Wright State University
CORE Scholar

Browse all Theses and Dissertations | Theses and Dissertations

2011

Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds

Fengguang Tian
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Sciences Commons

Repository Citation
Tian, Fengguang, "Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds" (2011). *Browse all Theses and Dissertations*. 511.
https://corescholar.libraries.wright.edu/etd_all/511

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by

Fengguang Tian
B.E., Beijing University of Posts and Telecommunications, 2007

2011
Wright State University

Wright State University
SCHOOL OF GRADUATE STUDIES

November 18, 2011

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Fengguang Tian ENTITLED Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Keke Chen, Ph.D
Thesis Director

Mateen Rizki, Ph.D
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Keke Chen, Ph.D

Bin Wang, Ph.D

TK Prasad, Ph.D

Andrew Hsu, Ph.D
Dean, School of Graduate Studies

ABSTRACT

Tian, Fengguang. M.S., Department of Computer Science and Engineering, Wright State University, 2011. *Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds.*

Running MapReduce programs in the public cloud introduces the important problem: how to optimize resource provisioning to minimize the financial charge for a specific job? In this thesis, We study the whole process of MapReduce processing and build up a cost function that explicitly models the relationship between the amount of input data, the available system resources (Map and Reduce slots), and the complexity of the Reduce function for the target MapReduce job. The model parameters can be learned from test runs with a small number of nodes on a small amount of data. Based on this cost model, we can solve a number of decision problems, such as the optimal amount of resources that can minimize the financial cost with a time deadline or minimize the time under certain financial budget. Experimental results show that this cost model performs well on tested MapReduce programs.

List of Symbols

m number of map slots of system 13
r number of reduce slots of system 13
$\mathbf{M}$ number of chunks of input data 13
$\mathbf{R}$ user-configured number of reduce slots 13
$\mathbf{T}$ total time 16
b size of data block 14
k keys of data 15
b_R total amount of data in each Reduce 15
Φ_r overall cost of the Reduce process 16
Θ cost of managing the Map and Reduce 16
β_i parameters describing the constant factors 17
T_1 another form of total time 17
T_2 another form of total time 18
u price of renting one VM instance 19
ϕ financial budget 20
γ number of slots per node for VM instance 19
α_i parameters describing the factors 20
R^2 measure of regression modeling 27
C_i real cost of the test 28
$\hat{C}_i$ estimated cost of the test by the trained model 28

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Related Work . . . 3

2 Preliminary **6**
2.1 Hadoop and MapReduce . . . 6
2.1.1 Introduction to Hadoop . . . 6
2.1.2 Architecture of HDFS . . . 7
2.1.3 Introduction to MapReduce . . . 8
2.2 Amazon Web Service . . . 9

3 Cost Model of MapReduce **11**
3.1 Factors Affecting the Performance of MapReduce . . . 11
3.2 Analysis of the MapReduce Process . . . 12
3.2.1 Map Process . . . 14
3.2.2 Reduce Process . . . 15
3.2.3 Putting All Together . . . 16
3.3 Optimization of Resource Provisioning . . . 18

4 Experiments **22**
4.1 Experimental Setup . . . 22
4.1.1 Hardware and Hadoop Configuration for Inhouse Cluster . . . 22
4.1.2 AWS cluster configurations . . . 23
4.1.3 Datasets . . . 23
4.1.4 MapReduce Programs for Testing . . . 24
4.1.5 Experiments Strategy . . . 25
4.2 Result Analysis . . . 26
4.2.1 Regression Model Analysis . . . 26

5 Conclusion and Future Work **33**
5.1 Conclusion . . . 33
5.2 Future Work . . . 33

Bibliography 34

List of Figures

2.1 Architecture of Hadoop . . . 7
2.2 Architecture of HDFS . . . 8

3.1 Components in Map and Reduce tasks and the sequence of execution. . . . 12
3.2 Illustration of parallel and sequential execution in the ideal situation. . . . 14

4.1 Trend for WordCount on Inhouse Cluster. . . . 29
4.2 Trend for WordCount on AWS Clusters. . . . 29
4.3 Trend for TeraSort on Inhouse Cluster. . . . 30
4.4 Trend for TeraSort on AWS Cluster. . . . 30
4.5 Trend for Join program on Inhouse Cluster. . . . 30
4.6 Trend for Join program on AWS Cluster. . . . 30
4.7 Trend for PageRank on Inhouse Cluster. . . . 30
4.8 Trend for PageRank on AWS Cluster. . . . 30
4.9 Fitting the model for WordCount on Inhouse Cluster(60 rounds). . . . 31
4.10 Fitting the model for WordCount on AWS Clusters (30 rounds). . . . 31
4.11 Fitting the model for TeraSort on Inhouse Cluster(60 rounds). . . . 31
4.12 Fitting the model for TeraSort on AWS Cluster(30 rounds). . . . 31
4.13 Fitting the model for Join program on Inhouse Cluster(40 rounds). . . . 31
4.14 Fitting the model for Join program on AWS Cluster(30 rounds). . . . 31
4.15 Fitting the model for PageRank on Inhouse Cluster(25 rounds). . . . 32
4.16 Fitting the model for PageRank on AWS Cluster(15 rounds). . . . 32

List of Tables

4.1 Result of regression analysis 27
4.2 Average Relative Error Rates Result 29

Acknowledgement

Foremost, my sincere gratitude goes to my advisor Prof. Keke Chen for receiving me to join the lab and for his kindness, motivation, expertise, and most of all, for his patience. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Master study.

Besides my advisor, I would like to thank the rest of my thesis committee members: Prof. Bin Wang and Prof. TK Prasad, for their insightful comments, and hard questions.

I would like to thank my lab mates in DIAC Group: Shumin Guo and Huiqi Xu. Also I would like to thank my friends in Wright State University: Wenbo Wang, Lu Chen and Ke Qin.

Dedicated to

my dear parents, Guoren Tian and Zhurong Guo; my brother, Fenglin Tian

without whom it would never have been accomplished

Introduction

1.1 Overview

Data Intensive Computing in the Cloud. With the deployment of web applications, scientific computing, and sensor networks, a large amount of data can be collected from users, applications, and the environment. For example, user click through data has been an important data source for improving web search relevance [11] and for understanding online user behaviors [28]. Such datasets can be easily in terabyte scale; they are also continuously produced. Thus, an urgent task is to efficiently analyze these large datasets so that the important information in the data can be promptly captured and understood. As a flexible and scalable parallel programming and processing model, recently MapReduce [5] (and its open source implementation Hadoop) has been widely used for processing and analyzing such large scale datasets [23, 9, 22, 13, 4, 17].

On the other hand, data analysts in most companies, research institutes, and government agencies have no luxury to access large private Hadoop/MapReduce clouds. Therefore, running Hadoop/MapReduce on top of the public cloud has become a realistic option for most users. In view of this requirement, Amazon has developed the Elastic MapReduce[1] that runs on-demand Hadoop/MapReduce clusters on top of Amazon EC2 nodes. There are also scripts[2] for users to manually setup Hadoop/MapReduce on EC2 nodes.

[1] aws.amazon.com/elasticmapreduce/.
[2] e.g., wiki.apache.org/hadoop/AmazonEC2

Running a Hadoop cluster on top of the public cloud shows different features from a private Hadoop cluster. First, for each job a dedicated Hadoop cluster will be started on a number of virtual nodes. There is no multi-user or multi-job resource competition happening within such a Hadoop cluster. Second, it is now the user's responsibility to set the appropriate number of virtual nodes for the Hadoop cluster. The optimal setting may differ from application to application and depend on the amount of input data. To my knowledge, there is no effective method helping the user make this decision.

The problem of optimal resource provisioning involves two intertwined factors: the cost of provisioning the virtual nodes and the time to finish the job. Intuitively, with a larger amount of resources, the job can take shorter time to finish. However, resources are provisioned at cost. It is tricky to find the best setting that minimizes the cost. With other constraints such as a time deadline or a financial budget to finish the job, this problem appears more complicated.

Scope of Our Research and Contributions. In this thesis, we develop a method to help the user make the decision of resource provisioning for running the MapReduce programs in public clouds. This method is based on the proposed MapReduce cost model that has a number of parameters to be determined for a specific application. The model parameters can be learned with tests running on a small number of virtual nodes and small test data. Based on the cost model and the estimated parameters, the user can find the optimal setting by solving certain optimization problems.

Our approach has several unique contributions.

- Different from existing work on the performance analysis of MapReduce program, our approach focuses on the relationship among the number of Map/Reduce slots, the amount of input data, and the complexity of application-specific components. The resulting cost model can be represented as a linear model in terms of transformed variables. Linear models provide robust generalization power that allows one to determine the parameters with the data collected on small scale tests.

