Cluster management system design for big data infrastructures

Gupta, Shekhar

DOI
10.4233/uuid:de1d4543-9bbe-4a2f-ac9a-f648f4066d0f

Publication date
2016

Document Version
Final published version

Citation (APA)
Gupta, S. (2016). *Cluster management system design for big data infrastructures*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:de1d4543-9bbe-4a2f-ac9a-f648f4066d0f

CLUSTER MANAGEMENT SYSTEM DESIGN FOR BIG DATA INFRASTRUCTURES

CLUSTER MANAGEMENT SYSTEM DESIGN FOR BIG DATA INFRASTRUCTURES

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
woensdag, 14 december 2016 om 12:30 uur

door

Shekhar GUPTA

informatica ingenieur
geboren te Bhopal, India.

This dissertation has been approved by the

promotor: Prof. dr. C. Witteveen and
copromotor: Dr. J. de Kleer

Composition of the doctoral committee:

Rector Magnificus,	voorzitter
Prof. dr. C. Witteveen,	Technische Universiteit Delft, promotor
Dr. J. de Kleer,	Palo Alto Research Center, copromotor

Independent members:

Prof. dr. F. Wotawa,	Graz University of Technology
Prof. dr. G. Provan,	University College Cork
Prof. dr. D. Epema,	Technische Universiteit Delft
Prof. dr. A. Nowe,	Vrije Universiteit Brussel

Other members:

Dr. C. Fritz,	Savioke

ISBN 978-94-6186-757-5

Dedicated to my parents, Gayatri and Virendra Gupta.

I love you Kriti.

PREFACE

Oh my god...I am writing Preface, which means I have finally completed the long journey of my PhD. I did my thesis in a rather unusual fashion. I was doing my research work at Palo Alto Research Center (Xerox PARC), which is thousands of miles away from TUDelft. I started my PhD right after finishing my masters at TUDelft, and this transition was not very easy in the beginning. For some reason, I always felt that I am missing out on many learning opportunities and enjoyable activities because I am not in a university. I was constantly intimidated by the smartness around me at PARC. Also, I was new in the Bay Area, and I couldn't make friends here in the beginning. As time passed, I met people and began to realize that Bay Area is flooded with innovative technical ideas and opportunities, and I am extremely lucky to be here. I also learned that the friendliness index of people at PARC is far above than their intelligence. All of this helped me move my research work in the right direction.

I am nearing the end of my journey, and there are many names that come to my mind who either helped me start or advised me throughout my quest. I cannot thank Arjan J. C. Van Gemund enough, who is responsible for every good thing that happened to me since I arrived in Delft. Arjan supervised my master's thesis, and then he referred me to Johan De Kleer for a PhD position at PARC. Also, Arjan introduced me to Cees Witteveen, who agreed to become my PhD supervisor at TUDelft. I would like to thank Johan for giving me this wonderful opportunity to work at PARC, and making sure that I get all the help I needed in terms of supervision and financial resources. I am also very thankful to Cees for his long distance supervision. Although we are in different time zones, he always found time for our Skype meetings and has been very supportive throughout this process. Many thanks to Christian Fritz at PARC for closely supervising my work and brainstorming many ideas as part of this thesis. I really appreciate his help with writing papers, and helping me learn many new things. I would also like to thank Bob Price at PARC for answering my statistics related questions, and assisting me with my papers. Thanks to Roger Hoover for helping me with the code base of Hadoop.

In addition, I have a long list of people to thank who were not directly involved in my thesis but rather supported me externally. First, I would like to thank my parents who provided their love and support throughout this period. They have always believed in me, although they reside in India, I know they are very proud of me. I would like to thank my sister and my brother-in-law, for their encouragement and support. Also, I am grateful to my in-laws, especially my mother-in-law who helped me design the cover of this thesis.

I would like to express my gratitude to the late Mr. Steven Shephard, who was my landlord in Sunnyvale. I felt as if I was a part of his family. I miss him very much. I am also thankful to my friend, Jonathan Rubin, who always provided support when needed. Many thanks to Brian Taylor for editing my thesis. I would also to thank Jeroen Latour for translating the thesis summary into dutch. I also want to thank my current manager at

Pepperdata, Sean Suchter, for providing support and allowing me to work on the thesis in conjunction with my job.

Last but not least, I am very grateful to my wife, Kriti, for supporting me through all the ups and downs during this time. There were moments, when things became difficult and I almost gave up, but her strong support and motivation kept me going. I could not have completed this journey without her.

Shekhar Gupta
San Jose, November 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

In the era of the modern internet, we are witnessing the tremendous growth in the amount of data that now streams from everywhere: phone communications; credit card transactions; smart televisions; computers; infrastructure of smart cities; sensors installed in cars, trains, buses, planes, etc. Additionally, there has been an explosion of data from social networking and e-commerce websites. For example, Figure 1.1 shows the growth of Facebook users during the period of 2004 and 2010. In 2013, Facebook recorded 1.1 billion users with an average growth of 1.5 percent monthly [3]. As of July 2014, Twitter has more than 284 million monthly active users with 500 million Tweets everyday [4]. As of May 2014, Amazon has 244 million customers and, on average, handles 0.5 million transactions every second [5].

As more and more data is being accumulated, the *data analysis* becomes more challenging. Data analysis is the process to learn the meaning of the data. To understand and show the meaning, the data is generally collected and displayed in the form of tables, bar charts, or line graphs. The process involves finding hidden patterns in the data, such as similarities, trends and other relationships, and learning the meanings of these patterns. Traditional solutions to analyze data are not efficient enough to process the massive data set, which are generally in the order of terabytes (TB) or petabytes (PB) in size. However, the analysis of this data is crucial due to the possible social and financial gains from such analysis. For example, by analyzing historical data or the data on social networks, mishaps such as criminal activities or suicides can be predicted and perhaps prevented [6] [7]. Additionally, customer preferences can be understood by analyzing massive e-commerce websites which can be then used to design future products. Based on the shopping records of customers, new products can be recommended to the customers that they may need or be interested in.

When it comes to managing massive data, *big data* is certainly one of the biggest buzz words in the IT industry. In Big data technologies, massive amount of data (in the order of Petabytes) is collected, stored and analyzed to fetch interesting hidden patterns and insights from the data. Big data can deliver new business markets, and has social and economical impacts, if such huge data is analyzed efficiently. Data analysis techniques

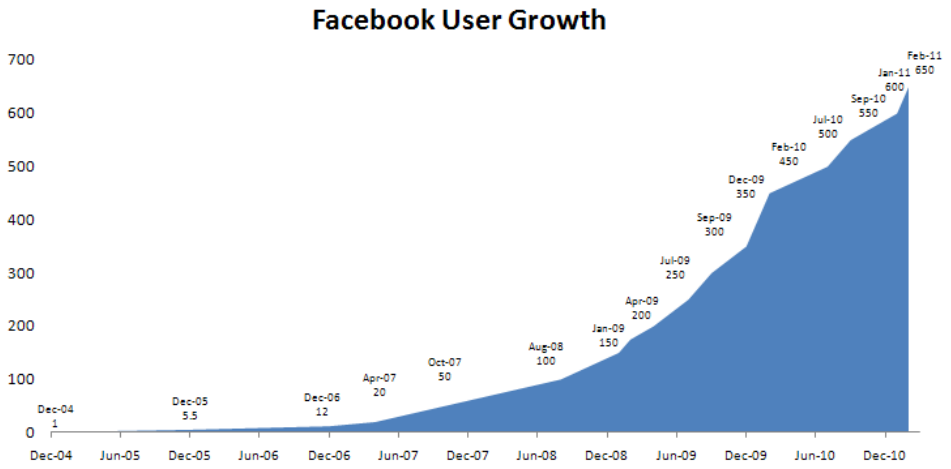


Figure 1.1: Growth of Facebook users (millions) over time

include data mining, text analytics, statistics, machine learning, natural language processing and predictive analytics,. Businesses can analyze their data to gain new insights that can help them to make better and faster decisions. According to IBM, big data has the following three major characteristics [8]:

- Variety: Big data is beyond structured data and includes semi-structured or unstructured data of all all kinds, such as text, audio, video, images, log files, clicks, and more.
- Volume: Big data always comes in huge size. Organizations are flooded with hundreds of terabytes and petabytes of information.
- Velocity: Data sources such as social networks are continuously producing information at an ever increasing rate. The data must be often analyzed in real time.

Unlike traditional technologies, Big data does not directly analyses the massive amount of stored data, rather it breaks data into smaller and incremental pieces. Each of the data pieces are analyzed separately and if needed, results from all the pieces are combined to get the analysis of entire dataset. Due to the large size of the data, the big data analysis is not suitable for traditional Online Transaction Processing (OLTP) or traditional SQL analysis tools.

Performance of OLTP and SQL tools depend on how well the hard disks are performing. Over the years, storage space of hard disks have significantly increased. However, the rate at which data can be read or written hasn't increased in step. Nowadays, 1 TB disks are standard, but the access rate of the disk is still only around 100 MBps. At that rate, reading all the data off the disk would consume more than two and a half hours. This access rate is too slow to satisfy today's IT demands. The obvious solution is to reduce the response time by reading from multiple disks together in parallel. If 100 disks are working in parallel and each is storing 1/100th of the data, then 1 TB data can be read

in less than two minutes. In practice, storing 1/100th of a data set per disk is a waste of resources. All 100 disks should be filled completely, and there should be a shared access of them. As a result, big data organizations are turning to a new class of distributed computing technologies such as Hadoop [9]. Hadoop is an open source software framework that enables the processing of enormous amount of data sets across clustered systems. Hadoop provides a reliable shared storage on commodity hardware, and an analysis system. The storage is provided by the Hadoop Distributed File System (HDFS) [10] and the analysis is provided by MapReduce [11], a software framework for processing large amounts of data. If data is spread across multiple drives, then it's critical to solve the hardware failure issues. Hadoop simply solves this problem by replicating copies of data on multiple machines. If a machine dies then the data stored in the machine can still be accessed from the replicated copies.

Various big data companies use Hadoop to store and analyze their data [12]. Twitter uses Hadoop to store and analyze Tweets, log files, and many other types of data generated across Twitter. Yahoo uses Hadoop to facilitate research for Ad Systems and Web Search. Hadoop is used by Facebook to store copies of internal log and data sources and uses it as a source for reporting, analytics and machine learning. eBay is using Hadoop for search optimization and research. These companies require enormous amounts of disk space to store massive data produced by various sources. Therefore, they have to assemble huge numbers of machines into their Hadoop clusters. For example, eBay has a 532-machine Hadoop cluster, with 4256 cores and around 5.3 PB space. Facebook has two major Hadoop clusters. The first one is a 1100-machine cluster with 8800 cores and about 12 PB space. The second is a 300-machine cluster with 2400 cores and about 3 PB raw storage. At Yahoo, more than 40000 machines are running Hadoop and the their biggest cluster has 4500 machines. These cluster sizes keep increasing as new data is being generated continuously.

The cluster size is mainly determined by the amount of storage required. Many big data industries experience a very high ingest rate of data. The more data are coming into the system, the more machines are needed. Let's consider a hypothetical example to illustrate the growth of Hadoop clusters. Let's assume a company receives 1 TB of new data flows in every day. A growth plan of cluster size can be designed to estimate many machines are needed to store the total amount of data. The cluster growth is projected for a few possible scenarios. For instance, data shown in Figure 1.2 represents a typical plan for flat growth, 5 percent monthly growth, and 10 percent monthly growth.

Figure 1.3 shows how the cluster would grow for the cluster setup shown in Figure 1.2. According to the analysis, if 1 TB is coming in every day, then in one year, the cluster size might increase by 6600 machines, despite 0 percent increment in cluster growth per month. With 10 percent cluster growth per month, and 1 TB data coming in everyday, the cluster size might increase by 10000 in a year. For an extreme case, if the incoming data rate is 3 TB per and the cluster is growing by 10 percent per month, the cluster size might grow by 33000 machines. All these possibilities represent practical cases for many organizations. Therefore, we assume that the cluster size is increasing very fast.

As the sizes of Hadoop clusters grow, the need for a Cluster Management System (CMS) becomes tantamount. CMS is a software tool that provides an abstract view of the performance of a cluster so that energy, resources and floor space can be used effi-

Average daily ingest rate	1 TB	
Replication factor	3 (copies of each block)	
Daily raw consumption	3 TB	Ingest × replication
Node raw storage	24 TB	12 × 2 TB SATA II HDD
MapReduce temp space reserve	25%	For intermediate MapReduce data
Node-usable raw storage	18 TB	Node raw storage – MapReduce reserve
1 year (flat growth)	61 nodes[a]	Ingest × replication × 365 / node raw storage
1 year (5% growth per month[b])	81 nodes[a]	
1 year (10% growth per month)	109 nodes[a]	

[a] Rounded to the nearest whole machine.

[b] To simplify, we treat the result of the daily ingest multiplied by 365, divided by 12, as one month. Growth is compounded each month.

Figure 1.2: Sample cluster growth plan based on storage.

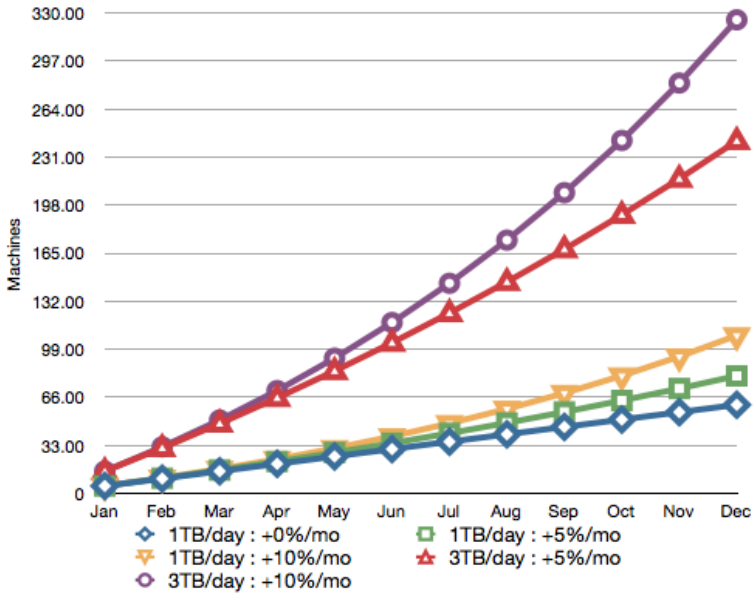


Figure 1.3: Cluster size growth projection for various scenarios.

