available worker
  W[j]= W[j] + M[i]                  # then schedule the map task

map_end = max(W)                   # map phase finished


S1 = w samples for shuffle durations of the first wave
S2 = r-w samples for shuffle durations of the second wave
R = r samples for reduce durations

for i = 1 to w           # adjust worker timer
   W[j] = map_end

# schedule first wave shuffle tasks and corresponding reduce tasks
for i = 1 to w
   W[i] = W[i] + S1[i] + R[i]

# schedule second wave shuffle tasks and corresponding reduce tasks
for i = 1 to r-w
  find worker j where W[j] = min (W)
  W[j] = W[j]+ S2[i] + R[i+w]
```

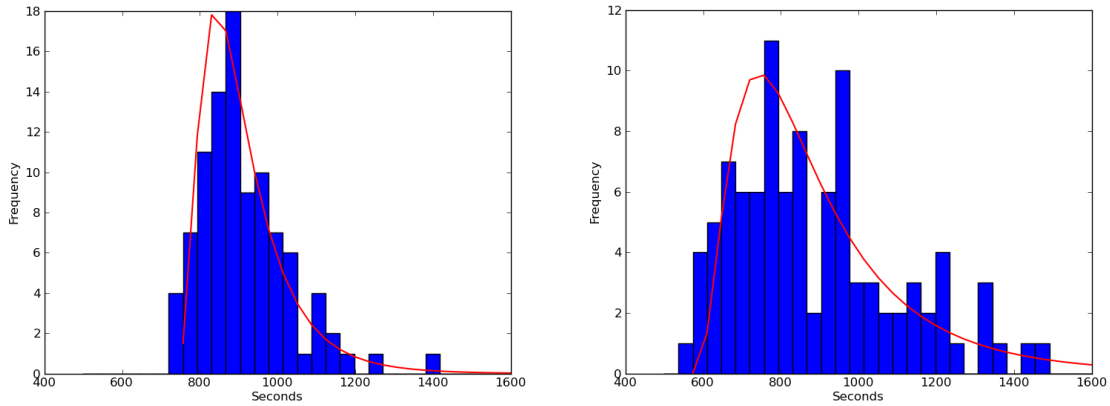

Figure 5.5. Histogram of Actual/Simluated Completion Time

```
simulated_completion_time = max(W)
                          # completion time is when all tasks finish
```

Performing the simulation multiple times yields multiple samples of simulated completion time. To describe the distribution, we take the mean and standard deviation of the completion times.

Figures 5.5 shows the completion time distribution of 100 actual executions (of a 500G/100 workers/120 reduce tasks Sort) and 100 simulated executions based on the statistics derived from the actual executions.

We can see that the completion time distribution also follows the log-normal distribution, and the predicted distribution resembles the actual distribution. However, the actual one is sharper than the predicted one, which leaves room to improve the model in future works.

5.2.3 Model Adjustment

In the previous sub-sections, we assumed that the duration of each phase is available as stochastic values. These values can be extracted from past executions of the same job with the same configuration on the same dataset. However, in practice, a MapReduce job rarely processes the same data even in the case of a production query that is executed repeatedly. Thus, the past history will consist of executions on data of different sizes. Moreover, if the job is a one-time query, there will be no past execution history to reference. In that case, we may want to execute the job

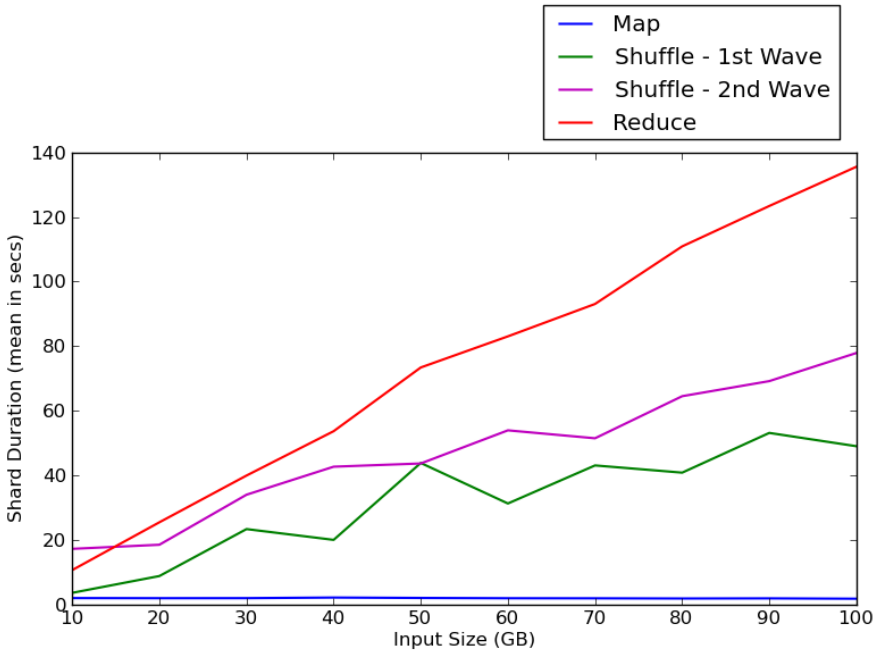

Figure 5.6. Task Duration with Different Input Sizes

on a smaller, trial dataset to build a model and then use the model to predict the performance of the job on an actual dataset that is larger. Hence, in general, we want to adjust the model parameters to apply to the job that processes different data sizes.

When the input data size changes, some of the durations may change while the others may stay the same. For example, when the input size increases, the duration of the reduce tasks will also increase, as the data mapped to each reduce task will be larger. In contrast, the duration of the map tasks will not be affected, as the size of the input split will be the same. Figure 5.6 shows how the task durations of each phase change when the input size varies. As we explained above, the map duration stays fairly constant, whereas the others increase linearly as the input data becomes bigger. Hence, we use linear regression to adjust the parameters in the model to apply the model to jobs that process data of different sizes.