- Based on this cost model, we formulate the important decision problems as several optimization problems. The resource requirement is mapped to the number of Map/Reduce slots; the financial cost of provisioning resources is the product between the cost function and the acquired Map/Reduce slots. With the explicit cost model, the resultant optimization problems are easy to formulate and solve.

- We have conducted a set of experiments on in-house hadoop cluster and the Amazon Cloud to validate the cost model. The experimental result shows this cost model fits the data collected from four tested MapReduce programs very well. The experiment on model prediction also shows low error rates.

The entire thesis is organized as follows. In Chapter 1, we introduce the cloud computing, the scope of our research and also the related work on MapReduce performance analysis is briefly discussed. In Chapter 2, we introduce the Hadoop and MapReduce Programming framework, also we introduce the Amazon Web Service (a useful commercial public cloud). In Chapter 3, we analyze the processes of MapReduce program and build the cost model, also the aforementioned decision problems on resource provisioning are formulated as several optimization problems based on the cost model in this chapter. In Chapter 4, we present the experimental results that validate the cost model. In Chapter 5, we conclude our contribution and the accuracy of our model, also we discuss the future work we need to do.

1.2 Related Work

The recent research on MapReduce has been focused on understanding and improving the performance of MapReduce processing in a dedicated private Hadoop cluster. The configuration parameters of Hadoop cluster are investigated in [9, 1] to find the optimal configuration for different types of job. In [31], the authors simulate the steps in MapReduce

processing and explore the effect of network topology, data layout, and the application I/O characteristics to the performance. Job scheduling algorithms in the multi-user multi-job environment are also studied in [35, 25, 36]. These studies have different goals from our work, but an optimal configuration of Hadoop will reduce the amount of required resources and time for jobs running in the public cloud as well. A theoretical study on the MapReduce programming model [14] characterizes the features of mixed sequential and parallel processing in MapReduce, which justifies our analysis in Section 3.2.

MapReduce performance prediction has been another important topic. Kambatla et al. [12] studied the effect of the setting of Map and Reduce slots to the performance and observed different MapReduce programs may have different CPU and I/O patterns. A fingerprint based method is used to predict the performance of a new MapReduce program based on the studied programs. Historical execution traces of MapReduce programs are also used for program profiling and performance prediction in [15]. For long MapReduce jobs, accurate progress indication is important, which is also studied in [19]. Kristi Morton[20],[18] introduces a PARALLAX PROGRESS ESTIMATOR, which can estimate the performance by estimating the remaining time of MapReduce pipelines based on the time had elapsed. The key strategy the author used is dividing the processes of MapReduce into five key pipelines and estimate the remaining time based on the time had elapsed. Another strategy used by [12, 15] and shared by our approach is to use test runs on small scale settings to characterize the behaviors of large scale settings. However, these above approaches do not study an explicit cost function that can be used in optimization problems. Guanying Wang, in his two papers [30],[29], gives another approach to predict the performance, which is that after indicates the key factor affect the performance, the author implements a simulator,MRPerf, on top of ns-2,and which capture aspects of the four key factors, and use these informations to predict the performance of the MapReduce without running in the real MapReduce platform. These two approaches indicate the key factors affect the performance, which is shared by our approach, but also does not support the

explicit cost function.

In addition to the recent studies on MapReduce parallel programming model, several other techniques have been widely used and studied in parallel computing and graph computing community. Wilhelm et al [33] gives an overview of time analysis techniques for a number of topics, these analysis techniques include Static Program Analysis,Simulation, Abstract Processor Models, Integer Linear Programming and so on. Also the author discusses two basic approaches: static methods and measurement-based methods. Our approach is a measurement-based method. Smith [27] uses historical information to categorize applications and builds prediction models for each category. The category for a new application is identified by the similarity between the application and the category. Similar approaches were used by [24, 7, 6] to predict resource usage and application performance in grid computing. Instance-based learning techniques are used to learn the performance model based on similar historical job execution [26]. The prediction models are very useful for resource provisioning. For example, Maleeha Kiran [16] uses an execution time prediction module to help users find the best resource configuration for efficiently running a specific application.

Preliminary

2.1 Hadoop and MapReduce

2.1.1 Introduction to Hadoop

Hadoop is a large-scale distributed batch processing infrastructure built on commodity computers, and it is designed to run the distributed processing of large massive internet data sets across clusters of commodity computers using a simple programming model based on google's paper [5]. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements a computational paradigm named MapReduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both MapReduce and the Hadoop Distributed File System are designed so that node failures are automatically handled by the framework[1]. Figure 2.1[2] shows the commodity hardware architecture of Hadoop.

A Hadoop cluster has one master node (and its backups) and a number of slave nodes. The master node manages MapReduce jobs and the HDFS storage system. It runs three services: Namenode, JobTracker and SecondaryNamenode. Each slave receives commands

[1] http://wiki.apache.org/hadoop/
[2] http://61.153.44.88/apache/hadoop/HadoopApacheConEu07.pdf

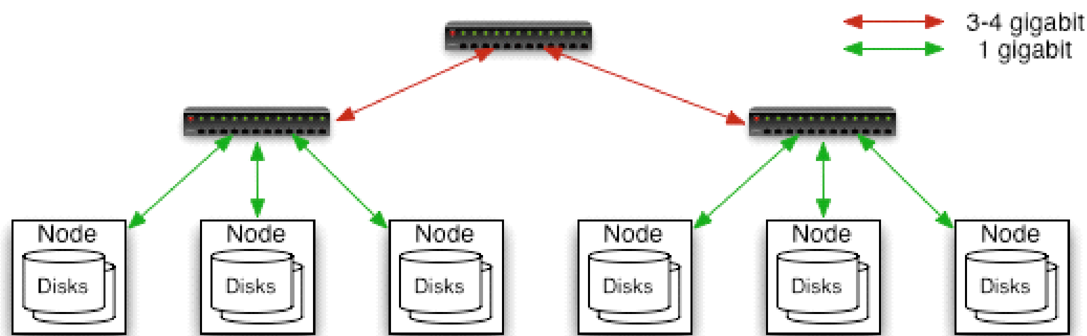

Figure 2.1: Architecture of Hadoop

from the master node to manage and process data. A slave node runs two services: Datanode and TaskTracker. The Namenode and Datanode services are a part of the HDFS system. The Namenode controls the allocation of the data and manage the Datanode to store the data. JobTracker and Tasktracker services work together to execute MapReduce jobs. JobTracker manages the MapReduce job, balances the workload of each node, assigns tasks to TaskTrackers, and monitors the status of tasks. TaskTracker accepts commands, executes the assigned tasks, reports the status, and outputs the result.

2.1.2 Architecture of HDFS

HDFS is a distributed file system designed to run on commodity hardware. HDFS is a block-structured file system: every files are split into blocks of a fixed size (default: 64MB), and these blocks are stored across the DataNodes of a cluster, into one or more machines with data storage capacity. A file is not necessarily stored on the same machine, usually it is stored across several DataNodes with several duplicates (default 2), and the target DataNode which hold each block are chosen randomly on a block-by-block basis. To access to a file, it require the NameNode to cooperate with the multiple DataNode machines[34]. The figure 2.2[3] shows the architecture of HDFS.