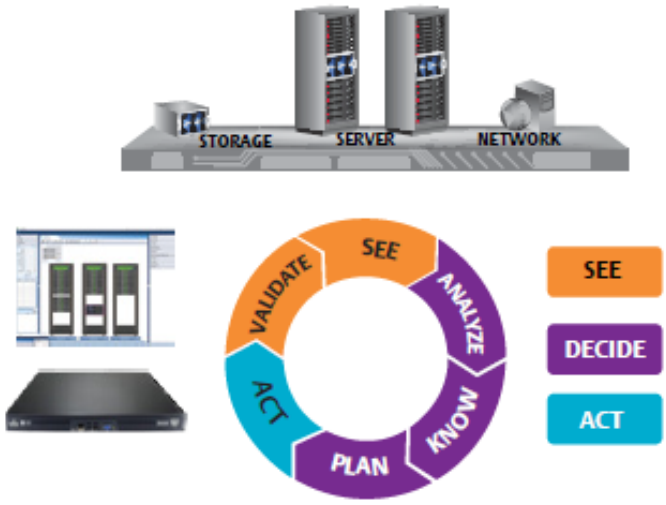


Figure 1.4: Various steps in CMS process. Cluster machines are observed, and their performance is analyzed. Based on the analysis, actions are taken to get the best performance from the cluster.

ciently. CMS software enables administrators to collect, store and analyze data related to performance, resource usage, power and cooling in real time. Generally, CMS software tools, including Nagios [13] and Ganglia [14], produce visualization of data, such as CPU usage, I/O activities, memory usage, and network traffic activities. Visualization of this data can be used by administrators to determine when maintenance is required or when extra capacity needs to be added to clusters.

In general, as shown in Figure 1.4, the CMS is a loop, consisting of multiple building blocks [15]. As shown in the figure, the primary CMS building blocks include, *See*, *Decide*, and *Act*. The data is generally collected from different machines of the clusters. The data constitutes resource, power consumption and temperature metrics of machines in the clusters. Collecting data and validating whether the data is useful or not are part of the building block *See*. The useful collected cluster metrics are analyzed to identify the current state of the cluster. In our work, we interpret the state of the cluster as the performance level of the cluster. The CMS tool also provides consoles to visualize the data, and this visualization is used by administrators to monitor the performance of the cluster. Based on the state of the cluster, plans are generated to run various applications on the cluster. Such analyses and plan generation steps are classified as *Decide*. As part of the *Act*, according to the generated plans, the applications are executed on the cluster.

CMS is one of primary functions of Data Center Infrastructure Management (DCIM). DCIM is a software tool designed to improve the performance and efficiency of IT infrastructures to enhance the business values. DCIM assists administrators by performing the following operations:

- DCIM provides an integrated view of all the physical assets of the data center infrastructure.

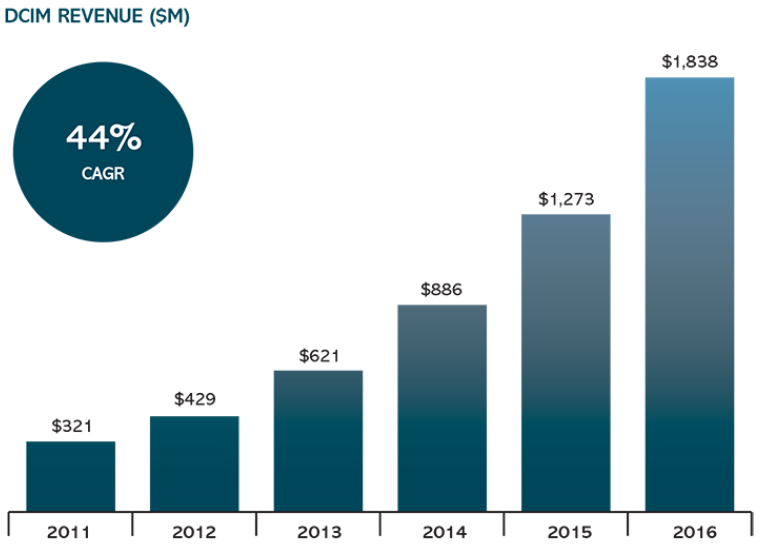


Figure 1.5: 451 Research Market Monitor: Datacenter Infrastructure Management Systems [1].

- DCIM automates the time-consuming, manual process of commissioning new equipment.
- Automation of capacity planning, and forecasting what kind resources will be needed in the future, is taken care by DCIM.
- DCIM technology reduces the energy consumption and energy costs.
- No matter how radically and rapidly IT demands are changing, DCIM must satisfy all those growing IT requirements.

Due to the potential large revenues from DCIM, there has been a tremendous interest from vendors, investors and researchers in making DCIMs more efficient. Figure 1.6 shows the projected growth of DCIM.

Based on a recent survey [1], the DCIM market is $321 million in revenue. It is expected that DCIM sales will grow at 44 percent Compound Annual Growth Rate (CAGR) to reach $1.8 billion in aggregate revenue in 2016.

As the interest is growing in the development of DCIM, it is critical to develop efficient CMS to maintain the cluster assets. For example, for distributed computing, IBM has designed a CMS tool for their AIX (Advanced Interactive eXecutive) operating system to maintain the cluster with lower cost [16]. The CMS is included with the IBM AIX V6.1 operating system, eases the process of cluster administration, by providing a single point-of-control tool to administer the cluster. Recently, CMS tools for Hadoop are gaining some attention. Pepperdata has received more than $20 million funding to manage and monitor Hadoop clusters [17]. The Pepperdata CMS tool monitors and controls the

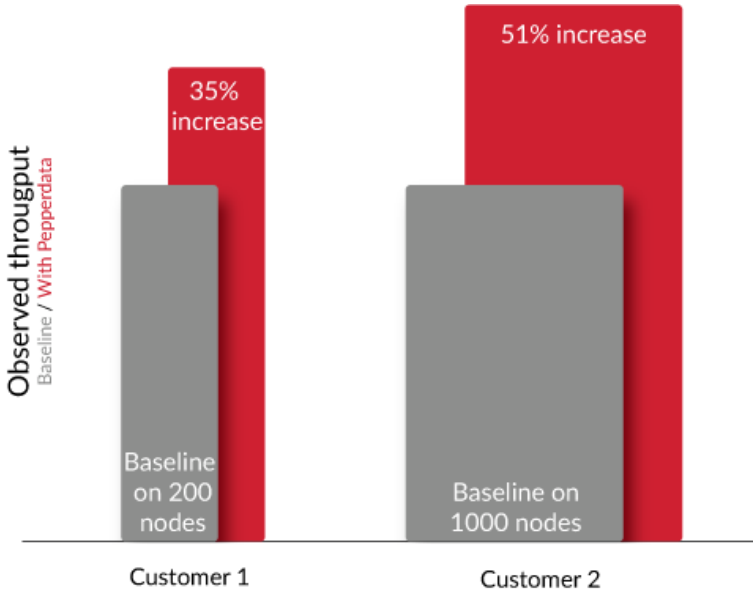


Figure 1.6: Throughput improvement provided by the Pepperdata tool.

resource usage of applications on machines and provides 30 to 50 percent more throughput [18].

Despite the potential revenues from CMS systems, the technology is still in its infancy. CMSs have challenges to further improve availability, utilization, and efficiency, given the increasing cost and demands. These challenges can be addressed by developing more efficient resource management and monitoring units in CMS. The job of the resource management unit is to exploit the available resources most efficiently and maximize the performance of the applications running on the cluster. The monitoring unit monitors the performance of every machine in the cluster. If machines have performance problems, the monitoring unit captures the anomalous behavior of those machines. In later sections, we demonstrate that impact of resource management and monitoring on businesses in the case of general distributed systems. The same kind of impact can be observed if we improve resource management and monitoring for Hadoop. Figure [18] shows the throughput improvements using the Pepperdata CMS with resource management and monitoring of the cluster

1.1. RESOURCE MANAGEMENT

Modern clusters run multiple applications on the machines in parallel to efficiently utilize the cluster. Generally, the resources of a server, including CPU, memory, storage, and network bandwidth are shared among multiple applications. The main challenges of resource management are learning the resource required by various applications, and efficiently assigning these applications to machines. Providing fewer resources than re-

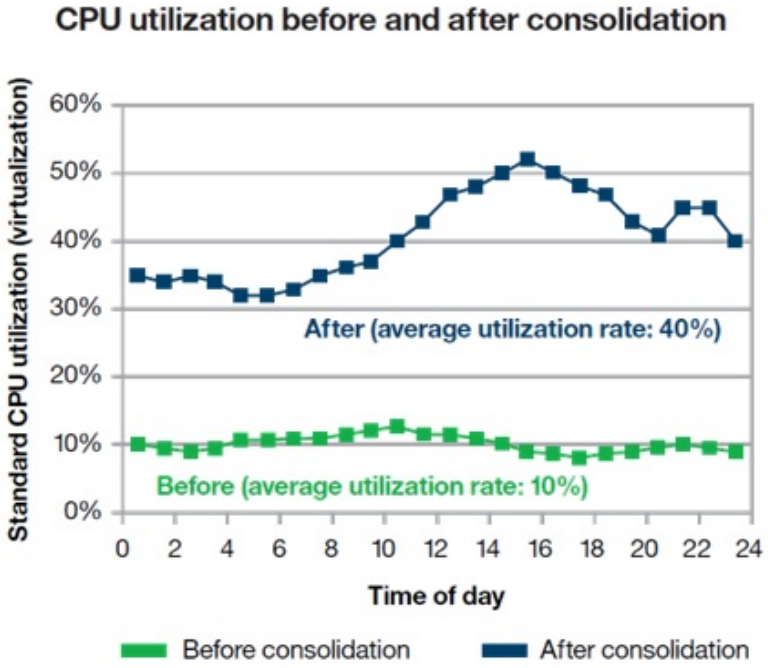


Figure 1.7: CPU Utilization of IBM cluster before and after assigning resources based on demand.

quired can slow down or completely prevent an application from successful completion. On the other hand, if more resources are reserved for an application than actually required, the resources might remain underutilized. A suboptimal resource management plan will severely reduce the performance of data centers.

To study the impact of better resource management on the performance of data, we present multiple examples where the cluster of performance is improved in terms of resource utilization with better resource management.

EXAMPLE 1:
IBM's virtualization platform provides programmatic interfaces that enables policy based automation [19]. Using analytics to model current and future demand on a client's server resources, IBM is able to automate workload placement, quadruple server utilization (shown in Figure 1.7) and reduce the number of servers by five times, with $4 million in savings estimated the very first year.

EXAMPLE 2:
To measure the efficiency of Amazon EC2 cluster, the CPU utilization is measured using 30 probing instances (each runs on a separate physical machine) for a whole week [20]. Figure 1.8 shows the CPU utilization of the cluster when the resources are assigned without considering the nature of the workloads. Figure 1.9 shows the CPU utilization when the resources are assigned based on workloads. Overall, the average CPU utilization is

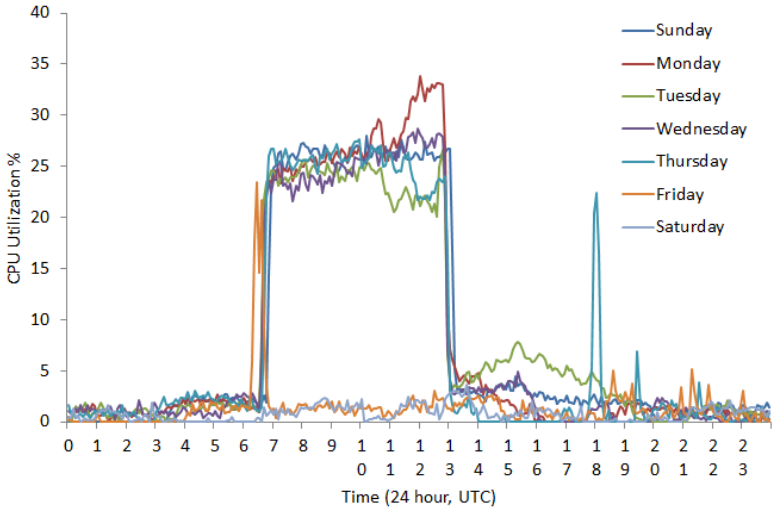


Figure 1.8: CPU Utilization of EC2 cluster before assigning resources based on demand.

not as high as it is expected. The average CPU utilization in EC2 over the entire week is 7.3 percent, which is lower than the maximum utilization of an internal data center. At one virtualized data center, the average utilization is 26 percent, which is more than three times higher than the average utilization of EC2 [20].

The lower CPU utilization is a result of the limitation of EC2; that is, EC2 fixes the CPU allocation for any instance. Even if the host has unused CPU capacity, EC2 would not distribute those free CPU cycles to other instances. Such kind of static allocation is necessary, because Amazon is a public cloud provider. A public cloud provider needs to make sure that virtual instances are isolated from each other, so that one user does not consume all the CPU resources. However, due to such allocation, cluster has to suffer lower CPU utilization. To increase the utilization, all instances running on a physical machine should use CPU at the same time, however, this does not happen very often.