5.3 Evaluation

In this section, we evaluate our model to predict the completion time of MapReduce jobs. We examine three aspects. Firstly, we use the model to predict the performance of the job with the same configuration. Secondly, we will apply the

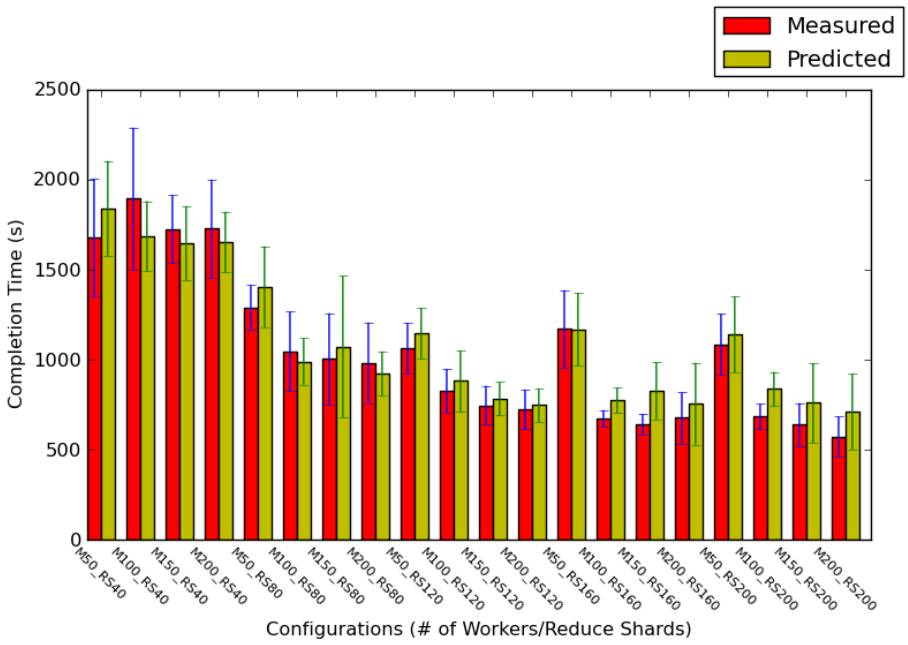

Figure 5.7. Reconstruction of Sort

model to the job run on a different number of workers. Lastly, we will adjust the model to predict the performance of the job with different input data sizes.

For experimental evaluation, we use two benchmarks - Sort and Saw. Sort generates 500G as the input data, processes the data, and produces output data of the same size as the input data. Saw is for processing a 240G fetchlog and producing small output (around 100KB).

5.3.1 Prediction with the same configuration

First, we examine how the model accurately reconstructs the executions. For the Sort benchmark, we use 50/100/150/200 workers with different numbers of reduce tasks (40/80/120/160/200). For Saw benchmark, we set the number of workers as 10/20/40/80/160/320 and the number of reduce tasks as 1/5/10. It turns out that the impact of changing the reduce task size is negligible, so we will just show the case of one reduce task here. We ran 10 experiments for each benchmark and ran a Monte-Carlo simulation with 10 iterations to generate the predicted completion time distribution.

Figure 5.7 shows the results of the Sort benchmark. Various configurations are enumerated along the X-axis. Each configuration is represented by the number of

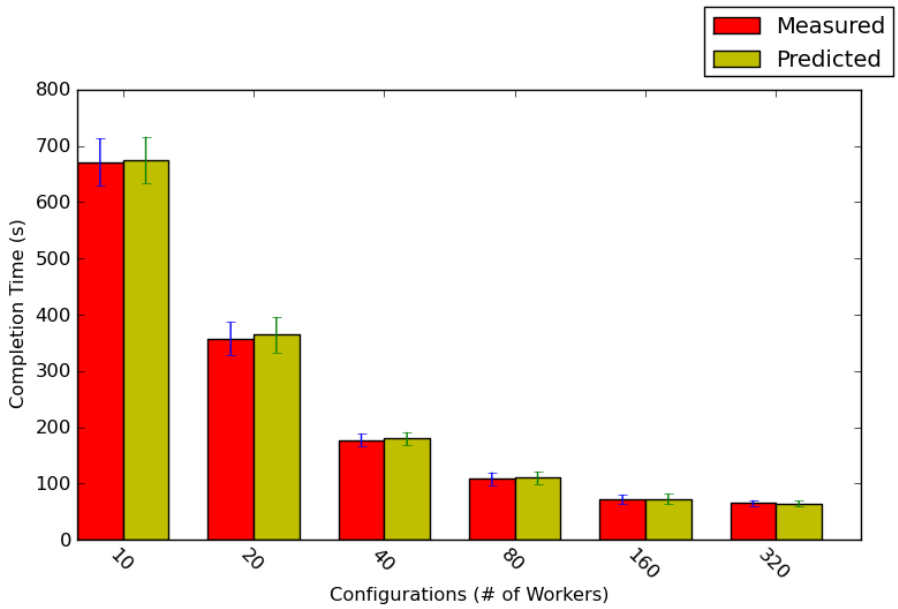

Figure 5.8. Reconstruction of Saw

workers and reduce tasks. For example, "M50_RS40" means that the job ran on 50 workers and the number reduce tasks was set to be 40. For each configuration, the red and yellow bars show the mean completion time of actual and simulated executions, respectively. The standard deviation is presented by the error bar. The results of the Saw benchmark are presented in Figure 5.8.

For both benchmarks, the results show that our simulation model gives us a fairly accurate prediction of the job completion time. In some cases, the prediction works better than in other cases, but overall, it looks accurate enough.

5.3.2 Prediction with different numbers of workers

Next, we examine how well the model can be used to predict the completion time of jobs with different numbers of workers. We use the Sort benchmark for this evaluation. To build the model, we first collect the statistics by using a small number of workers. Specifically, 50 workers are used for a Sort job with 120 reduce tasks. Then, we use the model to predict the completion time distribution of jobs with different numbers of workers.