[3] http://hadoop.apache.org/common/docs/current/hdfs$_design.html$

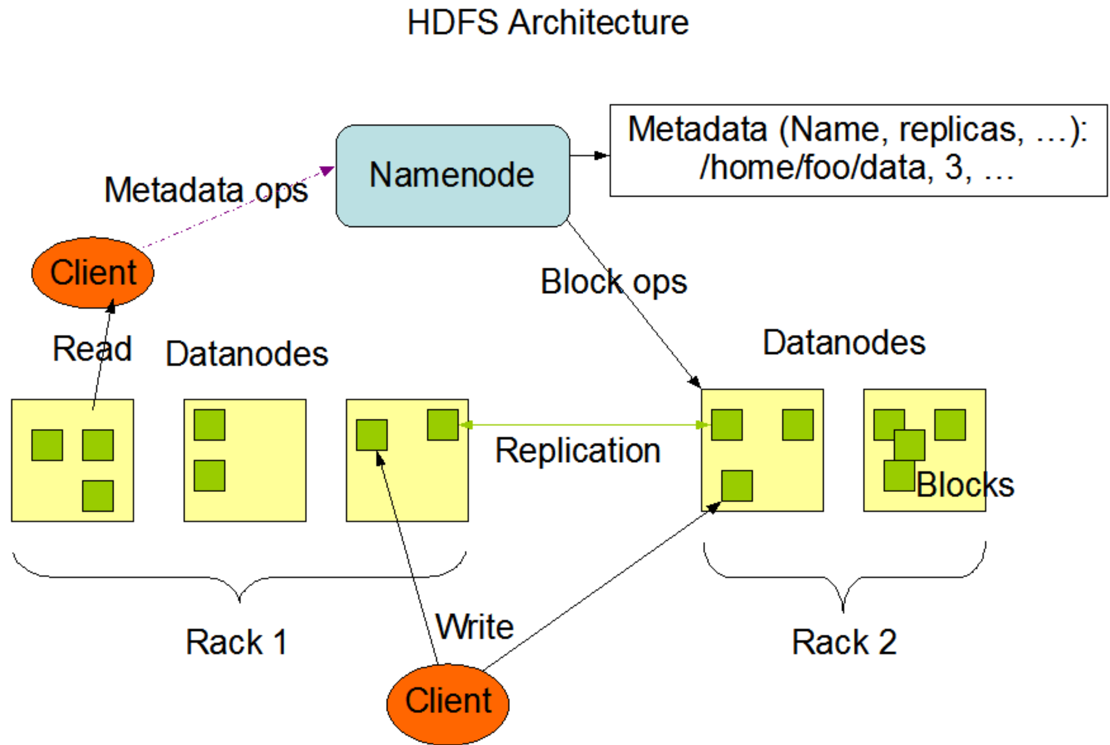

Figure 2.2: Architecture of HDFS

2.1.3 Introduction to MapReduce

Although MapReduce has been a common concept in program languages for decades, MapReduce programming for large-scale parallel data processing was just recently proposed by Dean et al. in Google [5]. First, MapReduce is a programming model designed for processing large volumes of data in parallel. MapReduce programs are written in a particular style influenced by functional programming constructs, specifically for processing lists of data. Second, MapReduce is more than a programming model - it also includes the system support for processing the MapReduce jobs in parallel in a large scale cluster. Apache Hadoop is a popular open source implementation of the MapReduce framework.

It is best to understand how MapReduce programming works with an example - the WordCount program. The following code snippet shows how this MapReduce program works. WordCount counts the frequency of each word in a large document collection. Its Map program partitions the input lines into words and emits tuples $\langle w, 1 \rangle$ for aggregation,

where 'w' represents a word and '1' means the occurrence of the word. In the Reduce program, the tuples with the same word are grouped together and their occurrences are summed up to get the final result.

Algorithm 1 The WordCount MapReduce program

```
1: map(file)
2: for each line in the file do
3:     for each word w in the line do
4:         Emit(⟨w, 1⟩)
5:     end for
6: end for

1: reduce(w, v)
2: w: word, v: list of counts.
3: d ← 0;
4: for each v_i in v do
5:     d ← d + v_i;
6: end for
7: Emit(⟨w, d⟩);
```

2.2 Amazon Web Service

Amazon Web Services (AWS)[4] is built by Amazon company in early 2006, which collect remote computing services together to compose the cloud computing clusters. AWS is a flexible cloud infrastructure that can accommodate user requests on various sizes of clusters, consisting of tens to thousands of virtual machines. Developing applications with AWS has several benefits: cost-effective, dependable, flexible and comprehensive. AWS provides several services including the well-known the Amazon Elastic Compute Cloud (AWS EC2), a typical infrastructure-as-a-service, and Amazon Simple Storage Service (AWS S3) for scalable cloud storage.

AWS EC2 is based on the virtualization technique []. It allow users to launch virtual machine instances with a variety of operating systems, load them with the custom application environment, manage the network access permissions, and scale up/down the cluster.

AWS S3 is a cloud storage system. AWS S3 provides a simple web service interface (in both SOAP and REST) that can be used to store and retrieve data objects. A data object

[4]http://aws.amazon.com/what-is-aws/

in S3 is stored in a bucket and identified by a key. Each data object can also be accessed via a URL. .

We use Amazon EC2 and S3 for experiments on the public cloud. Specifically, we use S3 to store large experimental datasets and EC2 to conveniently setup Hadoop clusters of different sizes. Once a Hadoop cluster is setup in EC2, the selected datasets in S3 will be loaded to the HDFS for experiments.

Cost Model of MapReduce

As we have discussed, running Hadoop/MapReduce programs on public clouds has to estimate the performance and the scale of resources needed to achieve the performance. We believe a complete understanding of the cost model of MapReduce is critical to address these problems. In this chapter, we analyze the components in the whole MapReduce execution process and derive a cost model in terms of the input data, the application-specific complexity, and the available system resources. This cost model is the core component for solving the resource prediction and optimization problems presented in Section 3.3.

3.1 Factors Affecting the Performance of MapReduce

The performance of a MapReduce program can be affected by many factors. The first set of factors is determined by the system configuration, including hardware, network topology, operating system, and the hadoop system. Jiang et al [10] and Wang et al [30, 29] have investigated how these system-level factors can affect the performance of a MapReduce program.

Once the system configuration is fixed, the most important factor is the amount of system resources available for a MapReduce job. This is translated to the number of compute nodes allocated to the hadoop cluster if the one-job-per-cluster strategy is used for the public cloud. Our study will focus on the number of compute nodes that can finish the jobs with minimized the cloud cost, assuming the optimal system-level configuration has been

used for each node and the whole Hadoop system.

3.2 Analysis of the MapReduce Process

We use a mixed "black-box" and "white-box" method to analyze the MapReduce process. Specifically, the whole MapReduce process is decomposed into a number of sequential and parallel processing components. We clearly identify the relationship between these sequential and parallel components. On the other hand, some components such as the Reduce function will be treated as black-boxes, needing user's input to determine their complexity.

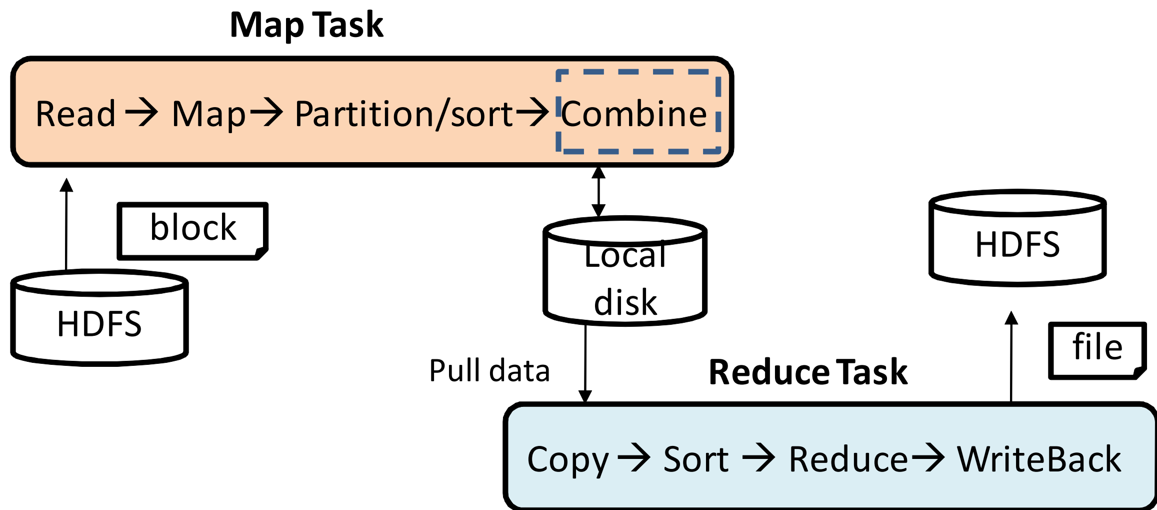

Figure 3.1: Components in Map and Reduce tasks and the sequence of execution.

The MapReduce processing is a mix of sequential and parallel processing. The Map phase is executed before the Reduce phase[1], as Figure 3.1 shows. However, in each phase many Map or Reduce processes are executed in parallel. To clearly describe the MapReduce execution, we would like to distinguish the concepts of *Map/Reduce slot* and *Map/Reduce process*. Each Map (or Reduce) process is executed in a Map (or Reduce) slot.

[1]The Copy operation in the Reduce phase overlaps the Map phase - when a Map's result is ready, Copy may start immediately.

A slot is a unit of computing resources allocated for the corresponding process. According to the system capacity, a computing node can only accommodate a fixed number of slots so that the processes can be run in the slots in parallel without serious competition. In Hadoop, the Tasktracker running in each slave node has to set the number of Map slots and the number of Reduce slots. A common setting for a multi-core computer is to have two Map or Reduce slots per core. Let's assume there are m Map slots and r Reduce slots in total over all slave nodes.

We define a Map/Reduce process as a Map/Reduce task running on a specific slot. By default, in Hadoop each Map process handles one chunk of data (e.g., 64MB). Therefore, if there are M chunks of data, M Map processes in total will be scheduled, which are assigned to the m slots. In the ideal case, m Map processes occupy the m slots and run in parallel - we call it one round of Map processes. If $M > m$, which is normal for large datasets, $\lceil M/m \rceil$ Map rounds are needed. Different from the total number of Map processes, the number of Reduce processes, say R, can be set by the user and determined by the application requirement. Similarly, if $R > r$, more than one round of Reduce processes are scheduled. In practice, to avoid the cost of scheduling multiple rounds of Reduce processes, the number of Reduce processes is often set to the same as or less than the number of Reduce slots in the cluster[2].