Amazon has many number of customers; thus, there is a possibility to get higher CPU utilization. Figure 1.9 shows the busiest physical machines. It appears that a few instances on these machines are running CPU hungry batch jobs. Two or three instances get busy around same time on Monday, and therefore, the CPU utilization increases significantly However, such kind of busy behavior is occurred only for a few hours in the week, hence, the average CPU utilization is only 16.9 percent. It must be noted that the busiest machine has a lower CPU utilization than the utilization (26 percent) of an internal data center.

These measurements show that a public cloud such as Amazon EC2 does not efficiently utilize its resources. It suggests that there is a need for better resource management tools to use the clusters more efficiently.

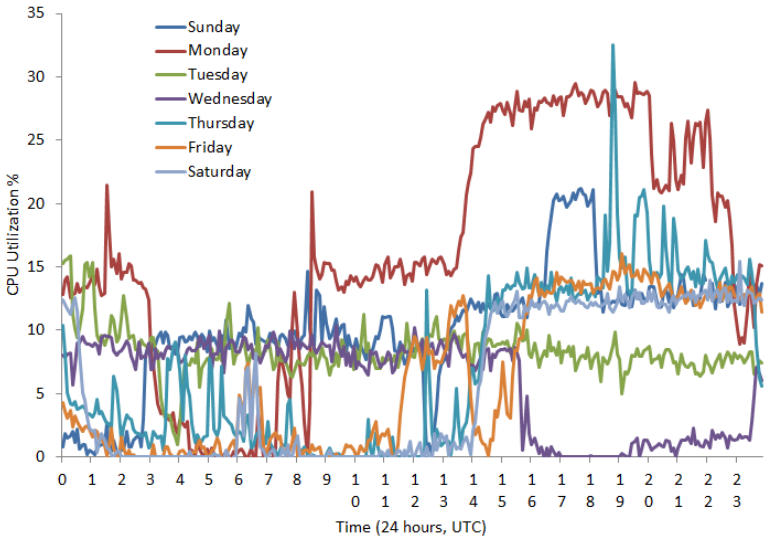


Figure 1.9: CPU Utilization of EC2 cluster after assigning resources based on demand.

1.2. FAULTS

Enterprises invest a great deal of resources to ensure that mission-critical applications run efficiently on data centers. However, such investments do not always give the expected results. Despite the advanced infrastructures, software and hardware faults are still common in many IT organizations. Such faults might shutdown the business for days.

In September 2010, Virgin Blue airline's check-in and online booking systems crashed. On September 26, as hardware failure happened in Virgin Blue's internet booking and reservation IT infrastructure. Virgin Blue business was severely affected by this outage for almost 11 days, and around 50,000 passengers and 400 flights were affected by this outage. Figure 1.10 shows customers waiting for Virgin America flights following a system disruption.

To understand the nature of faults in data centers, we present a data set [21] collected during 1995-2005 at Los Alamos National Laboratory (LANL). The data records all the faults that occurred in the 9-year period, and were responsible for application or machine failure. The data contains faults such as, software faults, hardware faults, operator error and environmental issues (e.g. power outages). The data is gathered from 22 high-performance computing systems, that include 4,750 machines and 24,101 processors.

The LANL data provides the classification of the root cause of failure into various faults, such as software, hardware, human, environment, and unknown. Figure 1.11 shows the frequency of the root cause classes. More than 50 percent of faults are hardware related problems, and around 20 percent of faults are because of software.

There can be multiple reasons for data center failures. Failures can be due to both software and hardware issues. Google's cluster management unit tries to run its data

Figure 1.10: Passengers waiting for Virgin America flights at an airport.

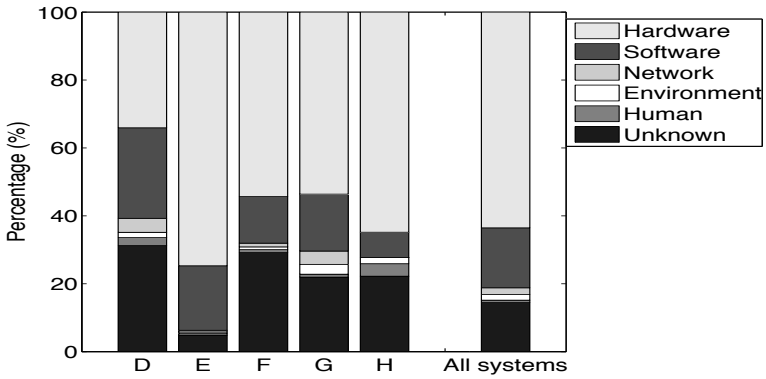


Figure 1.11: The breakdown of failures at LANL by root cause. Each bar shows the breakdown for the systems of one particular hardware platform.

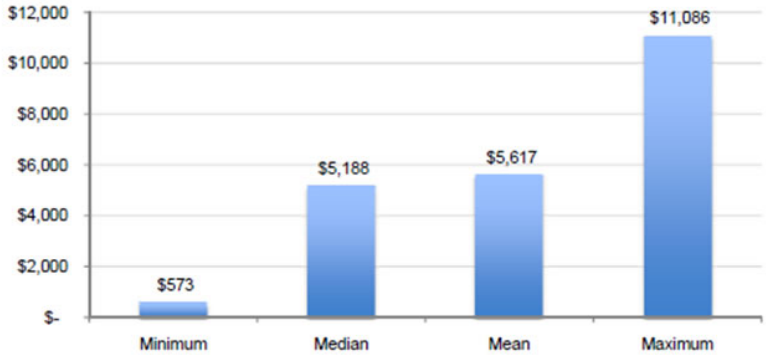


Figure 1.12: Unplanned IT downtime per minute.

centers flawlessly with high utilization. However, it encounters many kinds of hardware failures. At Google [22], the rate of uncorrectable DRAM errors per machine per year is more than 1 percent. The failure rate of the disk drives is 2 to 10 percent.

1.3. MANUAL REPAIR COST

One of the key problems in date center operations is to manually monitor and diagnose machine problems and faults. Sometimes faults are very difficult to diagnose, and therefore IT engineers invest a great deal of their time in finding those faults. Investing human hours to repair machine problems might significantly increase the data center operational cost. Figure 1.12 shows the estimated per-minutes costs of unplanned downtime in an IT organization. A study [23] illustrates a setup of 120 machines, each with a Mean Time To Failure (MTTF) of 500 minutes, and the repair staff average is 20 minutes to fix the system. To maintain the performance of the system up to 96 percent throughput, a staff of 10 IT engineers is needed. Extrapolating this setup to 50000 servers, each with an MTTF of 50000 minutes, a staff of more than 40 IT engineers is needed to achieve the same throughput. A staff of more than 40 IT engineers is needed. The average wage of an IT engineer is $44.85 per hour, which translates to $93,267 per year by counting 40 hours per week times 52 weeks per year. Therefore, the entire staff will IT cost around $3.3 million per year just to fix the errors so the data center can perform at a certain throughput. Generally, machines are not operational while the problems are being diagnosed. The time to fix problems can be defined as the downtime of the system. Such downtown will have an additional impact on a businesses that are running on those data centers.

On average, between $84,000 and $108,000 (US) lost is reported for every hour of IT system downtime, according to estimates from studies and surveys performed by IT industry analyst firms [24]. Moreover, financial services, telecommunications, manufacturing and energy industries lose their revenue even at a higher rate during downtime. For the average recovery time of 134 minutes, the average costs were $680,000, for a data center outage.

1.4. RESEARCH GOALS

In this chapter, we have observed that an efficient CMS tool is vital for achieving the best results from Hadoop clusters. An efficient tool may improve the revenue and productivity of the cluster. The task of implementing such an efficient CMS becomes challenging because the workload conditions and the machine characteristics are continuously changing. Therefore, in this thesis our primary goal is the following:

> "Implement a CMS tool for Hadoop to achieve the optimal performance from the clusters where the cluster states are continuously changing."

In order to accomplish the above goal, we set the following sub-goals in designing an automatic high throughput CMS system:

1. **Automatic Resource Management:** The CMS system should automatically assign workloads to machines such that the resource utilization is optimal and the applications can run efficiently. In the case of changing applications, the system should autonomously change its workload assignment to achieve the optimal performance.

2. **Self Adaption:** Due to degradation or faults, the performance of machines can change over the time. Rather than using manually fixing the performance problems, the CMS should be self-adaptive to automatically identify the faults or degradations in machines. In order to make resource manager adaptive to these changes in real time, these changes should be communicated to the resource manager.

3. **Minimize Downtime:** To minimize the downtime, the cluster's production should not stop while doing the resource management and identifying performance problems in machines. Therefore, there should be minimum human involvement in the CMS, and the cluster should be at the maximum production level.

In this thesis, we use the word "optimal" for scheduling policy, resource usage, and performance. A scheduling policy is optimal when it maximizes an objective function. Resource utilization is optimal when overall resource utilization of each cluster host is at the best possible level. Performance can be defined in terms of a certain objection; optimality means maximizing that objective function.

1.5. PROPOSED CMS DESIGN

To accomplish the above goals, in this section, we present the design of our CMS tool. The objective of our CMS design is to maximize the performance of Hadoop clusters; however, while designing the CMS we make sure that the design can be easily extended to any other distributed computing platform.

Performance of a cluster mainly depends on the performance of every machine in the cluster and how applications are scheduled on machines. To achieve the best possible production from a cluster, the available resources need to be exploited efficiently. The decision on how to use resources, which workload and how much workload to assign on which machine, is made by the scheduler. Scheduler enables us to achieve the

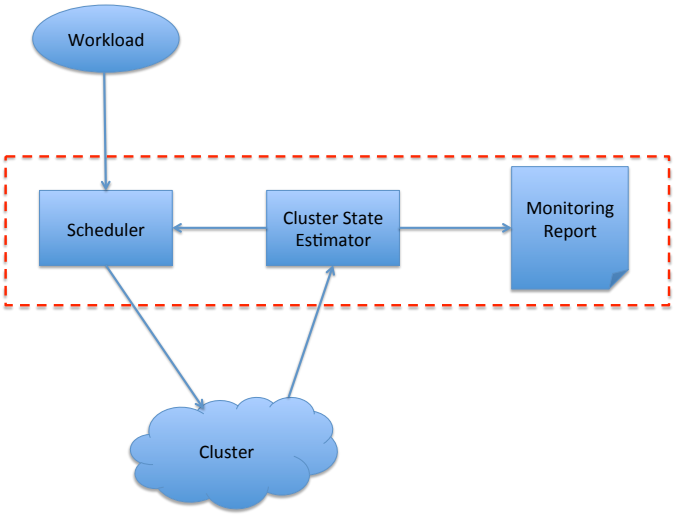


Figure 1.13: The proposed design of CMS system

first sub-goal, *Automatic Resource Management*. Therefore, the scheduler is very important component of our CMS design. In our design, scheduler is the unit that take cares of resource management of the cluster. The other important component in our design is the *cluster state estimator (CSE)*. The CSE keeps learning the state of the cluster, in terms of the performance of each machine in the cluster. The conceptual design of CMS, which incorporates the scheduler and CSE is shown in Figure 1.13. We can achieve the second sub-goal, *Self Adaption*, by efficiently implementing CSE.

In this work, we design a self-adaptive agent for CMS which performs scheduling and monitors the performance of the clusters. According to our design, workloads are submitted to the scheduler, which distributes the workloads to different machines. In principle, the scheduler assigns workloads to machines that maximize a certain objective function. In our case, we choose the performance of the cluster as our objective function. In our design, we assume that each workload has a certain kind of characterization and different kinds of workloads are concurrently submitted to the cluster. The performance of a workload depends on the characterization of the workloads and the current state of the cluster. To monitor the performance of the cluster and to generate the optimal scheduling decision, the current belief about the cluster state is estimated. To achieve the third sub-goal, *Minimize Downtime*, the performance of actual workload is analyzed in the real time to estimate the cluster state. Along with the scheduling, the output of the cluster state estimation is used to identify the performance problems in the cluster. In our design, we provide an automated solution to diagnose performance problems. Additionally, we also provide the monitoring data that can be used by the cluster administrators to identify the problems. In next section, we provide our contributions as part of the implementation of the proposed CMS design.

1.6. CONTRIBUTIONS

In this thesis, we address the challenges of Hadoop CMS design with the following contributions:

1. We provide an online framework to learn the resource usage of incoming workloads. The different workloads have different resource characterization, because the resource usage of a workload depends on the functionalities of the workloads. To implement an efficient resource manager, the resource usage of the workloads must be known. In practice, this information is not provided in advance; therefore, they must be learnt. An efficient resource managers helps us in achieving the first sub-goal that we described earlier.

2. We present the design of the scheduler to achieve the first sub-goal. To implement a scheduler, an objective function needs to be defined. The job of the scheduler is to optimize the objective function. We define and provide the analytical formulation of various objective functions. Each objective function formulates the performance of the cluster under different scenarios. Our objective function is formulated in terms of resource usage of workloads and the performance of cluster machines.

3. We propose an online tool to learn the performance of every machine in the cluster. The performance of a machine is also seen as the state of the machine. The machine states are continuously estimated to monitor the performance of the cluster in real time. This monitoring tool derived as part of this contribution helps us in achieving the second sub-goal.

4. To keep the production at a higher level, the scheduling and learning of various parameters is performed in real time. Our CMS does not stop the production to perform any of its task. Getting non stop production from a cluster where machines are failing dynamically, helps us in achieving the third sub-goal.

5. Many distributed frameworks assume that the underlying infrastructure is homogeneous, meaning that machines are identical to each other in terms of their performance. However, this assumption is not true in practice. Machines in the cluster are collected from different generations, and therefore the cluster naturally becomes heterogeneous. Additionally, the performance of machines degrade over time, and the performance of different machines will degrade differently. Therefore, as part of our contribution, we make sure that our CMS works for both homogeneous and heterogeneous clusters. In other words, we achieve all the sub-goals for both homogeneous and heterogenous clusters.