Figure 5.9 shows the result for the same number of workers (50), whereas Figures 5.10 and 5.11 present the results for 100 and 200 workers, respectively. In each

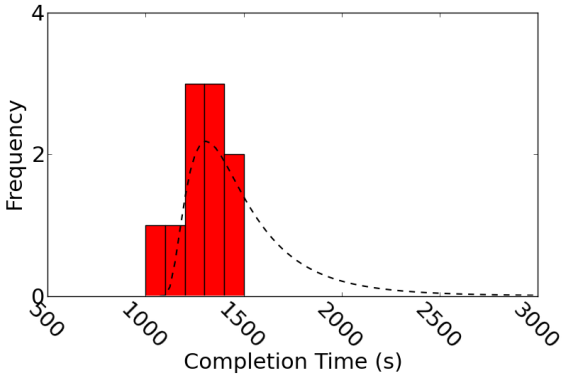

Figure 5.9. Sort with 50 workers

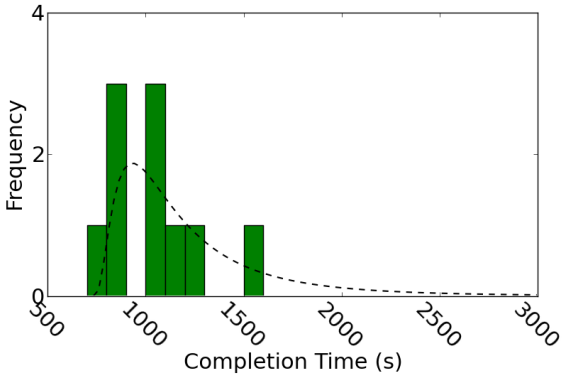

Figure 5.10. Sort with 100 workers

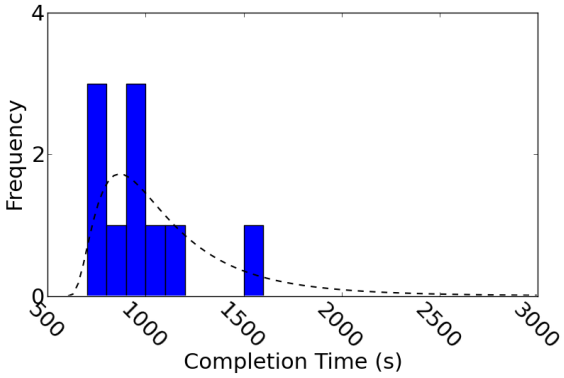

Figure 5.11. Sort with 200 workers

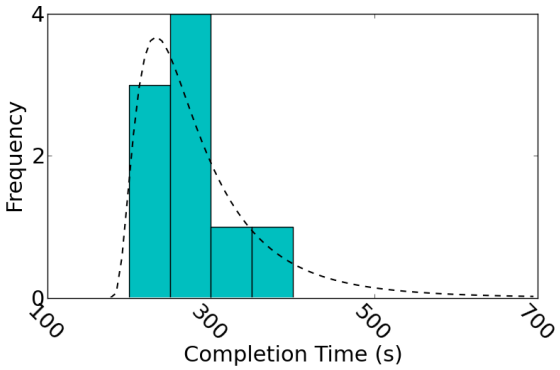

Figure 5.12. Sort for 50G data, interpolated from 10G and 100G

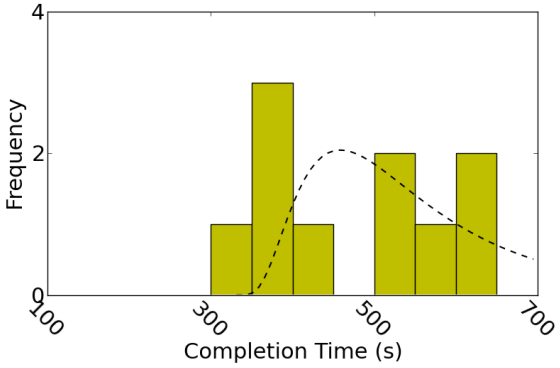

Figure 5.13. Sort for 100G data, extrapolated from 10G and 50G

graph, the histogram is for the measured completion time. The simulated results are fitted into a log-normal distribution, which is drawn with a dotted line. As the number of workers becomes bigger, the histogram moves towards the left, or a shorter completion time.

5.3.3 Prediction with different input sizes

Next, we examined how well the model can be used to predict the completion time of jobs with different input sizes. As we discussed in the previous section, the model must be adjusted for jobs with different input sizes. Specifically, the task durations of the shuffle and reduce phases change as the input size varies. Observations suggest that the relationship between task durations of these phases and the input

size is linear, so we can derive adjusted models for a particular input size from past executions by using interpolation or extrapolation.

Figure 5.12 shows a case of interpolation. The histogram in the figure presents the measured completion time of Sort for 50G input data, while the dotted line represents the predicted completion time distribution. The model for 50G Sort is derived from the statistics of 10G and 100G Sort.

Similarly, Figure 5.13 shows a case of extrapolation, with the histogram of the measured completion times and the dotted graph of the predicted completion time distribution of Sort for 100G input data. The model is derived from the statistics of 10G and 50G Sort.

5.4 Use Cases

In the previous section, we verified that our method is capable of predicting the completion time distribution fairly well. Once the model for a given job is built by referring to past executions or trial executions on smaller datasets, we can use the model to predict the completion time distribution with any number of workers. Because the prediction comes with a probability distribution, we can estimate the chance of finishing the job within a particular timeframe, which is not possible using a single-value prediction. In addition, as our method is based on Monte-Carlo simulation, we can perform a remaining time prediction by beginning the simulation steps from the current state of the job. In this section, we will discuss the benefit of our method by describing several cases in which it can be used.

5.4.1 Worker Allocation

One of the most straightforward use cases of our method is determining the number of workers required to finish a job on time with a certain confidence level. When submitting a MapReduce job, users usually specify the number of workers. Generally, having more workers reduces the completion time of a job because of increased parallelism. However, as users pay for the computing resources in the cloud environment, they need to make a trade-off between the cost and job completion time. To make such a trade-off, users need to predict the job completion time with a certain number of workers to estimate the cost of the resources. The completion time prediction is also necessary to see whether the amount of resources is enough to meet the job deadline. However, due to the variability of the cloud environment, a point-value

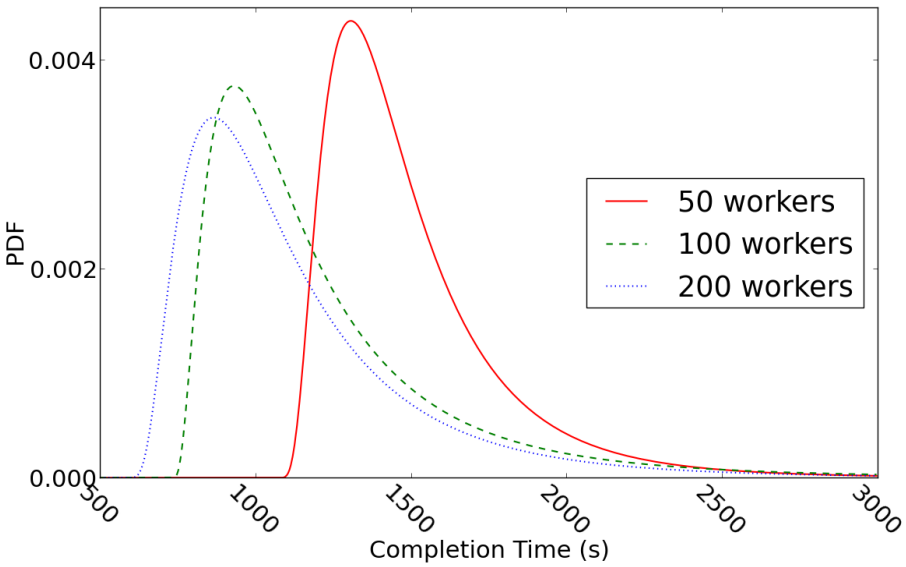

Figure 5.14. Online Prediction

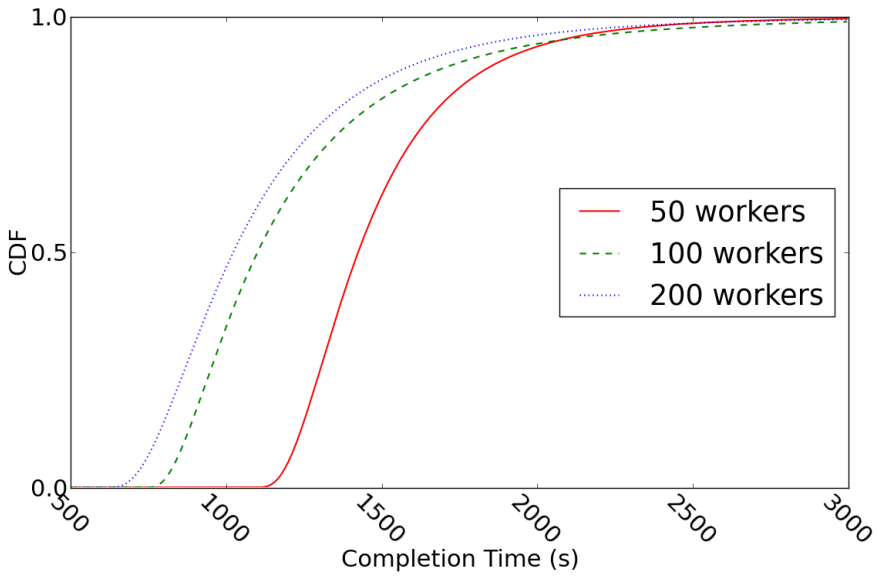

Figure 5.15. Online Prediction

prediction is not effective enough. For example, even if the number of workers was set so that the predicted completion time of a job occurs before the deadline in average cases, there is still a chance to miss the deadline. Users may add a few workers as a buffer, but it will still not be clear how likely it is that the deadline will be met.