Figure 3.2 illustrates the scheduling of Map and Reduce processes to the Map and Reduce slots in the ideal situation. In practice, in one round Map processes may not finish exactly at the same time - some may finish earlier or later than others due to the system configuration, the disk I/O, the network traffic, and the data distribution. But we can use the total number of rounds to roughly estimate the total time spent in the Map phase. We will consider the variance in cost modeling. Intuitively, the more available slots, the faster the whole MapReduce job can be finished. However, in the pay-as-you-go setting, there

[2]In general, the number of Reduce processes, R, is not larger than the number of Map output keys, because one Reduce process handles one or more output keys. In many applications, the number of Map output keys is so large that R is often set to the number of all available Reduce slots to optimize the performance [32].

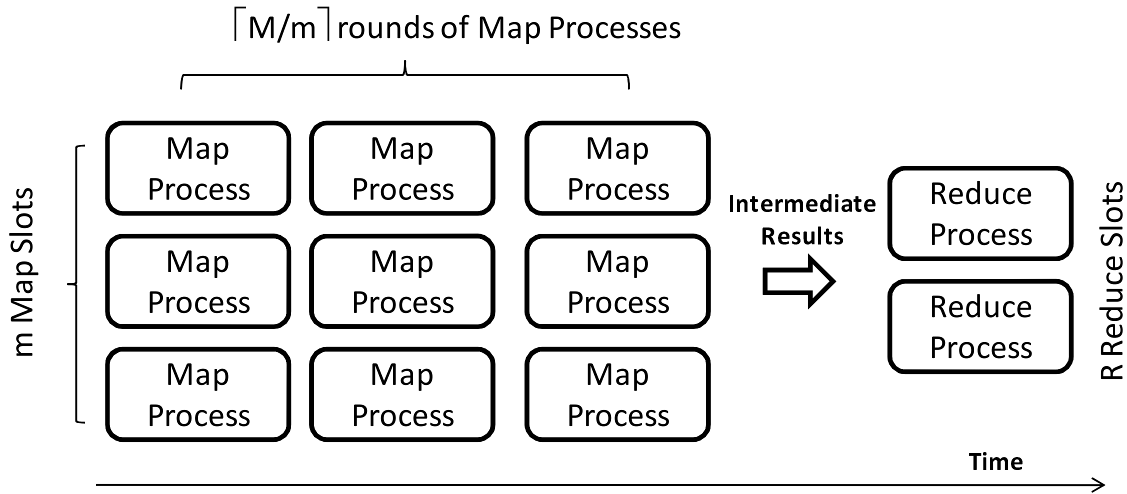

Figure 3.2: Illustration of parallel and sequential execution in the ideal situation.

is a tradeoff between the amount of the resources and the amount of time to finish the MapReduce job.

In addition to the cost of Map and Reduce processes, the system has some additional cost managing and scheduling the M Map processes and the R Reduce processes. Based on this understanding, we analyze the cost of each Map process and Reduce process, respectively, and then derive the overall cost model.

3.2.1 Map Process

A Map process can be divided into a number of sequential components, including Read, Map, Sort/Partition, and optionally Combine, as Figure 3.1 shows. We understand this process in term of a data flow - data sequentially flow through each component and the cost of each component depends on the amount of input data.

The first component is reading a block of data from the disk, which can be either local or remote data block. Let's assume the average cost is a function of the size of data block b: $i(b)$. The second component is the user defined Map function, the complexity of which is determined by the input data size b, denoted as $f(b)$. The Map function may output data in size of $o_m(b)$ that is often a linear function to the input size b. The output will be a list

of $\langle key, value\rangle$ pairs. The result will be sorted by the key and partitioned into R shares for the R Reduce processes. We denote the cost of partitioning and sorting with $s(o_m(b))$. Since the partitioning process uses a hash function to map the keys, the cost $s(o_m(b))$ is independent of R. Let's skip the Combiner component temporarily and we will discuss the situation having the Combiner component later.

In summary, the overall cost of a Map process is the sum of the costs (without the Combiner component):

$$\Phi_m = i(b) + f(b) + s(o_m(b)). \tag{3.1}$$

This cost is only related to the size of the data block b and the complexity of the Map function. It is independent of the parameters M,R and r.

3.2.2 Reduce Process

The Reduce process has the components: Copy, MergeSort, Reduce and WriteResult. These components are also sequentially executed in the Reduce process.

Assume the k keys of the Map result are equally distributed to the R Reduce processes[3]. In the Copy component, each Reduce process pulls its shares, i.e., k/R keys and the corresponding records, from the M Map processes' outputs. Thus, the total amount of data in each Reduce will be

$$b_R = M \cdot o_m(b) \cdot k/R. \tag{3.2}$$

The Copy cost is linear to b_R, denoted as $c(b_R)$. A MergeSort follows to merge the M shares from the Map results while keeping the records sorted, which has the complexity $O(b_R \log b_R)$, denoted as $ms(b_R)$.

The Reduce function will process the data with some complexity $g(b_R)$ that depends on the real application. Assume the output data of the Reduce function has an amount

[3]For this reason, the user normally selects R to satisfy $k \geq R$. If $R > k$, only k Reduces are actually used.

$o_r(b_R)$, which is often less than b_R. Finally, the result is duplicated and written back to multiple nodes, with the complexity linear to $o_r(b_R)$, denoted as $wr(o_r(b_R))$.

In summary, the cost of the Reduce process is the sum of the component costs,

$$\Phi_r = c(b_R) + ms(b_R) + g(b_R) + wr(o_r(b_R)). \tag{3.3}$$

3.2.3 Putting All Together

According to the parallel execution model we described in Figure 3.2, the overall time complexity T depends on the number of Map rounds and Reduce rounds. By including the cost of managing and scheduling the Map and Reduce processes $\Theta(M, R)$, which is assumed to be linear to M and R, we represent the overall cost as

$$T = \lceil \frac{M}{m} \rceil \Phi_m + \lceil \frac{R}{r} \rceil \Phi_r + \Theta(M, R). \tag{3.4}$$

We are more interested in the relationship among the total time T, the input data size $M \times b$, the user defined number of Reduce processes R, and the number of Map and Reduce slots, m and r. If we use a fixed block size b in the analysis, the cost of each Map process, Φ_m, is fixed. The cost of each Reduce process, Φ_r, is subject to the factor M and R. Since the user setting R is often the same as or less than the number of Reduce slots, r, we let $\lceil R/r \rceil = 1$. To make it more convenient to manipulate the equation, we also remove $\lceil\rceil$ from $\lceil M/m \rceil$ by assuming $M \geq m$ and M/m is an integer. After plugging in the equations 3.2 and 3.3 and keeping only the variables M, R, and m in the cost model, we get the detailed model

$$\begin{aligned} &T_1(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{M}{R} + \beta_3 \frac{M}{R} \log(\frac{M}{R}) \\ &+ g(\frac{M}{R}) + \beta_4 M + \beta_5 R + \epsilon, \end{aligned} \tag{3.5}$$

where β_i are the parameters describing the constant factors. $T_1(M, m, R)$ is not linear to its variables, but it is linear to the transformed components: M/m, M/R, $\frac{M}{R}\log(\frac{M}{R})$, $g(M/R)$, M, and R. The parameter β_i defines the contribution of each components in the model. Concretely, β_1 represents the fixed Map cost Φ_m; β_2 represents the parameter associated with the cost of Copy and Write Back in the Reduce phase; β_3 represents the parameter associated with the MergeSort component in the Reduce phase; β_4 and β_5 represent the parameters for the cost associated with the management cost $\Theta()$, i.e., we assume the cost is linearly associated with the number of Map and Reduce slots: $\Theta(M, R) = \beta_4 M + \beta_5 R$; β_0 represents some constant, and ϵ represents the noise component that covers the unknown or unmodeled factors in the system. We leave the discussion on the item $g(M/R)$ later.

The simplicity of the linear model has several advantages. If this model is valid, it will allow me to robustly estimate the time complexity of larger data (i.e., larger M) and more resources (larger m and R) based on the model parameters estimated with the small settings of M, m, and R. It can also reduce the complexity of solving the related optimization problems.

With Combiner. In the Map process, the Combiner function is used to aggregate the results by the key. If there are k keys in the Map output, the Combiner function reduces the Map result to k records. The cost of Combiner is only subject to the output of the Map function. Thus, it can be incorporated into the parameter β_1. However, the Combiner function reduces the output data of the Map process and thus affects the cost of the Reduce phase. With the Combiner, the amount of data that a Reduce process needs to pull from the Map is changed to

$$b_R = Mk/R. \tag{3.6}$$

Since the important factors M and R are still there, the cost model (Equation 3.5) applies without any change.