1.7. THESIS ORGANIZATION

The contributions mentioned in the previous section are described in six chapters of this thesis. Chapter 2 provides a brief description of various frameworks in distributed computing, and subsequently provides a detailed description of Hadoop and its various components. The chapter also describes related work in the field of resource management and monitoring of Hadoop clusters. In Chapter 3, we propose our online approach

to learn resource usage of applications. In the chapter, we also introduce a performance model. In Chapter 3, we also introduce a scheduler to assign workloads in heterogeneous cluster. In this chapter, we use the performance model derived in Chapter 3. In Chapter 4, we start with the shortcomings of the scheduler derived in Chapter 4. Subsequently, we introduce a new scheduler that eliminates all the shortcomings of the scheduler derived in Chapter 4. In Chapter 5, we provide our approach to monitor and diagnose the performance problems in heterogeneous Hadoop clusters. In Chapter 6, we provide a scheduling framework, which is inspired from the Pervasive Diagnosis approach [25]. Finally, in Chapter 8, we draw conclusions and present recommendations for future work.

1.8. PAPERS

As part of this thesis, we published the following papers that summarize the work described in various chapters of the thesis.

1. Chapter 3 and 4 are based on the following:
   Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan de Kleer and Cees Witteveen. ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters. In Proceedings of the *USENIX: 10th International Conference on Autonomic Computing (ICAC-2013)*. SanJose, CA, USA, June 26 - 28, 2013

2. Chapter 3 and 4 are based on the following:
   Shekhar Gupta. An Optimal Task Assignment Policy and Performance Diagnosis Strategy for Heterogeneous Hadoop Cluster. In the Proceedings of the *Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013)*. Bellevue, Washington, USA, July 14-18, 2013

3. Chapter 5 is based on the following:
   Shekhar Gupta, Christian Fritz, Bob Price, Johan de Kleer and Cees Witteveen. Continuous State Estimation for Heterogeneous Hadoop Clusters. In Proceedings *24th Int'l Workshop on the Principles of Diagnosis (DX'13)*. Jerusalem, Israel, October 2013

4. Chapter 5 is also based on the following:
   Shekhar Gupta, Christian Fritz, Johan de Kleer and Cees Witteveen. Diagnosing Heterogeneous Hadoop Clusters. In Proceedings of the *23rd Int'l Workshop on the Principles of Diagnosis (DX'12)*. Great Malvern, U.K, August 2012

2

BACKGROUND AND RELATED WORK

In the previous chapter, we mentioned the goals of this thesis that are related to the performance improvement of big data infrastructure. In our work, we choose Hadoop as our infrastructure to solve big data problems. Hadoop is widely used to store and analyze massive amounts of data collected from various sources. The challenge of how to store and analyze data in a distributed manner is not new. In the high-performance computing community, there have been many distributed computing platforms. In the following sections, we discuss other possible systems to solve big data problems and why they are not frequently used.

2.1. RELATIONAL DATABASE MANAGEMENT SYSTEM

A database is used to store information in such a way that information can be easily retrieved from it. In general, a relational database stores information in the form of table with rows and columns [26]. Relational databases are not only relatively easy to create and access, but also very easy to extend. They have become a common choice among businesses to store and analyze information. A relational database management system (RDBMS) is a framework that allows one to create, update, and administer a relational database [26]. The most popular RDBMS programs use the Structured Query Language (SQL) to access the database [27]. SQL statements are in the form of queries to retrieve information from the database.

The SQL disk access patterns are dominated by seeks. Seeking is the process of moving a disk head to a desired location on the disk where the data resides [28]. In the previous chapter, we discussed that the data transfer rate for disk is not improving fast enough, and the seek time improvements are even slower than that of transfer rates. Due to seek operations, reading from and writing to a large set of data takes longer than it does to stream through it. This is because streaming uses the transfer rate to batch processing, which speeds up access to the data. Therefore, an RDBMS is inherently slower

than MapReduce to analyze massive amounts of data because MapReduce analyzes data in a batch. Moreover, an RDBMS is suitable for datasets that are continuously updated, whereas MapReduce is better suited to application where data is written once and read many times. Another reason, why an RDBMS is not a good big data application is the structure of data. An RDBMS operates on structured data, where data is organized in the form of database tables or XML documents [28]. On the other hand, in the big data world, the data arrives in the form of unstructured data such as plain text or images. An RDBMS is not designed to work with unstructured data. On the other hand, MapReduce works well on unstructured or semi-structured data.

Generally, a number of SQL queries are grouped together to accomplish a certain job. To handle multiple jobs, SQL comes with a job scheduler, for example the Oracle scheduler [29], which is implemented by the procedures in the SQL package. The scheduler allows enterprises to efficiently manage and plan these jobs. The scheduler handles the execution of these jobs based on the business environment. When jobs are competing for resources, the scheduler allocates resources to them based on the business need. The business-oriented scheduling is implemented using following ways:

- Jobs that have similar characteristics are grouped together into job classes [30]. The scheduler controls the resource allocation to each class. This classification ensures that critical jobs get higher priority and sufficient resources to complete. For instance, if there is a critical project to load a data warehouse, then the scheduler may combine all the data warehousing jobs into one class and give priority to it over other jobs by allocating most of the available resources to it.

- The scheduler further extends the prioritization of jobs to a next level by dynamically changing the priority based on some criteria [30]. Over time, the definition of a critical job may change, and the scheduler accommodates these changed priorities among jobs over that time window. For instance, the extract, transfer, and load (ETL) jobs can be considered as critical jobs during non peak hours but not as critical during the peak hours. Other business related jobs may need higher priority during the peak hours. In these cases, the priority among the jobs can be changed by dynamically changing the resource allocation of each class.

Scheduling based on priority is always useful for any kind of application, including big data applications. The existing schedulers also take priorities into account while making the scheduling decision. Scheduling only based on priority does not guarantee optimal performance. Different priorities are assigned to the different users that use the same cluster. In general, a user submits more than one application at a time. To achieve the best performance, the scheduler should efficiently run all applications submitted by one user. The monitoring tool provided SQL framework only monitors the success of applications. This framework does not try to infer where the failures happened and why; however, these inferences could be very useful to efficiently run the cluster in future.

2.2. GRID COMPUTING

Grid computing is the another branch of distributed computing, which has been used used to process massive amount of data for years. A grid is defined as a collection of

machines, resources, members, donors, clients, hosts, engines, and many other such items [31]. They all contribute some amount of resources, such as CPU, storage and memory to the grid as a whole. Some resources of the grid may be accessed by all users, and access to some resources is restricted to some specific users.

The CPU is the most common resource shared by the processors of the machines on the grid. The processors can vary in terms of speed, architecture, software platform, memory and storage. Such sharing of CPU cycles enables the massive parallel CPU capacity to the grid, which is one of the most attractive features of the grid computing.

Storage is the second common resource shared by the machines on the grid. The grid provides an interesting view of storage, which is also called a *data grid* [31]. The storage in the grid is used in specific ways to increase capacity, performance and reliability of data. Capacity is increased by using the storage on multiple machines with a shared file system. A single file or database is distributed over multiple storage devises and machines. A single uniform name space is provided by a shared filed system. Because of the uniform name space, it is easier for users to refer data in the grid, without worrying about the exact location of the data. Many grid computing platforms use mountable file systems, such as Network File System (NFS), Distributed File System (DFSTM), Andrew File System (AFS), and General Parallel File System (GPFS).

Grid Computing communities have been working with large-scale data for many years. They use APIs, such as Message Passing Interface (MPI) to implement frameworks to process data. In principle, Grid Computing distributes the workloads across a cluster of machines, which has a shared flie-system, hosted by a Storage Area Network (SAN). Such approaches works well for compute intensive applications. It does not perform well when machines need to access to larger amount of data, because the network bandwidth becomes the bottleneck and machines become idle. On the other hand, Hadoop uses a distributed file system, and computation is performed at the machine where data is stored locally. This method avoids sending data over networks which might easily saturate the network bandwidth. In terms of implementation, MPI programming is much more difficult than developing a MapReduce program. MPI requires users to write low-level C programs, which is far more challenging than writing high level MapReduce programs.

To increase the reliability in the grid, expensive hardware is needed. The hardware is generally is consist of chips and redundant circuits, is contains logics to recover from failure. Machines use hardware redundancy by duplicating processors so that if one processors fails, other can be used. Power supplies and cooling systems are also duplicated. All of these duplications build a reliable system, but at a very high cost.

Most grid computing platforms use some kind of scheduler to run incoming applications on machines. In the simplest scenarios, applications are assigned to machines in a round-robin fashion. This scheduling policy is generally suboptimal, and there are more advanced schedulers which maximize the performance of applications over the grid. Some schedulers implement the scheduling policy based on a job priority system. This is sometimes done by creating several job queues and assigning a different priority to each queue. As soon as a machine becomes available, a job from the higher priority queue is scheduled on the machine. Schedulers can also use other kinds of policies, which can be based on various kinds of constraints on jobs, users and resources A

few schedulers also monitor the progress of scheduled applications. If the application crashes due to a system or network failure, the scheduler would automatically resubmit the application.

2.3. VOLUNTEER COMPUTING

Projects such as SETI@home [32], Search for Extra-Terrestrial Intelligence, run client software on voluntarily donated CPU time from otherwise idle computers to analyze astronomical data to find signs of life beyond Earth. The concept of donating idle computer cycles by users all around the world to solve a common problem is known as *volunteer computing* [33]. Most of the volunteer computing projects, including SETI@home use the framework BOINC [34] for the implementation. Therefore, we will refer BOINC as the volunteer computing framework.

In volunteer computing projects, the problem is divided into small chunks, which are sent to computers around to the globe to analyze the problem. These chunks are generally created by dividing a very large dataset into smaller sets. Each computer runs a client provided by the project administrators. The client runs in the background and waits for the computer to go idle. Once the computer is idle, the client starts receiving small chunks from the main server and starts analyzing them. When the analysis is done, the results are submitted back to the server, and the client starts working on another chunk. The same process continues until the computer is idle.

Although BOINC seems a suitable framework to solve big data problems, there are a number of limitations that make it unsuitable for big data. The SETI@home problem is mainly CPU intensive, and the time taken by computers to process the data is much higher than transferring the data from server to volunteer machines. In some cases, a big data problem might not use the CPU very extensively, and then bandwidth limitation would be a problem for SETI@home. BOINC runs a custom program on every computer for different chunks of data, and making frequent changes to the custom program is not easy. On the other hand, Hadoop provides the MapReduce framework, where users can easily write their own programs and modify them.

There are many other projects like SETI@home that use BOINC as their framework. A volunteer can attach their machines to any set of BOINC projects. The client maintains a queue of jobs, typically from different projects, and runs them on the volunteer machines. On the machines, the jobs are executed based on the scheduling policy of local operating system. The scheduler reports back the list of completed jobs and requests the new jobs to finish. In the context of this work, we only focus on the client's scheduling policy. Weighted round robin is the baseline scheduling policy of the BOINC client. Here weight is determined by how much resource are shared and projects are given time in proportion of their resource share. On top of the weighted round robin, BOINC also incorporates priorities for jobs. The priorities are decided based on the jobs' deadline, or whether the job needs CPU or Graphics Processor Unit (GPU). Generally, jobs that require GPU are given higher priority.

Volunteer computing consists of machines that are diverse in terms of hardware and software, reliability, availability, network connection, and other resource specific properties. In other words, rather than being stored in a sophisticated cluster environment, the resources are distributed all over the globe. Unlike cluster settings, there is no guarantee

about the reliability of resources. There are no monitoring tools to determine whether machines are working normally or there are any failures. However, due to the fact that volunteer computing has massive resources, it always runs jobs on multiple hosts to ensure the reliability. Therefore, fault resilience is an important challenge for volunteer computing platforms such as BOINC.

BOINC also uses a similar reliability approach, know as replicated computing [35]. In replicated computing, each job is sent to multiple hosts to process. If the results obtained from multiple hosts agree, they are assumed to be correct. Otherwise, more hosts are issued and until results from a certain number of hosts is consistent.

Due to these challenges with the existing distributed system infrastructures, Hadoop is widely accepted by the big data community. Therefore, in this thesis, we choose to use Hadoop as our big data infrastructure. In the next section, we provide a detailed overview of Hadoop architecture and its various components.

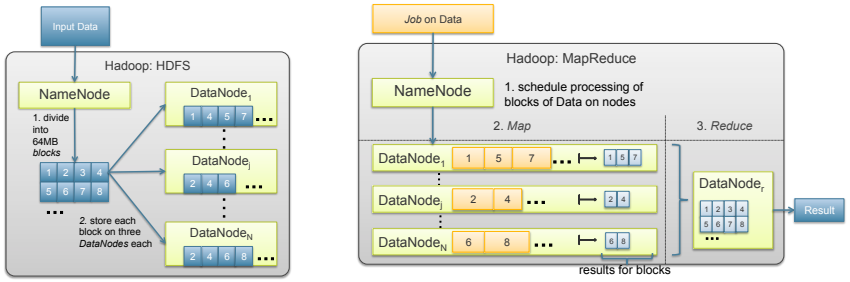
2.4. HADOOP

Hadoop is a free, JAVA-based platform for distributed computing that currently is the de facto standard for storing and analyzing very large amounts of data. Instead of relying on expensive, proprietary hardware and different systems to store and process data, Hadoop allows distributed parallel processing of massive amounts of data across cheap, industry-standard machines that both store and analyze the data, and can be easily scaled. A Hadoop cluster consists of one *master* and many *slave* (tens to thousands) machines. From any number of different sources, Hadoop can handle many kinds of of data: structured, unstructured, logs, images, audio files, email, communications records. Even when different types of data have been stored in unrelated systems, it can be dumped into the Hadoop clusters with no prior need for a schema. In other words, there is no need to know what kind of queries will be made from data before the data is stored. Hadoop lets us decide later, and time can reveal questions we never even thought to ask. For example, let's assume a Hadoop cluster is storing large amount of text files. Different kinds of queries can be made from the text files, such as, counting total number of words, computing frequencies of certain words or other statistical analysis.