With our method, in which users know the probability distribution of the job completion time, it is possible to make trade-off decisions based on the importance of meeting the deadline. We will describe this use case by providing an example. Figure 5.14 shows the probability density function (PDF) of the predicted completion time of a 500G Sort with 50, 100, and 200 workers using solid, dashed, and dotted lines, respectively. The figure shows that having more workers increases the probability of finishing the job earlier. For example, suppose that the job must be completed within 1,400 seconds. We can say that there is a 50-50 chance of meeting the deadline with 50 workers, and the chance rises as we add more workers. The corresponding cumulative distribution function (CDF) in Figure 5.15 gives us a clear idea of this concept. If we add 50 more workers (100 workers in total), the possibility of meeting the deadline will be 80%, which is much better than 50%. With additional 100 workers (200 workers in total), the possibility is slightly higher, but is not the best solution if we consider the cost of the additional workers. Hence, users may want to compare the cost and the likelihood of meeting the deadline, varying the number of workers. Such a sensitive trade-off is possible if the completion time is predicted in distribution.

In addition, this method enables a more sophisticated trade-off based on the performance requirement of a job. Suppose that a job has a certain deadline that is associated with a reward function. The function may indicate how important it is to meet the deadline. For example, if a job has a hard deadline but it does not matter how early it is finished, we can set the function to return a constant reward before the deadline and a large penalty (i.e., negative reward) after the deadline has passed. The following is another example. If we want a job to be finished as soon as possible but there is no specific deadline, we can set the reward function to monotonically decrease. Once such a reward function is specified for a job, users can determine the number of workers needed by comparing the cost of using computing resources until the job is finished and the reward for finishing the job, based on the completion time prediction. This kind of trade-off is particularly useful if a limited number of workers are shared among multiple jobs. By calculating expected outcomes, the user can distribute workers in a way that maximizes the reward.

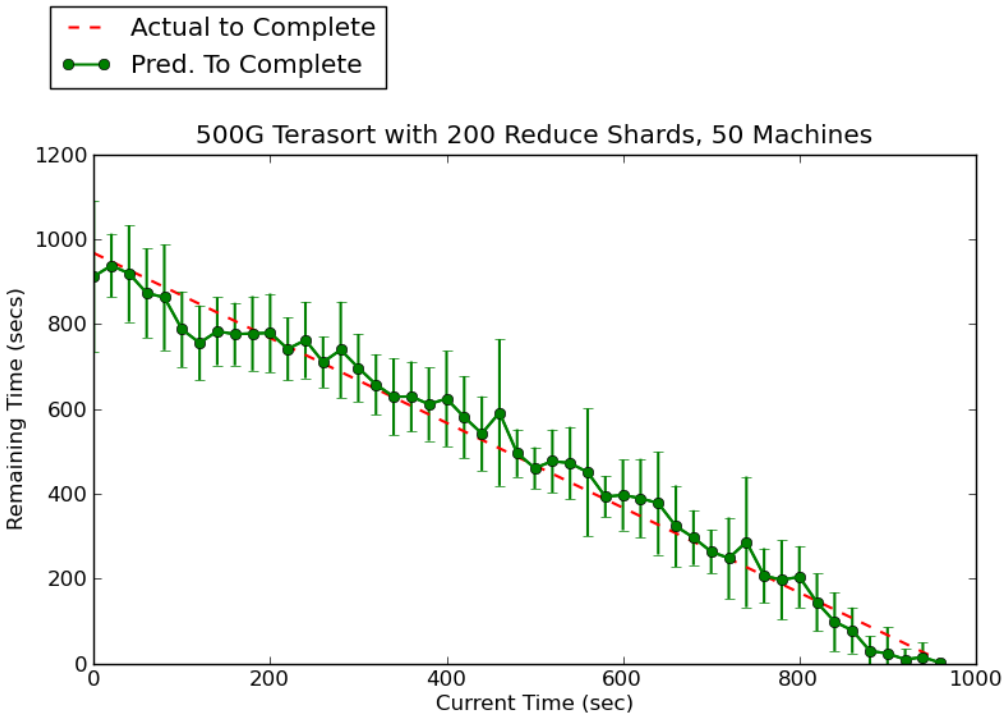

Figure 5.16. Online Prediction for 500G Sort with 200 Reduce Splits on 50 Machines

5.4.2 Online Remaining Time Prediction

Another interesting use-case for our prediction model is the on-line prediction of the remaining time to finish a job. As our method is simulation-based, we can perform a remaining time prediction by beginning the simulation steps from the current state of the job.

Figure 5.16 shows the actual and predicted remaining time of a 500G Sort with 200 reduce splits on 50 machines, over the course of the job execution. The remaining time prediction is performed every 20 seconds. The dashed line follows the actual remaining time that is calculated from the actual completion time. The solid line with circle markers represents the mean of the predicted remaining time at each prediction point, with an error bar that indicates one standard deviation. This gives users not only the expected remaining time but also the probability distribution, allowing users to make a more detailed estimation of the completion time. In addition, it can be used to detect performance anomalies, which occur at the extreme of the distribution.