Function g(). The complexity of Reduce function has to be estimated with the given application. There are some special cases that the $g()$ item can be removed from Equation

3.5. If $g()$ is linear to the size of the input data, then its contribution can be merged to the factor β_2, because $g(M/R) \sim M/R$. Similarly, if its complexity is $O(\frac{M}{R}\log(\frac{M}{R}))$, its contribution can be merged to β_3. In these two special cases, the cost model is simplified to

$$
\begin{aligned}
&T_2(M, m, R) = \\
&\beta_0 + \beta_1\frac{M}{m} + \beta_2\frac{M}{R} + \beta_3\frac{M}{R}\log(\frac{M}{R}) \\
&+\beta_4 M + \beta_5 R + \epsilon, \qquad (3.7)
\end{aligned}
$$

In practice, many applications can be covered by the special cases.

Observations. Let's look closer to the parameters of the simplified model T_2. First, let's fix M and R. We have $T_2 \sim 1/m$. This relationship indicates that when m is already large, the increase of m will not bring significant performance gain. In particular, if M is smaller than m, increasing m will not gain, at all. Second, let's fix M and m. Then, the function of R is more complicated, involving R, $1/R$, and $(\log R)/R$. We will have to depend on experiments to explore the function of R. Finally, if we fix m and R and increase the data size M, the complexity might be dominated by the item $\frac{M}{R}\log(\frac{M}{R})$. A Combiner function can significantly reduce the weight of this item.

3.3 Optimization of Resource Provisioning

With the cost model we are now ready to find the optimal settings for different decision problems. We try to find the best resource allocation for three typical situations: (1) with certain limited amount of financial budget; (2) with certain time constraint; (3) and without any constraint. We formulate these problems as optimization problems based on the cost model.

In all the scenarios we consider, we assume the model parameters are determined with

sample runs in small scale settings. We also assume $g()$ function is one of the two simple cases. Therefore, the simplified model T_2 is applied. Since the input data is fixed, M is constant. For simplicity, we also consider all general MapReduce system configurations [1, 9] are fixed for both small and large scale settings. With this setup, the time cost function becomes

$$T_3(m, R) = \alpha_0 + \frac{\alpha_1}{m} + \frac{\alpha_2}{R} + \frac{\alpha_3 \log R}{R} + \alpha_4 R \tag{3.8}$$

where

$$\begin{aligned}
\alpha_0 &= \beta_0 + \beta_4 M, \\
\alpha_1 &= \beta_1 M, \\
\alpha_2 &= \beta_2 M + \beta_4 M \log M, \\
\alpha_3 &= -\beta_3 M, \\
\alpha_4 &= \beta_5.
\end{aligned}$$

In the virtual machine (VM) based cloud infrastructure (e.g., Amazon EC2), the cost of cloud resources is calculated based on the number of VM instances used in time units (typically in hours). According to the capacity of a virtual machine (CPU cores, memory, disk and network bandwidth), a virtual node can only have a fixed number of Map/Reduce slots. Let's denote the number of slots per node as γ. Thus, the total number of slots $m + r$ required by a on-demand Hadoop cluster can be roughly transformed to the number of VMs, v, as

$$v = (m + r)/\gamma. \tag{3.9}$$

If the price of renting one VM instance for an hour is u, the total financial cost is determined by the result $uvT_3(m, R)$. Since we usually set R to r, it follows that the total financial cost

for renting the Hadoop cluster is

$$uvT_3(m, R) = u(m + R)T_3(m, R)/\gamma. \tag{3.10}$$

Therefore, given a financial budget ϕ, the problem of finding the best resource allocation to minimize the job time can be formulated as

$$\begin{aligned} & minimize\ T_3(m, R) \\ & subject\ to\ u(m + R)T_3(m, R)/\gamma \le \phi, \\ & \quad m > 0,\ and\ R > 0. \end{aligned} \tag{3.11}$$

If the constraint is about the time deadline τ for finishing the job, the problem of minimizing the financial cost can be formulated as

$$\begin{aligned} & minimize\ u(m + R)T_3(m, R)/\gamma \\ & subject\ to\ T_3(m, R) \le \tau, m > 0,\ and\ R > 0. \end{aligned} \tag{3.12}$$

The above optimization problem can also be slightly changed to describe the problem that the user simply wants to find the most economical solution for the job without time deadline, i.e., the constraint $T_3(m, R) \le \tau$ is removed.

Note that the T_3 model parameters might be specific for a particular type of VM instance that also determines the parameters u and γ. Therefore, by testing different types of VM instance and applying this optimization repeatedly on each instance type, we can also find which instance type is the best.

With the concrete setting of the T_3 model parameters (i.e., α_i be positive or negative), these optimization problems can be convex or non-convex [2]. However, they are in the category of well-studied optimization problems - there are plenty of papers and books dis-

cussing how to solve these optimization problems. Therefore, we will skip the details of solving these problems.

Experiments

In this section, we design a set of experiments on both inhouse cluster and on-demand AWS clusters. A number of testing MapReduce programs are developed and tested on simulated datasets to generate sample data for modeling the cost function. The modeling results are analyzed and tested on different input data sizes and cluster sizes to observe the prediction accuracy of these models.

In the following, we first present the setup of the experiments, including the experimental environment and the methods to generate simulated datasets. Then, we describe four tested MapReduce programs: WordCount, TeraSort, PageRank and Join. Each tested program will be executed in the cluster (both Inhouse and AWS) for a number of runs with different amounts of input data and different numbers of Map/Reduce slots. At last, the collected data is used for regression analysis and model prediction for different situations.

4.1 Experimental Setup

4.1.1 Hardware and Hadoop Configuration for Inhouse Cluster

The experiments are conducted in our inhouse 16-node Hadoop cluster. Each node has two quad-core 2.3Mhz AMD Opteron 2376, 16GB memory, and two 500GB hard drives, connected with a gigabit switch. The version 0.21.0 of Hadoop is installed in the cluster. One node serves as the master node and the other as the slave nodes. The single master node

runs the *JobTracker* and the *NameNode*, while each slave node runs both the *TaskTracker* and the *DataNode*. Each slave node is configured with eight Map slots and six Reduce slots (about two concurrent processes per core). Each Map/Reduce process uses 400MB memory. The data block size is set to 64 MB. We use the Hadoop fair scheduler to control the total number of Map/Reduce slots available for different testing jobs.

4.1.2 AWS cluster configurations

We use small EC2 instances to setup AWS clusters. Each small instance has one EC2 Compute Unit (one 1.2Ghz core), 1.7 GB memory, and 160 GB local storage. The version and configuration of Hadoop running on the EC2 cluster is the same as that in the local experiments. We use one instance to host one Map or Reduce slot. Therefore, the size of the cluster is determined by the total number of Map and Reduce slots.

4.1.3 Datasets

we use a number of generators to generate testing datasets for the benchmark programs. (1) We revised the RandomWriter tool in the Hadoop package to use a Gaussian random number generator to generate random float numbers. This data is used for the Sort program. (2) We also revised the RandomTextWriter tool to generate text data based on a list of 1000 words randomly sampled from the system dictionary /usr/share/dict/words. This dataset is used for both the WordCount program and the TableJoin program. (3) The third dataset is a random graph dataset. Each line of the data starts with a node ID, its initial PageRank, and followed with a list of node IDs representing the nodes outlinks. Both the node ID and the outlinks are randomly generated integers. Each type of data consists of 150 1GB files. For a specific testing task, we will randomly choose a number of the 1GB files to simulate different sizes of input data.

4.1.4 MapReduce Programs for Testing

We describe four MapReduce programs used in testing and give the complexity of each one's Reduce function, i.e., the $g()$ function. If $g()$ is linear to the input data, the simplified cost model Eq. 3.7 is used.

WordCount is a sample MapReduce program in the Hadoop package. The Map function splits the input text into words and the result is locally aggregated by word with a Combiner; the Reduce function sums up the local aggregation results $\langle word, count\rangle$ by words and output the final word counts. Since the number of words is limited, the amount of output data to the Reduce stage and the cost of Reduce stage are small, compared to the data and the processing cost for the Map stage. The complexity of the Reduce function, $g()$, is linear to Reduce's input data.

Sort is also a sample MapReduce program in the Hadoop package. It depends on a custom partitioner that uses a sorted list of $N-1$ sampled keys that define the key range for each Reduce. As a result, all keys such that sample$[i-1] <=$ key $<$ sample$[i]$ are sent to Reduce i. Then, the inherent MergeSort in the Shuffle stage sorts the input data to the Reduce. This guarantees that the output of Reduce i are all less than the output of Reduce i+1. Both the Map function and the Reduce function do nothing but simply pass the input to the output. Therefore, the function $g()$ is also linear to the size of the input of Reduce.