While storing the text files, Hadoop doesn't know what operations will be performed on the data in the files. Once data is written, users write their own functions to process the data. Hadoop has two primary components, *HDFS* for storage and MapReduce for processing. The following sections briefly describe these components of Hadoop.

2.4.1. HDFS

Hadoop Distributed File System (HDFS) is an open-source implementation of the Google filesystem [36], called HDFS in Hadoop, and Google's MapReduce framework [11]. HDFS is able to store tremendously large files across several machines and, using MapReduce, process these files in a distributed fashion, moving the computation to the data, rather than the other way round. Using Hadoop, we can run many applications on systems with hundreds to thousands of machines which involves data in order of terabytes. HDFS is implemented to enable fast data transfer among machines, and keep running system uninterrupted under machine failures. The implementation reduces possibility catas-



(a) HDFS (b) MapReduce

Figure 2.1: HDFS stores files in blocks of 64MB, and replicates these blocks on three cluster nodes each. For a new *job* MapReduce then processes (*maps*) each block locally first, and then *reduces* all these partial results in a central location on the cluster to generate the end result.

trophic system failures, in case of number of machines become dead.

Figure 2.1 depicts how Hadoop stores and processes data. When a data file is copied into the system, it is divided up into *blocks* of sizes such as 64MB or 128 MB. Each block is stored to more than one machines depending on the replication policy of the deployed Hadoop cluster . The replication provides both fault-tolerance and performance (Figure 2.1(a)).

The loss of a single disk or machine does not destroy the file from the system and a given block can be read from multiple machines, which improves the throughput. HDFS provides data availability by continuously monitoring the machines in a cluster and the blocks stored on those machines. Individual blocks include checksum. When a block is read the checksum is verified, and if the block has been damaged, it will be automatically restored from one of its replicas. If a machine fails, all the data that was stored in that machine is copied to some other machine from the collection of replicas stored in other machines. As a result, HDFS runs very efficiently on commodity hardware. It tolerates and compensates for failures in the cluster.

2.4.2. MAPREDUCE

Once the data is loaded in HDFS, computational *applications* can be executed over this data. New applications are submitted to the Master (Figure 2.1(b)), which is called NameNode. The Master will schedule *map* and *reduce tasks* onto the DataNodes.

- A map task processes one block and generates a result for this block, which gets written back to HDFS. Hadoop will schedule one map task for each block of the data, and it will do so, generally speaking, by selecting one of the three DataNodes that is storing a copy of that block to avoid moving large amounts of data over the network.

- A reduce task takes all these intermediate results and combines them into one, final result that forms the output of the computation.

Hadoop divides the input data into the fixed size pieces called *splits*. For each splits,

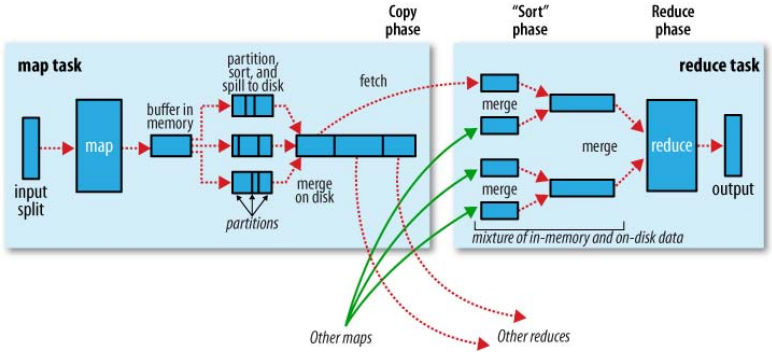


Figure 2.2: Data flow in Map and Reduce tasks.

Hadoop creates a map task, and a user defined map function is executed for each split. When a map function starts producing the output, it is not simply written to disk. The process takes advantage of buffering writes in memory and does some presorting for efficiency reasons. Because, data can be read at a faster rate from the memory, and if the data is presorted, then the reducer can easily generated the sorted data. Figure 2.2 shows the data flow in the map phase.

Map tasks write their output to circular memory buffer. As soon as the buffer is full, the content of buffer is spilled to the disk. Data is first divided into partitions, before its written to the disk. The partitions are used by reducers. Within each partition, data is sorted by key, and in case of a combiner function, the sorted data is combined according to the combiner function. In summary, mappers are either reading or writing data to the disk or it's processing the data as specified in map or combiner function.

To use Hadoop infrastructure, a programmer needs to write applications in the form of Map and Reduce functions. The functions are used to implement the computation in distributed manner, and they operate on the low level unit of text such as lines or words. The MapReduce code is deployed on the each machine of the cluster. The format of data can be specified in the code, and simple algorithms can be implemented. `WordCount` is a classic example of MapReduce program, which is used to calculate the number of words in a large file. The Map function calculates number of words per line of a file by breaking lines into words using a tokenizer, and then by counting words. The reduce function aggregates the number of words computed by each mapper to get the total number of words in the file

2.4.3. DAEMONS

There are two main daemons in Hadoop that accepts jobs from users, run them on the cluster, and make sure that jobs are successfully completing in the cluster. The two daemons are jobtracker and tasktracker:

JOBTRACKER

The jobtracker is the master process, which is responsible for accepting jobs from users, scheduling tasks on machines and monitoring the cluster. There is one jobtracker for

every MapReduce cluster, which means if a the jobtracker dies then the overall cluster fails. Jobtracker monitors the health of slave machines using the heartbeat protocol. Each slave machine sends heartbeats to the jobtracker with a regular interval. The heartbeat message also contains the available resources on machines to run map and reduce tasks. The process of deciding which task will be executed on which machine is known as task scheduling. If there are tasks that need to be completed and, through heartbeat protocol, the scheduler learns that a machine has free resources, then tasks will be executed on the machine. Hadoop comes with a few default schedulers. The description of those schedulers is provided in the later part of this chapter.

TASKTRACKER

The second daemon, tasktracker, which runs on every slave machine, is responsible for accepting tasks from jobtracker, executing those tasks, and reporting progress to jobtracker periodically. The tasktracker keeps track of free and used resources on the machine. Resources on machines are quantified as the number of slots, which indicates the number of parallel tasks. One slot executes one task. If a slot is running a task then it's classified as used; otherwise, it's unused and free to accept tasks. A designated number of slots are provided to map and reduce tasks. When a cluster starts, each tastracker is configured with a fixed number of map and reduce slots. This configuration can not be changed while running jobs on the cluster. When the cluster starts, all the slots are free. Upon receiving a task assignment from the jobtracker, the tasktracker starts filling the free tasks and marking them as used tasks. At a regular interval, the information about free and used slots is sent back to the jobtracker with the heartbeat message. The jobtracker uses this information to make its scheduling decision.

2.4.4. SCHEDULER

Jobtracker runs a scheduler to allocates resource to each application by distributing their tasks on various machines. The scheduler allocates tasks on machines to optimize certain objective function such as capacity utilization or fairness. Hadoop scheduler only does the scheduling, it does not monitor the status of tasks or applications. It also does not restart a task if it is failed on a machine, due to software or hardware failure. Hadoop scheduler is used a plug-in component. Examples of Hadoop schedulers are CapacityScheduler and FairScheduler:

CAPACITYSCHEDULER

The CapacityScheduler [37] allows sharing of a cluster among many users while giving each user a guaranteed capacity in terms of cluster resources. The main idea behind the scheduling policy is that the cluster resources are shared by various organizations who provide funds to the cluster based on their computing requirements. This system gives an additional benefit to the organizations in that they use any free capacity not being used by others. In this way, the organizations gain more flexibility at a less price. Sharing cluster capacity across organizations requires a strong support for multi-tenancy since each organization is guaranteed a certain amount of capacity. The CapacityScheduler ensures that a single application or user or queue is not taking a disproportionate amount of resources in the cluster. Also, the CapacityScheduler limits the initialized or

pending applications submitted from a single user and queue to enable fairness and stability of the cluster.

FAIRSCHEDULER

Fair scheduling [38] is a policy that assigns applications to machines such that all applications, on average, get an equal share of cluster resources over time. Initially, when there is a single application running, the application uses the entire cluster. Subsequently, when more applications are submitted, the resources that free up are allocated to new applications, so that each application roughly gets an equal amount of cluster resources. This allows the short application to finish in reasonable time while not starving long-lived applications. Moreover, FairScheduler is a reasonable tool to share a cluster among a number of users. Finally, FairScheduler works with application priorities—the priorities determine what fraction of total resources should be allocated to which application.

2.4.5. SHORTCOMINGS

Although Hadoop satisfies the requirements of big data problems, but the existing design has lots of room to improve the performance. The existing Hadoop design has the following shortcomings:

HOMOGENEITY ASSUMPTION

Hadoop, as of the current version, does not take any performance differences between the machines into account during the scheduling phase, but assumes a homogeneous cluster. The current Hadoop version assumes that machines in the cluster are equally fast with regard to CPU, disk I/O, RAM, and network bandwidth—the key-contributors to task completion time. Due to the homogeneous assumption, the same number of Map and Reduce slots are allocated on every machine.

IGNORING RESOURCE REQUIREMENTS

During the act of task scheduling, Hadoop does not take into account the actual resource requirements of tasks. Tasks belonging to different applications might have different resource usage; however, Hadoop assumes that tasks have identical resource requirements regardless of which application they belong to.

SUB-OPTIMAL PERFORMANCE

During the scheduling phase, Hadoop does not try to maximize any performance related metric. Hadoop uses a very naive scheduling policy, where it allocates a task to a tasktracker which reports first with the heartbeat.

LIMITED FAULT TOLERANCE

Even though the fault tolerance feature of Hadoop is widely acknowledged, the technology itself is still in its infancy. One of the problems that remains unsolved in the general case is the detection of faults and performance issues in the cluster. A Hadoop cluster may consist of hundreds to thousands of nodes and there can be various kinds of faults in any of the nodes. One can distinguish two types of faults, *hard faults* and *soft faults*.

Crashing a node, disk failure, or network failure can be seen as hard faults, resulting in failing jobs. Soft faults on the other hand, appear in nodes still processing assigned tasks successfully, but at a lower than usual rate. Limping hardware, unnecessary background processes, or poor task scheduling resulting in overloaded nodes can all lead to soft faults. These faults create resource congestions such as CPU or I/O or network congestion, resulting in slowdowns. Hadoop uses a heartbeat protocol to detect hard faults. However, due to their dynamic behavior, detection of soft faults remains challenging.

SINGLE POINT FAILURE

With the current design of Hadoop, the jobtracker runs on a single machine. Administrators try to run jobtracker on the most reliable machine, but no matter what, any machine can die. If the jobtracker dies, then the entire cluster is down, which can severely harm the productivity of clusters.

As we discussed above, Hadoop has many shortcomings, and the Hadoop community experiences these challenges, especially as they relate to efficient scheduling and resource management. Additionally, due to the existing design of Hadoop, a team of Yahoo engineers ran into a number of scalability problems, where they had a 4000-plus node cluster. The team learnt that it is very difficult to run jobtracker on one machine because it has multiple responsibilities, which significantly increases the resource requirement of jobtracker. Further, upgrades and single point failure issues of the jobtracker are very difficult to handle. The Hadoop community is well aware of all these issues and, therefore, created the next version Hadoop, which is known as YARN or "Yet Another Resource Negotiator" [39]. In the next section, we provide a detailed description of YARN. Subsequently, we also provide the related approaches that address the challenges of scheduling and resource management of Hadoop.

2.5. YARN

YARN is the next generation compute and resource management framework in Hadoop. YARN is a tool that decouples MapReduce's resource management and scheduling capabilities for the data processing, which extends Hadoop's data processing capabilities and a broader array of applications can be executed. For example, using YARN, interactive queries and streaming data applications can be executed with MapReduce applications on the same Hadoop cluster. In other words, YARN is not limited to MapReduce applications only, rather it provides a platform to other applications, such as Spark [40], Tez [41] and Slider [42]. In this thesis, we limit ourselves to only MapReduce applications; therefore, we will not be discussing other applications in detail. Figure 2.3 illustrates the YARN architecture.

To enhance Hadoop's scalability, YARN divides two major components of JobTracker, resource management and scheduling into two separate modules. The objective is to have a global Resource Manager (RM) to allocate cluster resources to all applications, per application Application Master (AM) that monitors performance of each application. Scheduler is part of Resource Manager. The following sections describe the RM and AM daemons:

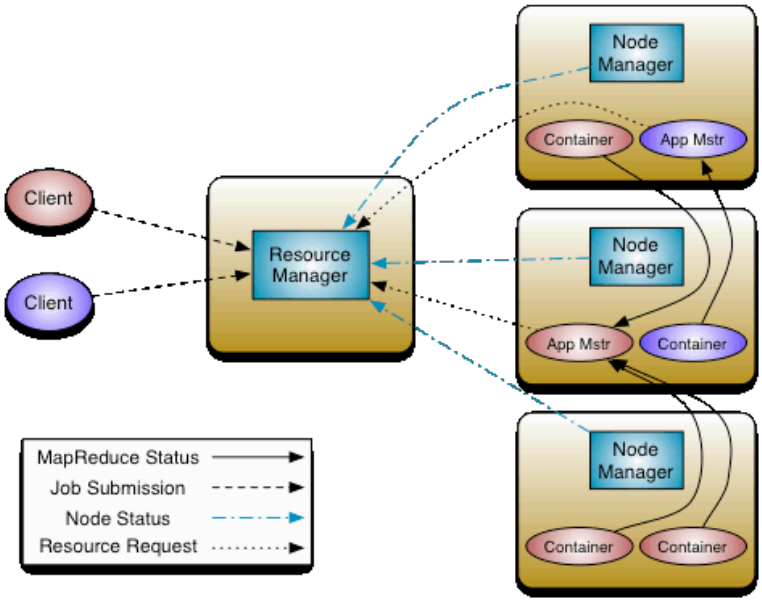


Figure 2.3: YARN architecture with resource manager and node manager [2].