Going further, such an on-line remaining time prediction can be used to see the effect of changing the number of workers (i.e., adding more workers when some become

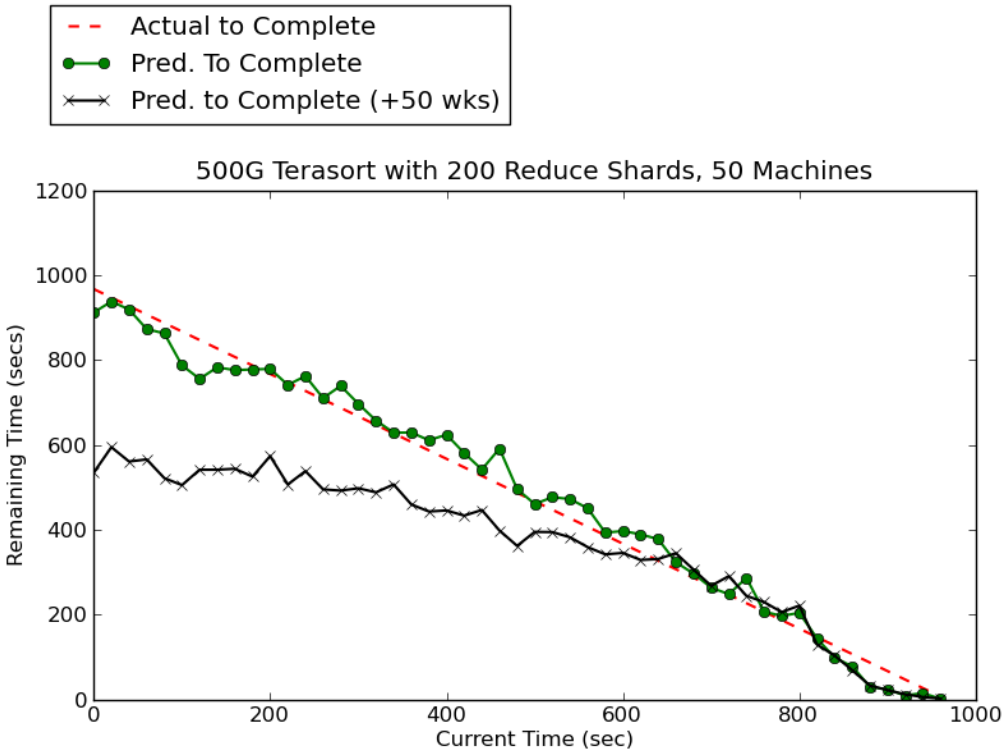

Figure 5.17. Online Prediction for 500G Sort with 200 Reduce Splits on 50+50 Machines

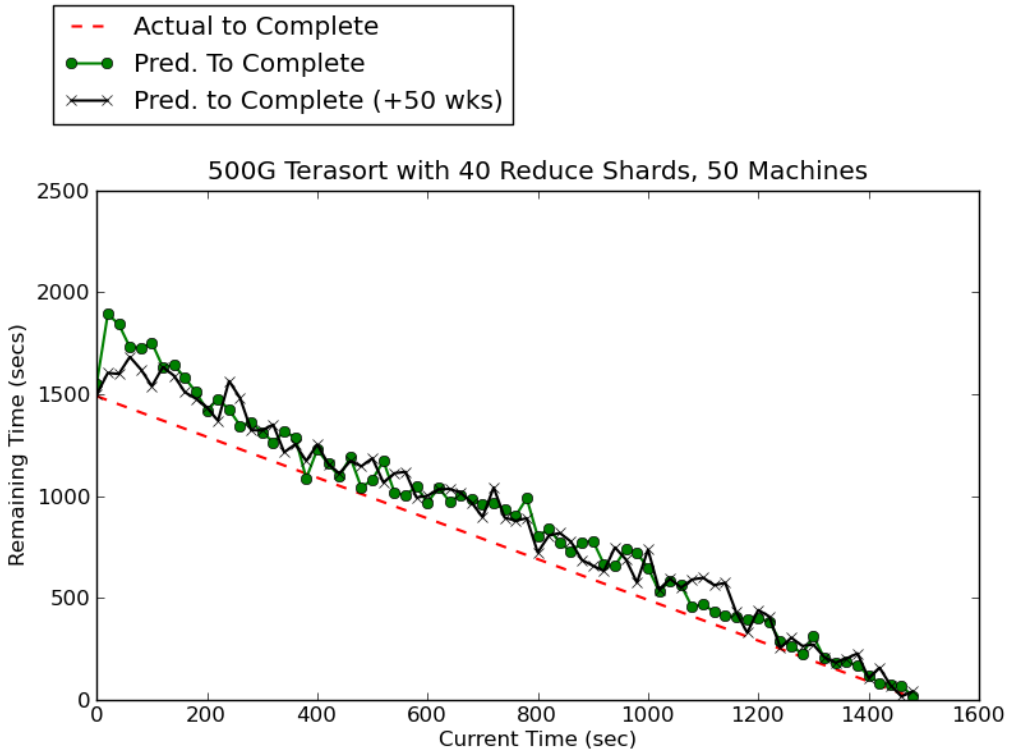

Figure 5.18. Online Prediction for 500G Sort with 40 Reduce Splits on 50+50 Machines

available). In Figure 5.17, The solid line with cross marker is added to indicate the predicted remaining time to finish the job if 50 more workers were added at each moment. Note that error bars are omitted to make the graph readable. As shown in the graph, this enables us to estimate how much benefit (i.e., reduced job completion time) is expected when more workers are added at a certain time. The graph tells us that the benefit of adding more machine is diminished as the time proceeds. For example, the completion time will be about the half if the additional machines are available from the beginning, but the reduction in the completion time is decreasing as the moment of adding more machine goes later. At around 600 seconds the reduction becomes zero, which means that adding more workers does not help in reducing the completion time.

The benefit of adding workers also depends on other parameters of the job. For example, if the job has not enough splits to make use of additional workers, the benefit will be very limited. Figure 5.18 shows the case that the job has 40 reduce splits rather than 200. Because there are not enough reduce splits to fill the workers, the user can expect no benefit of adding workers.

In addition, the user can perform a trade-off analysis on the cost and benefit of changing the number of workers on the fly as it provides the probability distribution. For example, if the user has multiple jobs running concurrently, and a worker becomes available, the user can decide to which job the worker should be assigned based on the benefit of having more workers and the importance (i.e., the scale of reward) of various jobs.