PageRank is a MapReduce implementation of the well known Google's PageRank algorithm [3]. PageRank is an iterative algorithm applied on a graph dataset. Assume each node p_i in the graph has a PageRank $PR(p_i)$. $M(p_i)$ represents the set of neighboring nodes of p_i that have an outlink pointing to p_i. $L(p_j)$ is the total number of outlinks the node p_j has. d is the damping factor and N is the total number of nodes. The following equation calculates the PageRank for each node p_i.

$$PR(p_i) = (1-d)/N + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{4.1}$$

PageRank values are updated in multiple rounds until they converge. In one round of PageRank MapReduce program, all nodes' PageRank values are updated in parallel based on the above equation. Concretely, the Map function distributes a share of each node's PageRank, i.e., $PR(p_j)/L(p_j)$, to all its outlink neighbors. The Reduce function collects the shares from its neighbors and applies the equation to update the PageRank. The complexity function $g()$ is also linear to the size of the input of Reduce.

Join is a MapReduce program that joins a large file and a small file based on a designated key, which mimics the Join operation in relational database. The large files are the text files randomly generated with RandomTextWriter. The small file consists of 50 randomly generated lines using the same method for generating the large text dataset. The first word of each line in both types of file serves as the join key. The Map function emits the lines of the input large and small files. Each line of the small file is labeled so that they can be distinguished from the Map output. In the Reduce, the lines are checked. If the lines from both files are found, a cartesian product is applied between the two sets of lines to generate the output. Depending on the key distribution, the size of output data may vary. In the Reduce function, assume there is a λ lines are from the large file and μ lines from the small file. The result of cartesian product is $\lambda\mu$ lines. Since $\mu \leq 50$ very small, the complexity function $g()$ is approximately linear to the input $\lambda + \mu$ lines.

4.1.5 Experiments Strategy

Inhouse Experiments For each MapReduce program, the following strategy is used. First, we generate random configurations of different input size and Map/Reduce slots. Then, we test the MapReduce program with these configurations to find the execution times. Each configuration x_i, together with the execution time y_i makes a training example (x_i, y_i) for learning the parameters in the cost function for the specific MapReduce program. A number of analysis methods are used to study the quality of cost models, including regression analysis and leave-one-out prediction analysis. We also use small scale settings (small data

and small number of slots) to train models, which are used to predict the large scale cases.

Algorithm 2 The Workflow of Experiments Strategy

1: **Before Experiment**
2: Set the number of rounds of experiments
3: Set the range of number of map slots
4: Set the range of number of reduce slots
5: Set the range of amount of input data
6: **In Experiment**
7: **for** each round in the rounds of experiments **do**
8:     randomly choose the size of input data and the number of map slots and reduce slots
9:     Run(Hadoop MapReduce with specified configuration)
10: **end for**

AWS experiments First, we upload all the simulated experimental data to AWS S3 storage. Then, for each round of experiments, we use the ec2-tools command to launch the corresponding clusters with the specific number of Map/Reduce slots. The selected datasets are transferred from S3 to the EC2 Hadoop cluster. Finally, we issue the commands to run MapReduce programs in the cluster.

4.2 Result Analysis

4.2.1 Regression Model Analysis

setup: We run a set of experiments to estimate the model parameters β_i for the four MapReduce programs. We randomly select the values for the three parameters M, m, and R. The number of data chunks M is calculated by the number of selected 1GB files (one file has 1024/64 = 16 blocks). The number of Map slots m is controlled by setting the maximum number of Map slots in the *fair scheduler*. R is randomly set to a number smaller than the total number of Reduce Slots in the system.

For each tested program, we generate 25 to 60 random settings of $< M, m, R >$. M is randomly selected from the integers $[1 \ldots 150] \times 16$, i.e., the number of 1GB files $\times$ 16 blocks/file. R is randomly selected from the integers $[1 \ldots 50]$. Since changing m will need to update the scheduler setting, we limit the choices of m to 30,60,90,and 120 - for each m. For each setting, we record the time (seconds) used to finish the program. We only use

	WC		Sort		PR		Join	
	Local	AWS	Local	AWS	Local	AWS	Local	AWS
β_0	35.62	13.10	0	0	0	0	0	0
β_1	30.00	31.52	0	0	46.33	46.79	2.04	3.71
β_2	0.02	0	0	0	0	0	0	0
β_3	0	0	3.15	3.01	9.66	0	3.27	3.18
β_4	0.01	0	0.58	0.66	2.44	2.66	0.42	0.38
β_5	0	0	0	0	5.44	5.42	0.01	0.21
R^2	**0.9967**	**0.9948**	**0.9644**	**0.9541**	**0.9030**	**0.9227**	**0.9859**	**0.9815**

Table 4.1: Result of regression analysis. R^2 values are all higher than 0.90, indicating good fit of the proposed model.

m = 30,60 and 90 for AWS because of the limit of the maximum number of nodes we can request. .

Regression Analysis. With $x_1 = M/m$, $x_2 = M/R$, $x_3 = \frac{M}{R}\log(\frac{M}{R})$, $x_4 = M$, and $x_5 = R$, we can conduct a linear regression on the transformed cost model

$$T(x_1, x_2, x_3, x_4, x_5) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5. \tag{4.2}$$

We use the matlab regression method, lsqnonneg, in modeling, which considers the constraint that β parameters are non-negative. Table 4.1 shows the result of regression analysis for both Inhouse and AWS experiments. R^2 is a measure for evaluating the goodness of fit in regression modeling. $R^2 = 1$ means a perfect fit, while $R^2 > 90\%$ indicates a very good fit.

Figure 4.9,4.10,4.11, 4.12,4.13,4.14,4.15 and 4.16 show the goodness of fit in a more intuitive way. To make the presentation clearer, we sort the experimental results by the time cost in an ascending order. The solid lines represent the real times observed in the experiment and the '+' marks represent the predicted times using the fitted model. The closer the two, the better quality the model has. All of the eight figures show excellent fit. Each pair of figures shows the comparison between the inhouse experiments and AWS experiments.

Understanding the Regression Models. From previous analysis, we know that the β parameters, together with the corresponding variables $x_1 = M/m$, $x_2 = M/R$, $x_3 = \frac{M}{R}\log(\frac{M}{R})$, $x_4 = M$, and $x_5 = R$, represent the costs of the sequential components in the MapReduce program. For the **WordCount** program, most cost is at the map phrase, in which the words in documents are parsed and segmented. The distribution of β supports this observation, where in which β_0 and β_1 dominate. The **Sort** program utilizes the sorting phase in the MapReduce framework to sort data, which is between the map and the reduce phrase (Section 3.2). Because x_3 represents the sorting phrase, the corresponding β_3 becomes the dominant parameter. The same analysis can be applied to the **PageRank** and **TableJoin**. Therefore, the experimental results confirm our cost models.

Cross Validation. We also perform the leave-one-out cross validation for both types of cluster to study the prediction accuracy of the model. The leave-one-out cross validation runs in n rounds if there are n training examples, i.e., the tuples of (M, m, R, T). In each round, it uses one of the n examples as the testing example and the other $n-1$ examples for training the model. The accuracy is defined as the average relative errors (ARE) over the n rounds of testing. Let C_i be the real cost and $\hat{C}_i$ be the estimated cost by the trained model in the round i. We calculate ARE with the following equation.

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|C_i - \hat{C}_i|}{C_i} \tag{4.3}$$

Table 4.2 shows the relative error rates in leave-one-out cross validation for both cases. For comparison, we also list the result of testing on training data. The result confirms these models are robust and perform well.

Trend Analysis In reality, we want to use the training data of the small cases to generate the model and predict the performances of the large cases. Small cases here mean both the size of data and the size of cluster are small. In this analysis, we simulate this scenario: we partition the results into two parts, smaller cases as the training samples and

	WC		Sort		PR		Join	
	Local	AWS	Local	AWS	Local	AWS	Local	AWS
Test-on-training	9.42%	16.02%	17.02%	16.32%	11.19%	9.46%	14.55%	14.44%
Leave-one-out	10.02%	17.26%	17.47%	17.08%	14.98%	12.10%	15.49%	16.42%

Table 4.2: Average relative error rates of the leave-one-out cross validation and of the testing result on training data for the four programs.

larger cases as the testing samples. Then we vary the size of training samples for several rounds, and find the average error rate of the corresponding testing data for each round.

Figure 4.1,4.2,4.3, 4.4,4.5,4.6,4.7 and 4.8 show the result: the x axis is the size of training sample and the solid line is the average error rates of testing samples, which decrease when the size of training sample increase at first, and then vary very small when the size of training sample is big enough. It supports our assumption: we can use the small cases to train a model to accurately predict the performances of large cases.