2.5.1. RESOURCEMANAGER

The ResourceManager assigns cluster resources to all the applications in the system. Each application negotiates resource with ResourceManager, which communicates with cluster machines to provide resources to each machine. ResourceManager has three major components, Scheduler, ApplicationManager, and NodeManager.

SCHEDULER

The YARN scheduler performs its scheduling function based the resource requirements of the applications. That is done based on the abstract notion of a resource container, which incorporates resource elements such as CPU and memory. Applications are divided into tasks and in YARN, for every task the scheduler assigns a container on each machine in the cluster. A container is the basic unit of processing capacity in YARN, and is an encapsulation of resources. In the current version, containers only consider memory and CPU cores. YARN allows the administrator to specify the amount of memory and number of CPU cores on each machine in the cluster and uses this information to allocate containers on machines. Containers can be interpreted task slots, but unlike slots, size of a container can be configured (in terms of memory and CPU cores). The total number of containers is dependent on the total memory of a machine, which can be determined by the following equation:

$$\text{Total number of Containers} = \min\left\{\frac{\text{TotalMemory}}{\text{ContainerMemorySize}}, \#\text{Cores}\right\} \tag{2.1}$$

Here, ContainerMemorySize is the configured, fixed amount of memory (typically 1GB or 512MB) and 1 CPU core. The number of containers determine the total parallel tasks executed by a machine. The scheduler has a policy plug-in module, which distribute cluster resources among various applications. Just like MapReduce, YARN uses scheduling plug-in policies, CapacityScheduler and FairScheduler. The core idea of both of these policies is the same as we described earlier.

APPLICATIONMANAGER

The role of ApplicationManager is to accept the submitted applications, create and execute the first container as ApplicationMaster for each application. If an application fails, then it restarts ApplicationMaster for the application. The per-application ApplicationMaster negotiates resource with ResourceManager for the corresponding application and tracks the progress of containers.

NODEMANAGER

NodeManager runs on each cluster machine to execute containers from various applications. In addition to the status of containers, NodeManager also monitors the resource usage such as CPU, disk and memory of those containers. NodeManager sends this monitoring information back to ResourceManager. NodeManager is a replacement of tasktracker in YARN.

2.6. YARN LIMITATIONS

YARN has emerged as one of the strongest tools to solve big data problems. YARN provides the scalability to Hadoop by decoupling scheduling and monitoring (ApplicationManager) units. YARN improves the resource management of Hadoop by implementing dynamically sized containers. However, the business value of the existing YARN architecture can be impacted significantly by a number of shortcomings, including abstraction of heterogeneity, fixed container size, and non-optimal throughput.

ABSTRACTION OF HETEROGENEITY

YARN creates containers on each machine based on the total memory and the number of CPU cores. If there are two machines with different memory size, then they will have different numbers of containers. In other words, unlike Hadoop, YARN takes resource heterogeneity into account, in the case of memory. However, YARN still does not consider heterogeneity in other resource characteristics, such as CPU speed, IO and network bandwidth. For example, let's assume that two machines have the same memory and the same number of cores, but the CPU speeds might differ significantly. In that case, running the same number of containers on both the machines might not be optimal.

FIXED CONTAINER SIZE

YARN creates containers of a fixed size, and the size is configured by administrators while initiating the clusters. In the current implementation of YARN, the container size cannot be changed while running applications. Each container runs one task, therefore, a resource provided by one container can only be used by one container. However, tasks belonging to different applications may have different resource requirements. Creating

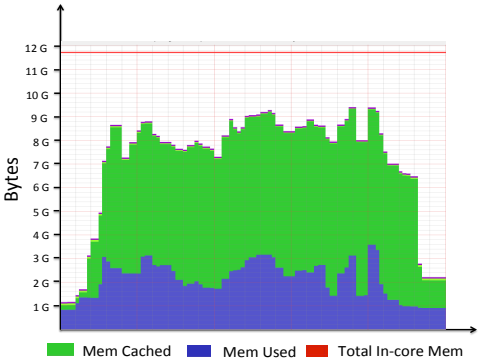


Figure 2.4: Memory usage of a node in the cluster when running 12 Pi and 6 Sort tasks.

containers of fixed size might not utilize resources efficiently. Table 2.1 shows the resource requirements of map tasks from various Hadoop benchmark applications.

Application	CPU Time (Sec)	RAM (MB)	Disk I/O (MB)
Pi (1000 Samples)	10	230	5
Sort (120 GB)	10	280	250
WordCount (60 GB)	30	300	150
RandomWriter (120 GB)	20	140	1024
AggregateWordCount (60 GB)	5	280	120

Table 2.1: Per task resource requirements of Hadoop benchmark applications

In the table, we can see differences of resource requirements. For example, *Pi* uses 230 MB of memory and *Sort* uses 280 MB. Creating containers of fixed size (500 MB or 1 GB), without considering the actual resource requirements might result in wasting resources. To verify this hypothesis, we ran 12 tasks of Pi and 6 tasks of Sort on a node with 12GB of RAM. Note that this is already more than the 12 containers that existing schedulers would allocate.

Figure 2.4 shows the memory usage on that node. As can be seen, and not surprisingly, running these 18 tasks in parallel does not cause a memory bottleneck. During the entire execution of these two applications, the memory usage was sufficiently lower than the total memory on the machine. This exercise proves that creating containers of fixed size can severely under-utilize the available resources.

NON-OPTIMAL THROUGHPUT

In Equation 2.1, the number of containers are dependent on available CPU cores and total memory. Limiting the number of containers by the number of cores avoids CPU bottlenecks, which in turn increases the completion time of tasks. On the flip side, of

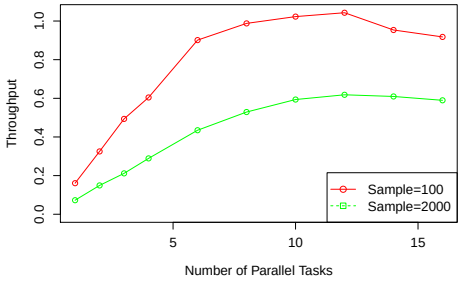


Figure 2.5: Throughput of Pi for different number of parallel tasks.

course, running more tasks on a node, increases the number of completed tasks. This leads to the classical trade-off in multi-processor scheduling of determining the optimal number of concurrent processes to run. This number depends heavily on the relative use of a CPU core by processes.

To demonstrate that it is not always clear a-priori how many concurrently executing tasks maximizes throughput, we ran variations of Pi for varying numbers of containers on nodes with 8 cores. As a proxy for throughput, which we formally define in Chapter 4, we use the total number of map tasks finished per unit of time. Figure 2.5 shows the average values over all map tasks in the application. Maximal throughput is attained when running 12 tasks in parallel. This is neither the number of cores, nor twice the amount—as could be speculated due to hyper-threading. The hyper-threading enables multiple (generally 2) logical cores on a single physical core. It is hence not straightforward to correctly configure the container size in terms of CPU resources.

2.6.1. FAULT TOLERANCE

YARN has better resource management than Hadoop; however, the fault tolerance component of YARN is still same as that of Hadoop. YARN uses the heartbeat protocol to identify faulty machines and, therefore, it struggles detecting soft faults.

2.7. OTHER HADOOP IMPROVEMENTS

In previous sections, we mentioned numerous shortcomings with Hadoop and YARN. Broadly, these challenges can be divided into *Scheduling* and *Monitoring*. In this section, we investigate the contributions that have been made in the field of scheduling and monitoring of Hadoop and other related distributed computing systems.

2.7.1. SCHEDULING

Researchers and engineers have shown great interest [43][44][45] in the space of scheduling for MapReduce-based systems. However, most of the work has been done on the improvement of the *slot*-based architecture of Hadoop. Therefore, some of that work might not be needed or useful to the container-based architecture.

For the slot-based structure, many resource aware schedulers have been proposed, which dynamically try to allocate resources (slots) to tasks. *RAS* [46], is a resource aware scheduler that dynamically allocates Map and Reduce slots to improve the resource utilization. *RAS* uses the application resource requirements to implement this allocation policy. *RAS* generates a performance model of the incoming applications to predict the task completion time under different scenarios. The model is generated by running the applications in various scenarios. This kind of approach is impractical because as soon as a job is submitted, it cannot be halted to generate a model. This will severely impact the productivity of the cluster. Moreover, data locality is not addressed in the work. Unused MapReduce slots are dynamically assigned to active Map or Reduce tasks to improve the performance by dynamically providing fairness [47]. However, this work does not apply to YARN because containers have no notion of Map or Reduce slots. Therefore, a free container can be assigned to either Map or Reduce task. A similar work is proposed to dynamically assign to passive (unused) slots to other tasks [48]. MROrchastration [49] uses resource profile information to detect CPU and memory contention. MROrchastration detects contentions by comparing resource usage of tasks among various nodes and provides them more resources. Google has recently published their cluster management system called, Borg [50], which manages the execution of thousands of Google's applications on tens of thousands of machines. Borg assumes a worst case resource requirement for all the production jobs (MapReduce jobs), and it only accepts a new production job when there are enough resources to run that job. Whenever there are free resources available, Borg runs batch jobs (MapReduce jobs). In the case of resource contention, Borg only kills tasks from batch jobs to reduce the contention. This type of scheduling policy enables Borg to achieve high resource utilization without sacrificing the performance of production jobs. There is no study to analyze the performance of Borg when it has to only run batch jobs. Additionally, Borg does not have a mechanism to learn the resource requirements of jobs.

The *Context Aware Scheduler for Hadoop* (CASH) [51] assigns tasks to the nodes that are most capable of satisfying the tasks' resource requirements. Similar to our approach, CASH learns resource capabilities and resource requirements to enable efficient scheduling. CASH mainly assigns tasks to machines that satisfy the requirements most efficiently. Machines are assumed different by considering their static resource configuration. Unlike our work, they do not differentiate nodes in terms of real time load. Machines with different loads have different performance. Also, CASH derives resource requirements in offline mode. Triple-queue is a dynamic scheduler that classifies jobs based on their CPU and I/O requirements and puts them in different queues. However, these queues work independently in First Come First Serve (FCFS) manner. Resource utilization and throughput both can be improved if tasks from different queues are mixed optimally. Dominant resource fairness (DRF) is a resource allocation policy based on users' resource requirements. DRF allocates resources to users to achieve the maximum fairness.

Studies [52] [53] [54] show the negative impact of resource contention in multicore systems. Therefore, in our approach we learn the performance model of every node to maximize the throughput. The model characterizes the performance of a machine when tasks from multiple applications start exploiting shared resources at the same time. The

performance models [55] of MapReduce tasks are derived to optimize MapReduce workflow. These models have two major limitations. First, they do not consider the real time load on nodes to predict the execution time. Second, they use many low level details that might not be accessible during the execution of applications. Late scheduler [56] predicts task finishing time to take a decision about speculative execution in a heterogeneous Hadoop cluster. This approach uses a heuristic that assumes that the progress rate of tasks is constant in a Hadoop cluster. However, under the resource contention, the assumption is no longer true. Therefore, in order to estimate the task execution time under contention, a more sophisticated model is required. An abstraction of MapReduce is discussed [57] to improve the job completion time; however, it does not take into account the actual resource requirements. [58] investigates scheduling issues in heterogeneous clusters, however, they do not characterize Hadoop applications but rather propose a scheduling strategy that speculatively executes tasks redundantly for tasks that are projected to run longer than any other.

Most of these scheduling approaches depend on the resource requirements of the incoming workload. Therefore, we also review a few approaches in the direction of learning the resource characterization of applications, including workload characterization.

WORKLOAD CHARACTERIZATION

In cloud computing, workload characterization has been studied extensively. For example, [59] describes an approach to workload classification for more efficient scheduling. However, rather than determining the workload characterization explicitly, it merely clusters tasks with similar resource consumptions. In [60], the authors characterize workloads by identifying repeated patterns and finding groups of servers that frequently exhibit correlated workload patterns. [61–63] describe workload characterization for Quality-of-Service (QoS) prediction in web servers. Unlike these, in this paper we characterize the cluster workload directly in terms of resource usage of jobs. We do this passively; that is, without injecting code or adding monitoring to computational nodes. There has been some research using machine learning for Hadoop task characterization. [64] studies an unsupervised learning approach, producing clusters of similarly behaving jobs, but no detailed resource requirements are learned. In terms of heterogeneous Hadoop clusters, there has not been much work in the literature yet.

2.7.2. MONITORING

Kahuna [65] is a diagnosis approach that uses a simple *peer similarity* model to identify faulty machines in a Hadoop cluster. Roughly, the idea underlying its approach is that the same task should take approximately the same amount of time on each node in the cluster. More precisely, the authors build histograms of the time each task of a job takes on each machine. A node is identified as faulty when its histogram deviates from those of the other nodes. The authors show that this approach can detect slowdowns caused by various kinds of issues including CPU hogging, disk I/O hogging, and to a limited degree network package loss. The authors also show that different workloads have different "diagnostic power" in the sense that certain issues are not uncovered by certain jobs. This is consistent with our assumption of different job classes. The authors do not describe whether Kahuna is able to detect what kind of fault may have occurred on a machine.

Kahuna assumes that the cluster is homogeneous, meaning that tasks take roughly the same amount of time across machines. However, unsurprisingly, on heterogeneous clusters, the same task can take significantly longer or shorter depending on which machine is being used. Hence, diagnosis cannot be based on the assumption that the same task should take equally long to execute on every node.