5.5 Chapter Summary

The cloud environment is highly variable and unpredictable, which makes predicting job performance difficult. As seen in the previous chapters, the performance prediction is critical to find an optimal placement of virtual machines in cloud environments and to determine the adequate amount of resources allocated to finish the job in a reasonable time. However, the variability and unpredictability make predictions inaccurate, which may lead to an improper resource allocation and/or scheduling decision.

If it is difficult to monitor the cause of variability and feed it into the performance prediction routine, an alternative might be coming up with the performance prediction associated with a confidence level. In other words, if we can predict the job completion time with various confidence levels, we can pick one according to the tightness of the deadline. For example, if the deadline is hard, we will want to allocate enough resources so that the job is highly likely to finish in time, say, with 90% confidence. In other cases that meeting the deadline is desirable but not critical, we can use the prediction with lower confidence level.

To realize this, we proposed a method to predict the probability distribution of a MapReduce job completion time. Our method can enable a user to predict the distribution fairly well and to make sophisticated trade-off decisions that are difficult when using a point-value prediction.

Chapter 6

Conclusion and Future Research Directions

Two players in cloud computing environments, cloud providers and cloud users, pursue different goals; providers want to maximize revenue by achieving high resource utilization, while users want to minimize expenses while meeting their performance requirements. However, it is difficult to allocate resources in a mutually optimal way due to the lack of information sharing between them. Moreover, ever-increasing heterogeneity and variability of the environment poses even harder challenges for both parties. In this thesis, we addressed these problems of the cloud environments using MapReduce as a target application.

Our contribution is three-fold. The first contribution of this work is a resource placement framework that is application- and resource-aware. This deals with the issues of cloud providers in allocating resources to the user's application efficiently so that the application runs faster with the fixed amount of resources.

The second contribution is a heterogeneity-aware resource allocation and job scheduling scheme for MapReduce workloads. We proposed an overhauled scheduler that considers heterogeneity while maintaining the simplicity of the current scheduler.

The third contribution is a method with which to predict the performance of a job in the form of completion time distribution rather than a single point value. It enables us to allocate resources to meet the deadline with a certain level of confidence, which is difficult to achieve with a point-value prediction.

In this chapter, we summarize our efforts to improve the Cloud and present future research directions.

6.1 Topology-Aware Resource Placement

We proposed and evaluated a topology-aware resource allocation solution in the cloud environment. Our approach derives application-specific information with little manual input and finds an optimized allocation with low latency and high confidence. In this work, we build a solution for resource placement for providers based on a lightweight MapReduce simulator and a genetic algorithm-based search optimization to guide resource allocation. We have developed a prototype of our architecture and demonstrated the benefits of our architecture on a cluster with 80 nodes on a sort and analytics-based benchmark. Our results show that TARA can reduce completion time by up to 59% compared to simple allocation policies.

6.2 Resource Allocation and Scheduling in Heterogeneous Environments

We also presented a system architecture for users to make resource requests in a cost-effective manner, and discussed a scheduling scheme that provides good performance and fairness simultaneously in a heterogeneous cluster, by adopting progress share as a share metric. By considering various configurations possible in a heterogeneous environment, we could cut the cost of maintaining such a cluster by 28%. In addition, we proposed a scheduling algorithm that provides good performance and fairness simultaneously in a heterogeneous cluster. By adopting progress share as a share metric, we were able to improve the performance of a job that can utilize GPUs by 30% while ensuring fairness among multiple jobs.

6.3 Performance Prediction in an Unpredictable Environment

Finally, we proposed a method to predict the probability distribution of a MapReduce job completion time in highly variable and unpredictable cloud environments. Our method can enable a user to predict the distribution fairly well and to make sophisticated trade-off decisions that are difficult when using a point-value prediction. For example, users can choose the number of machines to meet the job deadline with a certain confidence level, which depends on the importance of meeting the deadline.

It also enables users to reason about the benefit of adding more machines while a job is running.

6.4 Future Research Directions

In this thesis, we developed methods to make resource allocation and job scheduling decisions more effective in the Cloud. These methods are based on the current abstraction of interaction between providers and users as follows: the provider offers a set of resource containers with certain specifications, the user makes a resource request that includes the number and the type of resource containers, the provider finds available resources to meet the request and rents them to the users, and then the user runs his application on the allocated resources. However, we realize that the current abstraction of interaction is too limited to exploit the full potential of the Cloud. In this final section of the thesis, we describe some specific future research directions that involve revamping the abstraction.

6.4.1 SLA-based Resource Allocation

Form the users' perspective, it is desirable to use as few resources as possible to minimize their costs, as long as they are sufficient to meet application-level requirements such as service-level agreements (SLA). Suppose that the resources are provided in the form of VMs with a certain resource configuration (e.g., the number of Compute Units and the amount of memory/disk space in Amazon's EC2); the question is how many and what kinds of VMs should be used to meet application-level requirements. Usually, it is the user's responsibility to make such a decision.

To answer the question, several solutions have been proposed. Bodík et al. [28] used machine-learning techniques to derive a model of resource-level requirements to meet SLA and adjust the number of VMs according to the demand. Chang et al. [32] formulated the problem as an optimization problem to determine the number and the types of VMs that minimize the cost.

However, the ultimate concern of the user is to meet application-level requirements. In this sense, the number and the type of VMs does not matter as long as the allocated resources are sufficient to keep the desired level of quality of applications (e.g., job completion time or response time). In other words, if providers are informed of the application-level quality metric, users might delegate the decision to providers. It will also give providers a higher degree of freedom in managing their resources.

6.4.2 VM Migration

VM migration [95] is a valuable tool to balance loads and mitigate resource contention. When multiple applications compete for the same resources, live VM migration techniques can be used to mitigate the problem. However, several conditions should be met to apply VM migration; The VM image should be small enough and in a shared storage. In addition, migration should be done within the same subnet. Otherwise, VM migration is impossible or imposes a significant performance penalty. The cost of VM migration also depends on the application. Hence, it is difficult for providers to tell whether it is applicable to a particular VM. These challenges cause cloud providers to refrain from using VM migration actively.

However, some users might be willing to allow providers to migrate VMs as long as enough incentives are offered. Hence, it will make more sense to let users decide whether VM migration is desirable for their applications and give them incentives (e.g., lower cost, better performance, higher priority, etc.) to opt for enabling migration. This will let providers organize resources more effectively. For example, suppose that some machines with GPUs are occupied by non GPU-accelerable but migration-enabled applications and a large resource request for GPU-equipped machines has arrived. The provider is now able to migrate out these non-GPU-accelerable applications to host GPU-accelerable ones.