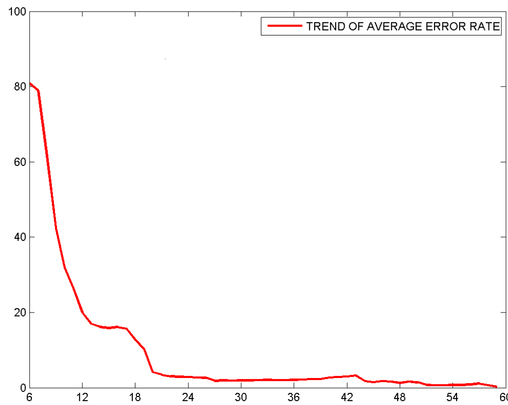

Figure 4.1: Trend for WordCount on In-house Cluster.

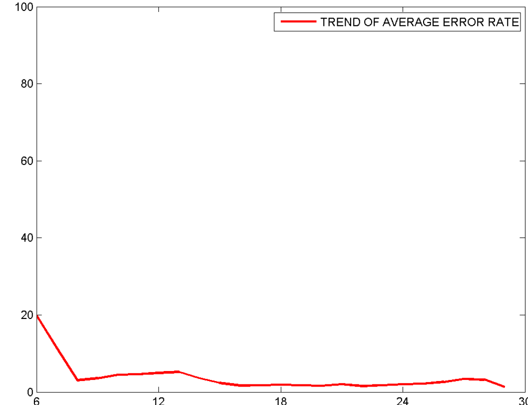

Figure 4.2: Trend for WordCount on AWS Clusters.

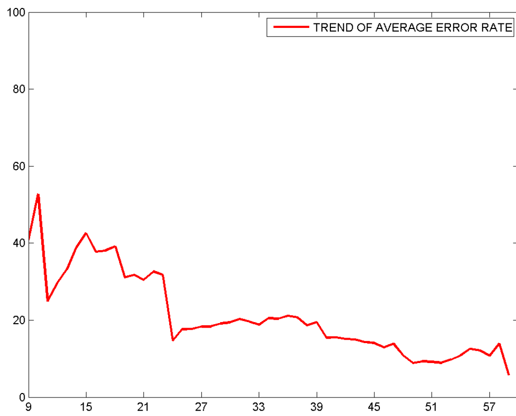

Figure 4.3: Trend for TeraSort on Inhouse Cluster.

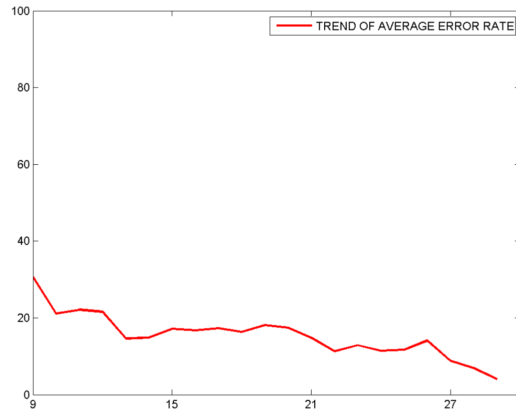

Figure 4.4: Trend for TeraSort on AWS Cluster.

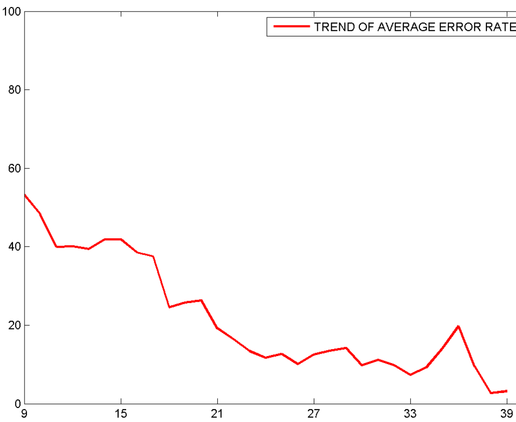

Figure 4.5: Trend for Join program on Inhouse Cluster.

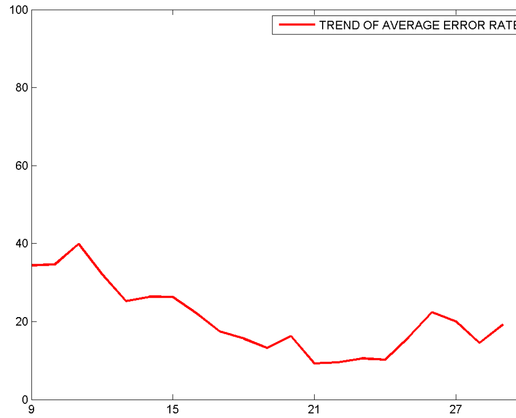

Figure 4.6: Trend for Join program on AWS Cluster.

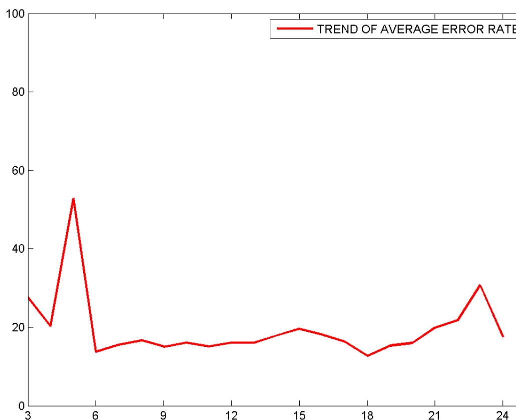

Figure 4.7: Trend for PageRank on Inhouse Cluster.

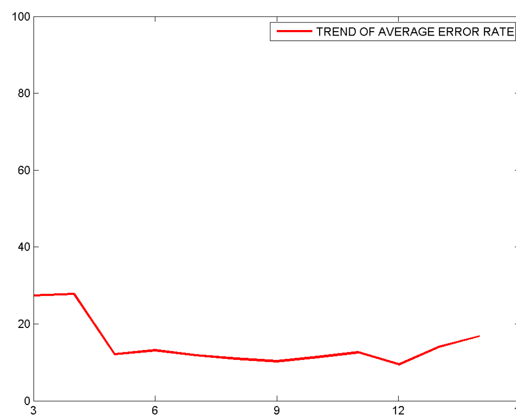

Figure 4.8: Trend for PageRank on AWS Cluster.

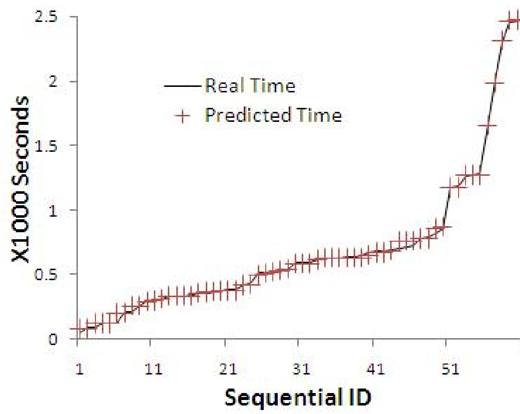

Figure 4.9: Fitting the model for WordCount on Inhouse Cluster(60 rounds).

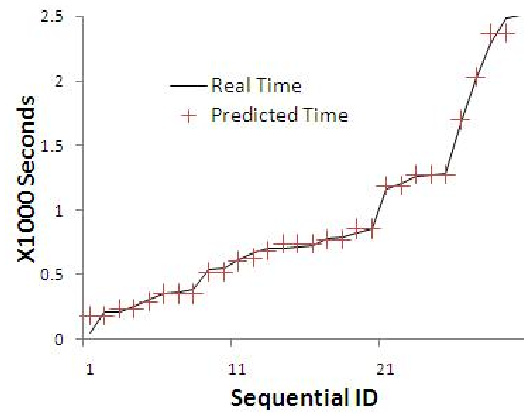

Figure 4.10: Fitting the model for WordCount on AWS Clusters (30 rounds).

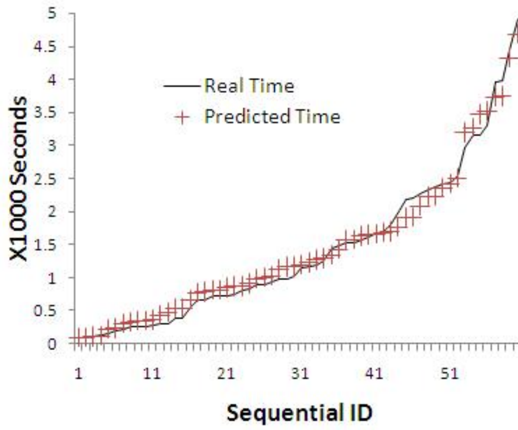

Figure 4.11: Fitting the model for TeraSort on Inhouse Cluster(60 rounds).

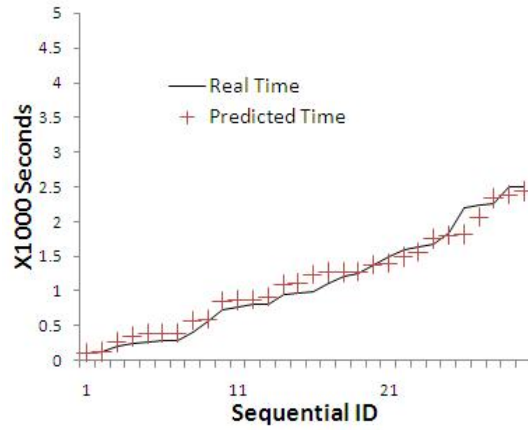

Figure 4.12: Fitting the model for TeraSort on AWS Cluster(30 rounds).