Many root cause analysis techniques use distributed monitoring tools that require active human intervention to locate the fault [66]. Gangilia [14] is a well known distributed monitoring system, which is capable of handling large clusters and grids. X-Trace [67] and Pinpoint [68] are tracing techniques to identify faults in distributed systems. [69] developed a visualization tool to aid humans in debugging performance related issues of Hadoop clusters. The tool uses the log analysis technique SALSA [70], which uses the Hadoop log files and visualizes a state-machine based view of every nodes' behavior. Ganesha [71] is another diagnosis technique for Hadoop, which locates faults in MapReduce systems by exploiting OS level metrics. [72] uses X-Trace to instrument Hadoop systems to investigate Hadoop's behavior under different situations and tune its performance. All these tools, however, provide a visualization of resource metrics. Cluster administrators need to manually analyze these visualizations to locate the problems. None of these tools provide an automatic diagnosis to identify performance problems.

Automated performance diagnosis in service-based cloud infrastructures is also possible via the identification of components (software or hardware) that are involved in a specific query response [73]. The violation of a specified Service Level Agreement (SLA), i.e., expected response time, for one or more queries implies problems in one or more components involved in processing these queries. The methodology is widely accepted for many distributed systems. To a degree, Hadoop is using this approach as well as described above, but since the processing time for MapReduce jobs depend on many factors, including the size of the data, only very crude limits can be used as a cut-off. The determination of a reasonable cut-off is further hindered on heterogeneous clusters, where processing times can vary strongly between machines.

2.8. RESEARCH QUESTIONS

In this thesis, we address the above shortcomings of Hadoop, and propose methodologies to solve them. We formulate these shortcomings as a number of research questions, and in each chapter of this thesis, we answer one of these questions. Following is the list of the research questions that we discuss in the subsequent chapters:

- As we discussed, that the earlier versions of Hadoop assume heterogeneous clusters, and therefore its schedulers do not work efficiently when the clusters are heterogeneous. Schedulers assume that all the cluster machines are same, and distribute workloads uniformly over the clusters. To implement a scheduler that takes into account heterogeneity, the scheduler needs to know what is the capacity of each machine, and what is the resource usage of each workload. In Chapter 3, we address these issues by implementing a scheduler that learns the machine capabilities and workload resource profiles, and use them for making the scheduling decisions.

- YARN is the next version Hadoop that has the ability to work with heterogeneous clusters. However, YARN can only differentiate machines in terms of CPU and memory, it fails to consider disk I/O. Additionally, YARN schedulers do not consider the real time loads on various resource of machines when assign workloads. Such kind of scheduling results in lower cluster throughput. In Chapter 4, we improve the YARN scheduling by proposing a resource aware scheduler that takes CPU, memory, and disk I/O into account. In the chapter we formulate cluster throughput in terms of various parameters, such as machine capabilities, resource usage of workloads and the real time loads on machines. The proposed scheduler tries to maximize the cluster throughput by selecting the values of these parameters that maximize the throughput.

- Earlier versions of Hadoop and YARN both have a fault resilient feature, that can only detect whether machines have are functioning or not. The existing fault resilient component fails to identify if machines are performing slower than the their expected performance, and why they are performing slower. From the scheduling point of view it is critical to know the real time performance state of machines. Therefore, in Chapter 5, we propose a monitoring module for Hadoop clusters that monitors resource specific performance of each machine. The proposed module monitors slowdowns in resources, such CPU and disk I/O.

- In Chapter 6, we combine the scheduling module designed and Chapter 4, and the monitoring module designed in Chapter 5. Goal of this chapter is to design a scheduler that attains an optimal throughput in clusters where machines are failing or their performance is degrading.

3

SCHEDULING IN HETEROGENEOUS ENVIRONMENTS

Hadoop splits applications into various tasks whose execution is distributed over the cluster. Every cluster machine processes a certain number of tasks to finish the applications. Tasks from various applications use different resources of the cluster. Critical to the performance of a Hadoop cluster is the efficient utilization of available resources. The decision on how to use resources; meaning which tasks to execute on which machine, is made by the *scheduler*.

As we discussed in Chapter 1, the scheduler is an important component of CMS design. We use the scheduler to accomplish the first research goal in the thesis, which is the following: The CMS system should automatically assign workloads to machines such that the resource utilization is as optimal as possible and the applications can be executed efficiently. In case of changing applications, the system should autonomously change its workload assignment to keep the performance at the best possible level.

In the case of Hadoop, the scheduler decides how many tasks should be executed in parallel on every machine. This decision is critical for the performance of the overall cluster. If machines are running many tasks in parallel, then they may be completing many tasks together. However, running many tasks on machines can oversubscribe machines, which can slow down all the tasks individually. On the other hand, if machines are running fewer tasks, then tasks may be completed at the faster rate but resources might be underutilized. Therefore, the scheduler's job is to find an optimal assignment, meaning that it assigns as many as tasks possible on every machine without under- or overutilizing resources.

The decisions are made by the scheduler to optimize certain performance metrics, such as application completion time, task completion time, and throughput. Here, throughput can be seen as either the number of applications completed in per unit of time or the number of tasks completed in per unit of time. Resource utilization is another metric.

Existing Hadoop schedulers do not optimize any of these metrics. For example, FairScheduler makes sure that cluster resources are evenly distributed among all the

users and applications, and the Hadoop FIFOScheduler provides all the resources to the application that is first submitted to the cluster. These schedulers fail to optimize the performance of applications while executing them on the cluster. None of these schedulers consider the resource usage of applications while making the scheduling decision.

Moreover, the earlier versions of Hadoop face a challenge; that is, the schedulers assume homogeneous clusters. In practice, clusters may contain a heterogeneous mix of machines. Distributed systems, such as Hadoop, consist of various kinds of software and hardware that work together to execute the workloads. In this thesis, our goal is to optimize the performance of a cluster, and resource heterogeneity has a direct impact on its performance. Therefore, in our work, we only consider hardware heterogeneity. How machines can be different from each other in terms of CPU cores and speeds, memory space and bandwidth, disk I/O speed, and machine aging is described in the following points:

CPU CORES AND SPEEDS

As CPU technology evolves, it is likely that new-generation machines have more cores, and each core is faster compared to previous generation machines. Additionally, different processors can have different micro-architectures that cannot be easily quantified. For instance, Atom processors have an in-order execution pipeline with a slower floating-point unit, while Xeon processors have an out-of-order execution pipeline with a faster floating-point unit. Moreover, a few machines can have special purpose hardware, such as graphics-processing units (GPUs) to accelerate the performance of graphical applications.

MEMORY SPACE AND BANDWIDTH

Machines may have varying amounts of free available physical memory. Performance of applications vary depending on the memory available on machines. Capabilities of memory subsystems might also be heterogeneous. Machines can have many combinations of memory modules, such as DDR2 and DDR3, and memory controllers. The memory bandwidth and access latency can also vary from machine to machine. These different parameters impact the performance of applications and make clusters heterogeneous.

DISK I/O SPEED

Machines can have varying kinds of disks. Some machines have faster disks and some have slower ones. Sometimes, files are read locally, and sometimes, files are read remotely from other machines, which makes disk-access time heterogeneous. Moreover, the adoption of solid-state drives (SSDs) makes the performance of disk the more heterogeneous. Replicas stored on SSDs will be accessed faster than replicas stored HDDs.

MACHINE AGING

Apart from the different hardware capabilities of machines, a cluster of initially identical machines can also become heterogeneous. The performance of machines degrades over time, and the degradation can be non-uniform among machines. Machines contain many electrical components, and these components age over time. Other components,

such as fans and heat sinks get clogged with dust, which reduce the efficiency and performance of machines. Therefore, machines that were identical at the time of purchase might have different performance after a certain amount of time.

To illustrate the argument that even homogeneous machines perform differently under faults, we run an application on a homogeneous Hadoop cluster and measure the task completion times on each machine. To simulate variety among the machines, we run background processes on the machines' CPU cores. If map tasks from one application are operating on similar amounts of data, then on a homogeneous cluster, their task completion times should be similar on each machine. Figure 3.1 illustrates that, even on the homogeneous cluster, the tasks from the same applications take significantly longer or shorter depending on what application is being executed.

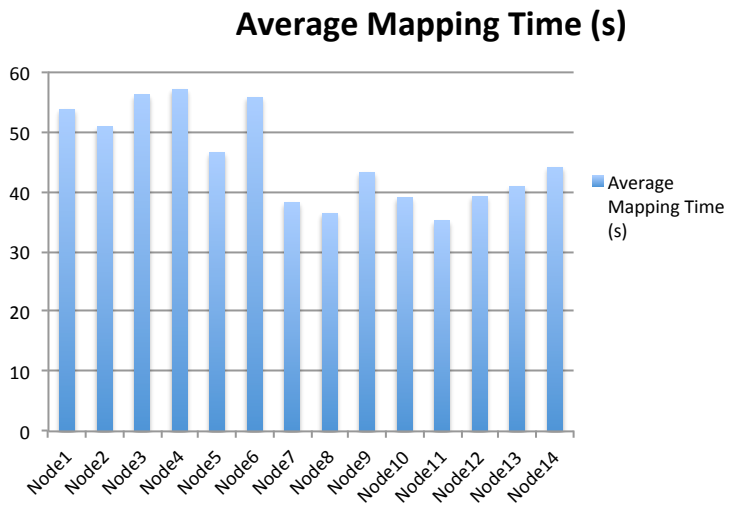


Figure 3.1: Average execution time of mapping tasks of the `WordCount` application on homogeneous machines of the Hadoop cluster at PARC. Although the machines are identical in terms of their hardware specifications, their performances are different due to aging and/or faults.

Whether a cluster is homogeneous or not is important because the creation of slots on each machine depends on the nature of the cluster. The Hadoop scheduler creates a certain number of slots on each machine. A slot is the smallest computation unit on a machine. On each machine, a fixed number of slots are created by the administrator. The maximum number of parallel tasks on a machine is equal to the number of slots on the machine. Each slot executes one map task on the machine. Hadoop assumes a homogeneous cluster and allocates the same number of slots to every machine, regardless of a machine's actual capability. The number of slots is set by the cluster administrator while starting the cluster, and this value cannot be changed when the cluster is in production. Fixing the number of slots might result in over- or undersubscription of the cluster resources.

In order to efficiently utilize the clusters' resources, it is critical to create slots based

on the application resource requirements and machine capabilities. First, for newly submitted applications, it is unknown what their actual runtime resource requirements are, meaning what fraction of the available bandwidth of a resource of a task will utilize on average (for example, the share of CPU time or disk I/O bandwidth). Second, even if these resource requirements can be discovered, it is not immediately obvious which combination of tasks running on a given machine would maximize the performance. For example, it may happen that running a few CPU-intensive tasks with a few disk-intensive tasks may improve the productivity of the cluster [74–76].

To overcome the limitations of existing schedulers in heterogenous environments, we present the *ThroughputScheduler*. The proposed scheduler handles heterogeneity by assigning tasks to machines based on the resources required by tasks and the resource capabilities provided by machines. The proposed scheduler actively exploits the heterogeneity of a cluster to reduce the overall execution time of a collection of concurrently executing applications with varying resource requirements. To enable this type of scheduling, the scheduler requires knowledge of the resource requirements of applications and the resource capabilities of servers, meaning their relative CPU and disk I/O speeds. In our approach, we accomplish this goal without *any* additional input from the user or the cluster administrator. The ThroughputScheduler derives machine capabilities by running "probe" applications on the cluster nodes. These capabilities drift very slowly in practice and can be evaluated at infrequent intervals, such as at cluster set-up time. In contrast, each new application has a priori unknown resource requirements. We therefore derive an online methodology to learn the resource requirements of incoming applications.

The practicality of our solution relies on the structure of applications in Hadoop. These applications are subdivided into *tasks*, often numbering in the thousands, which are executed in parallel on different machines. Mapping tasks belonging to different applications can have very different resource requirements, while mapping tasks belonging to the same application are very similar. This is true for the large majority of practical mapping tasks, as Hadoop divides the data to be processed into evenly-sized blocks. For a given application, we can therefore use online learning to learn a model of its resource requirements from a small number of mapping tasks in an *explore* phase. We can then *exploit* this model to optimize the allocation of the remaining tasks. As we will show, this can result in a significant increase in throughput, and never reduce throughput, compared to Hadoop's baseline schedulers (FIFOScheduler and FairScheduler).

We focus on minimizing the overall time to completion of mapping tasks, which is typically the primary driver of overall application completion time. In order to implement a scheduler that minimizes the task completion, we need an analytical model that predicts the task completion time. In this chapter, we first present our static model, which determines the task completion time in terms of the resource usage of tasks and the resource capabilities of machines. Subsequently, we describe our approach to implement the ThroughputScheduler. In the description of ThroughputScheduler, we introduce our online methodology to learn the task-resource profile and our offline approach to estimate machine capabilities. Finally, we show empirically that ThroughputScheduler can reduce overall application execution time by up to 40 percent on a heterogeneous Hadoop cluster.

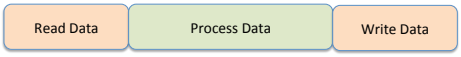


Figure 3.2: Resource-oriented, abstract timeline of a map task.

3.1. STATIC MODEL

A static task model predicts the execution time of a task as a function of its resource usage and the capabilities of the machine it is running on. As we observed before, Map tasks work with one data input and compute continuously until they have finished. Map tasks, therefore, have a simple execution pattern. Before presenting the model, we first introduce the terminology that we will use in the model description. This model is derived by analyzing the process behind the map tasks. Every task exploits a certain amount of machine resources, such as CPU, disk I/O, memory, and network resources. The task resource requirements can be described by a vector, $\theta^i = [\theta^i_1, \theta^i_2, \ldots, \theta^i_N]$, where each component represents the total requirement for an operation type (for example, the number of instructions to process or bytes of I/O to read). Here, θ^i_k denotes how much of resource k is used by task i. Resource k can be either computation time, disk I/O, memory, or network bandwidth. Similarly, the capabilities of the machines are described by a corresponding vector, $\kappa^j = [\kappa^j_1, \kappa^j_2, \ldots, \kappa^j_N]$, whose entires represent rates for processing the respective operation type (for example, FLOPS or I/O per second). Here κ^j_k is the capability of machine j for a resource, k. Resource capabilities are described in terms of their CPU power, disk I/O, memory speed, and network bandwidth.