6.4.3 Cooperative Scheduling

In most cases, the interaction between providers and users occur as shown in Figure 1.1 in Chapter 1. A user sends a request for resources to a provider, and then the provider looks for the resources to meet the request. Once resources are handed to the user, the provider does virtually nothing in terms of resource allocation. This might not be desirable for both providers and users, especially when the resource availability of the provider changes frequently and the resource requirement of the user is flexible.

Mesos [49] suggests one interesting alternative. Mesos [49] is a platform for running multiple diverse cluster computing frameworks, such as Hadoop, MPI, and web services, on commodity clusters. With Mesos, providers "offer" resources to users, and users decide to either accept or reject the offer. In this way, users can make fine-grained decisions based on their time-varying resource preference while making providers' job simple. One caveat of the current implementation is, though, that this model does not work well unless users return assigned resources frequently due to the

lack of preemption. However, by developing more features that are suitable for the cloud environments, we believe that it could evolve into a cloud-oriented datacenter operating system.

6.5 Summary

In this thesis, we addressed "the cloud resource management problem", which is to allocate and schedule computing resources in a way that providers achieve high resource utilization and users meet their applications' performance requirements with minimum expenditure.

We approached the problem from various aspects, using MapReduce as our target application. From provider's perspective, we proposed and evaluated a topology-aware resource placement solution to overcome the lack of information sharing between providers and users. From user's point of view, we presented a resource allocation scheme to maintain a pool of leased resources in a cost-effective way and a progress share-based job scheduling algorithm that achieves high performance and fairness simultaneously in a heterogeneous cloud environment. To deal with variability in resource capacity and application performance in the Cloud, we developed a method to predict the job completion time distribution that is applicable to making sophisticated trade-off decisions in resource allocation and scheduling.

Our solutions are based on the current abstraction of interaction between providers and users that limits both parties' ability to exploit the full potential of the Cloud. We envision that revamping the abstraction will open up the possibility to solve the problem in a fundamental way.

Bibliography

[1] Open Cirrus cloud computing testbed. http://www.opencirrus.org/.

[2] Arm - the architecture for the digital world. http://www.arm.com.

[3] Intel atom processor. http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html.

[4] Amazon ec2. http://aws.amazon.com/ec2.

[5] Gridmix. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[6] Hadoop. http://hadoop.apache.org, .

[7] Capacity scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html, .

[8] Hadoop fair scheduler. http://hadoop.apache.org/common/docs/r0.20.1/fair_scheduler.html, .

[9] Memristors. http://http://www.memristor.org.

[10] Early experiments in cloud computing. http://cloudstorming.wordpress.com/tag/amazon-s3-ec2-new_york_times-nasdaq/.

[11] Dileep bhandarkar on datacenter energy efficienty. http://perspectives.mvdirona.com/2011/2/20/DileepBhandarkarOnDatacenterEnergyEfficiency.aspx.

[12] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[13] Self-service, prorated supercomputing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

[14] Phase-change memory. http://en.wikipedia.org/wiki/Phase-change_memory.

[15] Pharmaceuticals test-drive the cloud. http://www.datacenterknowledge.com/archives/2009/05/26/pharmaceuticals-test-drive-the-cloud/.

[16] Solid state storage 101: An introduction to solid state storage. SNIA White Paper, January 2009.

[17] V. A. F. Almeida, I. M. M. Vasconcelos, J. N. C. rabe, and D. A. Menasc. Using random task graphs to investigave the potential benefits of heterogeneity in parallel systems. In *ACM/IEEE conference on Supercomputing*, 1992.

[18] Virglio Almeida and Daniel Menasc. Cost-performance analysis of heterogeneity in supercomputer architectures. In *ACM/IEEE conference on Supercomputing*, 1990.

[19] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using mantri. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '10)*, 2010.

[20] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Parveen Patel, and Ion Stoica. What slows datacenter jobs and what to do about it. RADLab Retreat, Tahoe City, CA, January 2010.

[21] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[22] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *22nd ACM Symposium on Operating Systems Principles(SOSP)*, October 2009.

[23] Michael Armbrust and Gunho Lee. Evaluating amazon's ec2 as a research platform. http://radlab.cs.berkeley.edu/w/upload/0/0c/EC2_Performance_v2.ppt.pdf.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion

Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[25] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, 2003.

[26] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[27] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 171, april 2004.

[28] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *HotCloud'09: Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 12–12, 2009.

[29] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Blni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, Jun, 2001.

[30] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and et al. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2008.

[31] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[32] Fangzhe Chang, Jennifer Ren, and Ramesh Viswanathan. Optimal resource allocation in clouds. *Cloud Computing, IEEE International Conference on*, pages 418–425, 2010.

[33] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Chandra Krintz. See spot run: Using spot instances for mapreduce workflows. In *Workshop on Hot Topics in Cloud Computing (HotCloud '10)*, 2010.

[34] Condor Project. `http://www.cs.wisc.edu/condor/`.

[35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, CA, December 2004.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, 2004.

[37] Mathijs den Burger, Thilo Kielmann, and Henri E. Bal. TOPOMON: A monitoring tool for grid network topology. In *Proceedings of the International Conference on Computational Science*, pages 558–567, Amsterdam, Netherlands, April 2002.

[38] A. Ganapathi, Yanpei Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87 –92, march 2010. doi: 10.1109/ICDEW.2010.5452742.

[39] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM Symposium on Operating Systems Principles(SOSP)*, 2003.

[40] Globus Toolkit. `http://www.globus.org/`.

[41] P. Brighten Godfrey and Richard M. Karp. On the price of heterogeneity in parallel systems. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 84–92, 2006.

[42] David E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[43] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of*

the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 51–62, Barcelona, Spain, August 2009.

[44] Albert Greenberg et al. VL2: A scalable and flexible data center network. In *proc-sigcomm*, pages 51–62, 2009.

[45] Chona S. Guiang, Kent F. Milfeld, Avijit Purkayastha, and John R. Boisseau. Memory performance of dual-processor nodes: comparison of intel xeon and amd opteron memory subsystem architectures. In *Proceedings for ClusterWorld Conference and Expo 2003*, 2003.

[46] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A scalable and fault-tolerant network structure for datacenters. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 75–86, Seattle, WA, August 2008.

[47] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Chen Tian Yunfeng Shi, Yongguang Zhang, and Songwu Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74, Barcelona, Spain, August 2009.

[48] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *5th Conference on Innovative Data Systems Research (CIDR '11)*, January 2011.

[49] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Network Systems Design and Implementation(NSDI*, 2011.

[50] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.

[51] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, MT, October 2009.