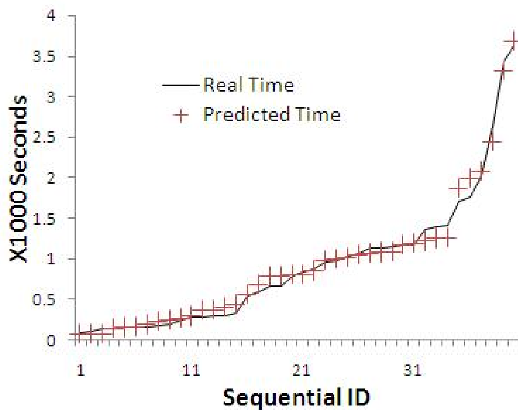

Figure 4.13: Fitting the model for Join program on Inhouse Cluster(40 rounds).

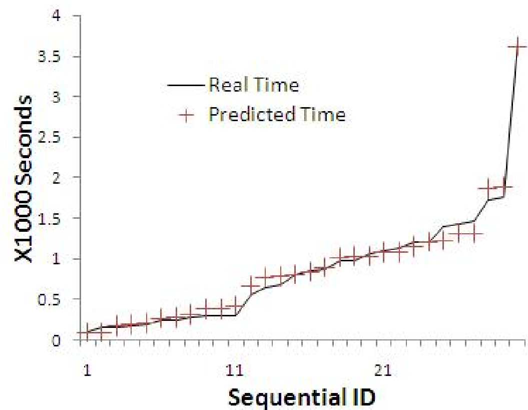

Figure 4.14: Fitting the model for Join program on AWS Cluster(30 rounds).

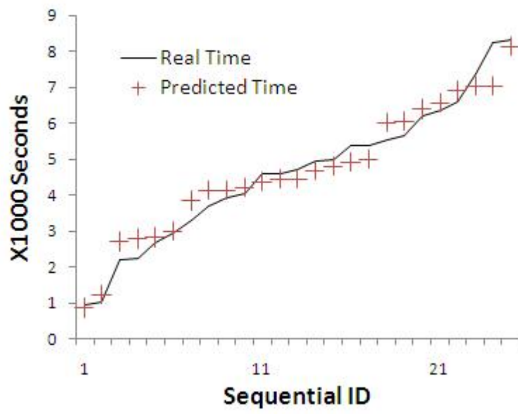

Figure 4.15: Fitting the model for PageRank on Inhouse Cluster(25 rounds).

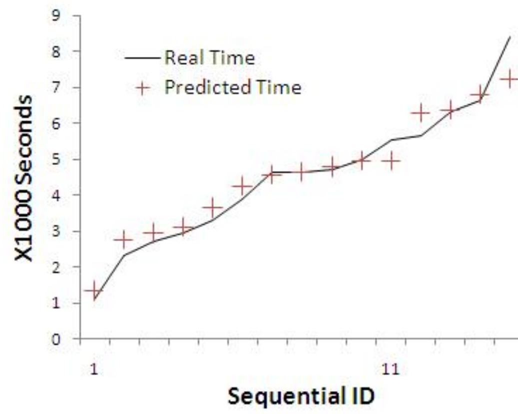

Figure 4.16: Fitting the model for PageRank on AWS Cluster(15 rounds).

Conclusion and Future Work

5.1 Conclusion

Running MapReduce programs in the public cloud raises the important problem: how to optimize resource provisioning to minimize the financial cost for a specific job? In this thesis, we study the components in MapReduce processing and build a cost function that explicitly models the relationship among the amount of data, the available system resources (Map and Reduce slots), and the complexity of the Reduce function for the target MapReduce program. The model parameters can be learned from test runs with small scale settings on the target program. Based on this cost model, we can solve a number of decision problems, such as the optimal amount of resources that can minimize the financial cost with the constraints of financial budget or time deadline. We have also conducted a set of experiments on both a local hadoop cluster and Amazon EC2 on-demand clusters to validate the model. The result shows that this cost model fits well on four tested MapReduce programs.

5.2 Future Work

The results have shown the proposed cost model has high fidelity to fit the real MapReduce programs. However, there is still a significant amount of prediction error. We want to investigate where the error comes from and how to further improve the model to reduce the

error. Another task is to apply the learned cost model to the optimization problems to study the properties of the optimization results. This work also establishes a general framework for analyzing parallel data intensive computing in the cloud, which we hope to extend to other computing models such as Dryad [8].

Bibliography

[1] S. Babu, "Towards automatic optimization of mapreduce programs," in *Proceedings of the 1st ACM symposium on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 137–142.

[2] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *International Conference on World Wide Web*, 1998.

[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *International Conference on World Wide Web*. New York, NY, USA: ACM, 2007, pp. 271–280.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[6] A. Goyeneche, G. Terstyanszky, T. Delaitre, and S. Winter, "Improving grid computing performance prediction using weighted templates," in *Conf. Proc. of the UK e-Science*, 2007.

[7] F. Guim, A. Goyeneche, J. Corbalan, J. Labarta, and G. Terstyansky, “Grid computing performance prediction based in historical information 1,” 2004.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *EuroSys ’07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

[9] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, “The performance of mapreduce: An in-depth study,” in *Proceedings of Very Large Databases Conference (VLDB)*, 2010.

[10] ——, “The performance of mapreduce: An in-depth study,” in *The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.*, 2010.

[11] T. Joachims, L. Granka, B. Pan, and G. Gay, “Accurately interpreting clickthrough data as implicit feedback,” in *Proceedings of ACM SIGIR Conference*, 2005.

[12] K. Kambatla, A. Pathak, and H. Pucha, “Towards optimizing hadoop provisioning in the cloud,” in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud09)*, 2009.

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “Pegasus: Mining peta-scale graphs,” *Knowledge and Information Systems (KAIS)*, 2010.

[14] H. Karloff, S. Suri, and S. Vassilvitskii, “A model of computation for mapreduce,” in *Symposium on Discrete Algorithms (SODA) (2010)*, 2010.

[15] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “An analysis of traces from a production mapreduce cluster,” in *IEEE/ACM International Conference on Cluster Cloud and Grid Computing*, 2010, pp. 94–103.

[16] M. Kiran, A.-H. A. Hashim, L. M. Kuan, and Y. Y. Jiun, "Execution time prediction of imperative paradigm tasks for grid scheduling optimization," *IJCSNS International Journal of Computer Science and Network Security*, vol. 9, no. 2, February 2009.

[17] J. Lin and C. Dyer, *Data-intensive text processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[18] K. Morton and A. Friesen, "Kamd a progress estimator for mapreduce pipelines," 2009, this paper is written by Kristi Morton,but I can't find whether it had been published.

[19] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of mapreduce pipelines," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2010.

[20] ——, "Estimating the progress of mapreduce pipelines," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA 2010*, 2010.

[21] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: Massively parall learning of tree ensembles with mapreduce," in *Proceedings of Very Large Databases Conference (VLDB)*, 2009.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of ACM SIGMOD Conference*, 2009.

[23] R. M. Piro, A. Guarise, G. Patania, and A. Werbrouck, "Using historical accounting information to predict the resource usage of grid jobs," *Future Gener. Comput. Syst.*, vol. 25, May 2009.

[24] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *SIGMETRICS/Performance09*, 2009.

[25] W. Smith, "Prediction services for distributed computing," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[26] W. Smith, I. Foster, and V. Taylor, "Predicting application run times using historical information," 1997.

[27] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of ACM SIGMOD Conference*. ACM, 2010, pp. 1013–1020.

[28] G. Wang, ali r. Butt, P. Pandey, and K. Gupta, "a simulation approach to evaluating design decisions in mapreduce setups," in *mascots09,the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2009.

[29] ——, "using realistic simulation for performance analysis of mapreduce setups," in *LSAP '09,ACM workshop on Large-Scale System and Application Performance (LSAP '09)*, 2009.

[30] G. Wang, A. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups," in *the IEEE/ACM Intl. Symposium on Modelling, Analysis and Simulation of Computer and Telecomm. Systems*, 2009.

[31] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstro?m, "The worst-case execution time problem overview of methods and

survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, April 2008.

[33] yahoo, "Yahoo's blog," http://developer.yahoo.com/hadoop/tutorial/module2.html.

[34] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," University of California at Berkeley, Tech. Rep. UCB/EECS-2009-55, april 2009.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *8th USENIX Symposium on Operating Systems Design and Implementation(OSDI08)*, 2008.