[52] Michael Isard et al. Dryad: Dsitributed data-parallel programs from sequential building blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[53] Michael A. Iverson, Fusun Ozguner, and Gregory J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. *High-Performance Distributed Computing, International Symposium on*, 0:263, 1996. ISSN 1082-8907.

[54] Michael A. Iverson, Füsun Özgüner, and Lee Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Trans. Comput.*, 48(12):1374–1379, 1999.

[55] Jeffrey M. Jaffe. An analysis of preemptive multiprocessor job scheduling. *Mathematics of Operations Research*, 5(3):pp. 415–421, 1980.

[56] Joyent Smart Computing. `http://www.joyent.com/`.

[57] Tim Kaldewey, Theodore M. Wong, Richard A. Golding, Anna Povzner, Scott A. Brandt, and Carlos Maltzahn. Virtualizing disk performance. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 319–330, St. Louis, MO, April 2008.

[58] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, San Diego, CA, June 2009.

[59] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86, 2010.

[60] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, 2005.

[61] Shyh-Chang Lin, W.F Punch, and E.D.Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 28–37, 1994.

[62] Chuang Liu, Lingyun Yang, Ian T. Foster, and Dave Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages 63–72, Edinburgh, Scotland, July 2002.

[63] G.Q. Liu, K.L. Poh, and M. Xie. Iterative list scheduling for heterogeneous computing. *Journal of Parallel and Distributed Computing*, 65(5):654 – 665, 2005.

[64] M. Maheswaran, S. Ali, H.J. Siegal, D.; Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99)*, 1999.

[65] Juan C. Martinez, Lixi Wang, Ming Zhao, and S. Masoud Sadjadi. Experimental study of large-scale computing on virtualized resources. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing (VTDC '09)*, pages 35–42, Barcelona, Spain, June 2009.

[66] Richard McClatchey, Ashiq Anjum, Heinz Stockinger, Arshad Ali, Ian Willers, and Michael Thomas. Data intensive and network aware (DIANA) grid scheduling. *Journal on Grid Computing*, 5(1):43–64, 2007.

[67] Peter Mell and Timothy Grance. The nist definition of cloud computing. *NIST Special Publication 800-145*.

[68] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: http://doi.acm.org/10.1145/1807167.1807223. URL `http://doi.acm.org/10.1145/1807167.1807223`.

[69] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *proc-nsdi*, 2010.

[70] Mumak: Map-Reduce Simulator, January 2010. `http://issues.apache.org/jira/browse/MAPREDUCE-728`.

[71] Dan Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source

cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pages 124–131, Shanghai, China, May 2009.

[72] Owen O'Malley and Arun C. Murthy. Winning a 60 second dash with a yellow elephant, April 2009. `http://sortbenchmark.org/Yahoo2009.pdf`.

[73] Jord Polo, David Carrera, Yolanda Becerra, Vicen Beltran, and Jordi Torres andEduard Ayguad. Performance management of accelerated mapreduce workloads in heterogeneous clusters. In *39th International Conference on Parallel Processing (ICPP2010)*, 2010.

[74] Rackspace Cloud, January 2010. `http://www.rackspacecloud.com/`.

[75] Barath Raghavan, Kashi Venkatesh Vishwanath, Sriram Ramabhadran, Ken Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*, pages 337–348, Kyoto, Japan, August 2007.

[76] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. *SIGPLAN Lisp Pointers*, VII(3):65–78, 1994.

[77] Vinay J. Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *In Passive and Active Measurement Workshop*, 2003.

[78] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security*, Chicago, IL, November 2009.

[79] Rumen: a tool to extract job characterization data from job tracker logs, January 2010. `http://issues.apache.org/jira/browse/MAPREDUCE-751`.

[80] S. Ratna Sandeep, M. Swapna, Thirumale Niranjan, Sai Susarla, and Siddhartha Nandi. Cluebox: A performance log analyzer for automated troubleshooting. In *Proceedings of the 1st USENIX Workshop on the Analysis of System Logs, (WASL '08)*, San Diego, CA, December 2008.

[81] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *Proceedings of the 11th International Joint Conference*

on Measurement and Modeling of Computer Systems (SIGMETRICS), pages 299–310, 2009.

[82] Cipriano A. Santos, Akhil Sahai, Xiaoyun Zhu, Dirk Beyer, Vijay Machiraju, and Sharad Singhal. Policy-based resource assignment in utility computing environments. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '04)*, pages 100–111, Davis, CA, November 2004.

[83] J.M. Schopf and F. Berman. Performance prediction in production environments. In *12th International Parallel Processing Symposium (IPPS 1998)*, pages 647 –653, mar-3 apr 1998.

[84] Ozan Sonmez, Hashim Mohamed, and Dick Epema. Communication-aware job placement policies for the KOALA grid scheduler. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 79, Amsterdam, Netherlands, December 2006.

[85] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the 1st USENIX Workshop on the Analysis of System Logs, (WASL '08)*, San Diego, CA, December 2008.

[86] Xueyan Tang and S.T. Chanson. Optimizing static job scheduling in a network of heterogeneous computers. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 373 –382, 2000.

[87] Eno Thereska and Gregory R. Ganger. Ironmodel: robust performance models in the wild. In *proc-sigmetric*, pages 253–264, Annapolis, MD, June 2008.

[88] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *GCC '09*, pages 218–224, 2009.

[89] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering energy proportionality with non energy-proportional systems – optimizing the ensemble. In *proc-hotpower*, San Diego, CA, December 2008.

[90] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems*

Design and Implementation (OSDI), pages 271–284, Boston, MA, December 2002.

[91] Abhishek Verma, Ludmila Cherkasova, and Roy Campbell. Slo-driven right-sizing and resource provisioning of mapreduce jobs. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*, September 2011.

[92] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Proceedings of the 17th International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, London, UK, September 2009.

[93] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[94] Rich Wolski, James S. Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational grid. *International Journal of High Performance Computing Applications*, 15(3):258–281, 2001.

[95] Timothy Wood, Prashant Shenoy, Arun Venkataramani, , and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *4th USENIX Symposium on Networked Systems Design and Implementation*, pages 229–242, 2007.

[96] Jaehyung Yang, Ishfaq Ahmad, and Arif Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. *Parallel Processing, International Conference on*, 1:219–226, 1993.

[97] Chee Shin Yeo and R. Buyya. Service level agreement based allocation of cluster resources: Handling penalty to enhance utility. In *Cluster Computing, 2005*, 2005.

[98] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI '08)*, pages 29–42, San Diego, CA, December 2008.

[99] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, University of California, Berkeley, April 2009.

[100] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.