The task completion time is determined by the ability of the machine to supply the needed resources. The task completion model could be very complicated due the complex process of a realistic task execution. To keep the model simple, we assume that a task exploits each resource exclusively and sequentially. In other words, only one resource at a time is used by a task. For instance, a task might be using either CPU cycles or disk I/O, but it will not use both resources in parallel. Hence, we present a simplified model (Equation 3.1) as a starting point for analysis. In our simplified model, the time taken to finish task i on machine j, $T^{i,j}$, is the sum of the task's resource-specific needs, divided by the machine-specific capabilities in terms of this resource:

$$T^{i,j} = \sum_k \frac{\theta^i_k}{\kappa^j_k} + \Omega^j \tag{3.1}$$

Here, Ω^j is the overhead to start the task on the machine, j. The overhead of a machine can be seen as the time to start JVM to run a task. We have observed that on a given machine, all the applications impose that same amount of overhead. In this case, $\frac{\theta^i_k}{\kappa^j_k}$ determines the time taken by machine j to process resource k requirements of task i. In the previous chapter, we discussed the execution pattern of map tasks, which illustrates how resources are being utilized while executing map tasks. Figure 3.2 provides an abstract view of the map task execution in terms of usage of CPU cycles and disk I/O.

As illustrated in Figure 3.2, we further reduce this model to only two resources: CPU time, denoted by c, and disk I/O, denoted by d. We can express this as a two-dimensional

system with task-resource profile $\theta^i = [\theta^i_c, \theta^i_d]$ and machine capabilities $\kappa^j = [\kappa^j_c, \kappa^j_d]$. Hence, the task duration model will be as follows:

$$T^{i,j} = \frac{\theta^i_c}{\kappa^j_c} + \frac{\theta^i_d}{\kappa^j_d} + \Omega^j \tag{3.2}$$

3.2. THROUGHPUTSCHEDULER

In this section, we describe the design of the ThroughputScheduler [77] [78], which optimizes the assignment of tasks to machines. ThroughputScheduler is designed to optimally schedule tasks on heterogeneous Hadoop clusters. We choose task completion as our performance metric, which ThroughputScheduler tries to minimize by optimally scheduling tasks on machines. In the ThroughputScheduler implementation, we use the static task-completion model to compute the task-completion time. Since ThroughputScheduler extends the Hadoop scheduler, it assumes that each machine has the same number of slots. Every time a slot is available to run a task on a given machine the scheduler selects a task from an application whose resource requirements can be most efficiently satisfied by the machine resources. This results in the faster processing of tasks on machines. To implement this matchmaking, the scheduler uses the static model to predict the task-completion time.

Unfortunately, these requirements and capabilities are not directly observable as there is no simple way of translating machine hardware specifications and task program code into resource parameters. We also assume that we do not have root access on the cluster machines, therefore, resource requirements of tasks can not be directly measured from the kernel. We take a learning-based approach, which starts with an *explore phase*, where parameters of tasks and machines are learned. This is followed by an *exploit phase*, in which the parameters are used to allocate tasks to machines.

3.2.1. EXPLORE

We learn machine resource capabilities and task resource requirements separately. First, we learn machine capabilities offline. In this chapter, we assume that machine capabilities don't change frequently, and therefore, we learn them once. Then using these capabilities, we actively learn the resource requirements for applications online.

LEARNING MACHINE CAPABILITIES

We assume machine capabilities κ^j's and overhead Ω^j remain same for a certain duration time and can be estimated offline for that duration. The machine parameters are estimated by executing probe applications. We determine them by choosing a *unit* map task to define a baseline. The unit map task has an empty map function, and it does not read from or write to HDFS.

The computation (θ_c) and disk I/O task requirements (θ_d) are both zero for the unit map task; therefore, Equation 3.2 allows us to estimate Ω. Multiple executions are averaged to create an accurate point estimate. Note that Ω includes some computation and disk I/O that occur during start up.

One could imagine attempting to isolate the remaining parameters in the same fashion, however, it is difficult to construct an application with zero computation or zero

disk I/O. Instead, we construct applications with two different levels of resource usage defined by a fixed ratio, η.

Let's assume we aim to determine κ_c for a particular machine. First, we run an application $J^1 = \langle \theta_c, \theta_d \rangle$ with a fixed disk requirement θ_d (J^1 might be an application that simply reads an input file and processes the text in the file). For simplicity, in this section, we are ignoring the superscript notation of θ_c and θ_d. We compute the average execution time of this application on each machine. According to our task model, the average mapping time for every machine, j, can be given as follows:

$$T^{1,j} = \frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j \tag{3.3}$$

Next, we run an application J^η, which reads the same input, but the processing is increased by η times compared to J^1. The processing is increased by running the map routine of application J^1 in a *for* loop. Therefore, the resource requirements of J^η can be given as $J^\eta = \langle \eta\theta_c, \theta_d \rangle$. The average mapping time for every machine can be given as follows:

$$T^{\eta,j} = \frac{\eta\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j \tag{3.4}$$

We solve for $\frac{\theta_d}{\kappa_d}$ in Equation 3.3 and 3.4, and obtain the following ratio:

$$\kappa_c^j = \frac{\theta_c(\eta - 1)}{T^{\eta,j} - T^{1,j}} \tag{3.5}$$

This equation gives us κ_c^j in terms of a ratio. To make it absolute, we arbitrarily choose one machine, $j = 1$, as the reference machine. We set $\kappa_c^1 = 1$ and $\kappa_d^1 = 1$ and then solve Equation 3.5 for θ_c. Once we have the task requirements, θ_c in terms of the base units for machine one, we can use this application requirement to solve for the machine capabilities on all the other machines. Similarly, we estimate κ_d.

Normally, in Hadoop, the output of map tasks goes to multiple reducers and may be replicated on several machines. This would have the effect of introducing network communication costs into the system. To avoid this conflation while learning machine capabilities, we set the number of reducers to zero and set the replication factor to one.

We employ this learning approach to determine the resource capabilities of our five-machine cluster at PARC. The cluster contains two classes of machines. Two of the machines (machine1 and machine2) are older machines and have 2 CPU cores. The remaining three machines (machine3, machine4, and machine5) are newer and have 8 CPU cores. Table 3.1 presents the machine capabilities estimated by our approach. Our algorithm correctly discovers that there are two classes of machines, where newer machines are 7.5 times faster in terms of CPU capability and 2.5 times faster in terms of disk capability.

LEARNING APPLICATION RESOURCE REQUIREMENTS

So far in this chapter, we reviewed that in order to implement efficient schedulers, it is important to learn the resource requirements of applications submitted to the cluster.

Machine	κ_c	κ_d	Ω
machine1	1	1	45
machine2	1	1	45
machine3	7.5	2.5	5.3
machine4	7.5	2.5	5.3
machine5	7.8	2.6	4.8

Table 3.1: Recorded machine capabilities and overhead.

Learning resource requirements can be helpful in other ways, too. For instance, if the cluster administrators are planning to buy new machines to extend the cluster infrastructure, they can make their decision based on what kind of resources are mainly used by applications. For example, if it is known that applications are mainly using the I/O resource of machines, then administrators should buy machines with better I/O capabilities. Hadoop clusters typically execute a multitude of concurrent applications in parallel. Task requirements, such as total computational load, disk I/O required and memory footprint, can vary considerably depending on the exact algorithm and data the task encompasses. In practice, it is unreasonable to assume that specific resource requirements are known in advance or expect a programmer to estimate and specify them. Therefore, we need a learning approach to determine the application resource requirements by determining resource requirements of tasks corresponding to the application.

When applications are running on machines, the application resource requirement can be determined by measuring the resource usage of those machines. However, making such measurements requires complete access (root access) over the cluster machines. In many scenarios, obtaining this access is not practical. For example, in case of a public cloud, generally root access is not given to most users. Even in the case of private clouds, mainly in mid- or larged-sized industries or in universities, root access is not provided to employees or students. Therefore, measuring resource usage from the OS kernel can not always be performed. Hence, we need an approach to learn the resource requirements of applications that doesn't require any special privileges over the cluster. Thanks to the resource heterogeneity in the clusters, we can learn the resource requirements of applications without any special privileges.

Every new application may have a varying resource requirement. Therefore, for every new incoming application, the information about the resource usage of the application needs to be learnt. However, using the offline learning approaches might severely affect the cluster productivity. To maintain a certain production level, it is important to execute the applications on the cluster as soon as they are submitted. Therefore, our learning happens during application execution and hence without significant loss in productivity. We execute tasks from various applications on machines that are different in terms of their capabilities, and we learn the resource requirements by observing how the performance of tasks change from one machine to another. As the input, the learner uses the time it takes to execute a certain task on a specific machine in the cluster. Example 3.2.1 describes how we use the resource heterogeneity to learn the application resource requirements.

Example 3.2.1. Let's assume two machines, M_1 and M_2, with different resource capabilities. M_1 has 4 CPU cores with the latest Pentium 4 and one HDD. M_2 is a 4-core ma-

chine, too, but has an old Pentium 1 and has multiple SSDs. In other words, M_1 has faster CPU than M_2, and M_2 has better disk bandwidth than M_1. We assume that there is an application, A, and its resource requirements are unknown. We execute tasks from the application on machines M_1 and M_2. The average task completion time of tasks from the application on M_1 and M_2 is denoted by T_1 and T_2, respectively. An observation where $T_1 < T_2$ suggests that task is finished faster on the machine with faster CPU cores, and better disk I/O does not reduce the task completion time. Therefore, it can be concluded that application tasks use more CPU processing than disk I/O. An observation where $T_1 > T_2$ would conclude vice-versa. This example demonstrates how resource heterogeneity can be used to learn resource requirements of applications.

Example 3.2.1 describes our intuition behind the approach that we control the execution of tasks on machines in order to learn the resource requirements. The example also shows that in order to infer the resource requirements from the task completion time, there must be a relation between them. The static model derived in Equation 3.2 predicts the time a task will take to complete on a specific machine in the cluster, based on its resource requirements and the machine's capabilities. We treat the task completion time as the observation, and implement Bayesian updates to perform the actual learning of application resource requirements.

One of the contributions of this chapter includes the derivation of an *analytical*, closed form for the posterior distribution of the resource requirements in terms of the task completed and machine capabilities. Machine capabilities are treated as point values, and an offline approach is used to determine those values. At the time samples are observed, the posterior distributions are updated. Since our learning approach is online, it is important to learn as fast as possible, such that tasks can be efficiently scheduled. Therefore, while learning we control the generation of observations. In other words, we control the execution of tasks during the learning phase. Every new observation tries to minimize the variance of posterior distributions.

This kind of information gathering exercise is generally known as experimental design. An optimal experiment design will maximize the information gain regarding the resource requirements after each experiment. The objective of each operation is to generate the most informative observations. In the next section, we provide a brief description of our experimental design approach. The experimental design results in a schedule of servers to execute a task on that results in maximum information.

Experimental Design to Learn Resource Requirements: General Setup

Performing exhaustive experiments on a task would require more time than we would save through our optimization process. Instead, we propose an active, learning-based approach to determine the application requirements online during execution. Alternatively, one could view this as an experimental design [79] [80] problem in which prior knowledge about a phenomenon is used to select the next experiment, from a set of possible experiments, in order to maximize expected utility. In our context, the set of possible experiments corresponds to the set of machines a task could be executed on. The outcome of executing task i on machine j with capability κ^j is the measured execution time, $T^{i,j}$. Map tasks from the same job run same function on similar amount of

data, consequently, their execution time on a given machine will also be similar. Therefore, we assume $T^{i,j}$ is a normally distributed random variable with a standard deviation σ^j. Here, we implicitly assume that every machine has different observation variance. There will be a certain utility associated with learning this outcome. The utility of refining task profiles can ultimately be measured by the increase in value (or reduction in cost) of the schedules we can create with the more accurate profiles. This calculation, however, is complex and time consuming. We therefore approximate the utility by using the information gain about the task profile. The task requirements of application i are completely characterized by a set of scalar parameters θ^i.

Our current state of information about requirements for task i is captured by a probability distribution, $P(\theta^i)$. The observation model for the system (likelihood) gives the relationship between observations and the task profile, $p(T^{i,j} \mid \theta^i, \kappa^j, \sigma^j)$. Here, σ^j denotes the measurements noise in machine j. The posterior probability over task requirements represents our updated beliefs and can be calculated using Bayes' theorem:

$$p(\theta^i \mid T^{i,j}, \kappa^j, \sigma^j) = \frac{p(T^{i,j} \mid \theta^i, \kappa^j, \sigma^j) p(\theta^i)}{\int_{\theta^i} p(T^{i,j} \mid \theta^i, \kappa^j, \sigma^j) p(\theta^i) d\theta^i} \tag{3.6}$$

The information gain between the prior distribution over task parameters and the posterior distribution is measured by the Kullback-Leibler (KL) divergence:

$$D_{KL}(p(\theta^i \mid T^{i,j}, \kappa^j) \parallel p(\theta^i)) = \int_{\theta^i} p(\theta^i \mid T^{i,j}, \kappa^j) \cdot \ln \frac{p(\theta^i \mid T^{i,j}, \kappa^j)}{p(\theta^i)} \, d\theta^i$$

To compute the expected information gain before running the actual experiment, we compute the expected value of KL divergence:

$$\int_{T_{i,j}} p(T^{i,j} \mid \kappa^j) D_{KL}(p(\theta^i \mid T^{i,j}, \kappa^j) \parallel p(\theta^i)) dT^{i,j} \tag{3.7}$$

Information theory tells us that the expected KL divergence (information gain) is simply the mutual information between the observation and the task requirements, $I(\theta^i; T^{i,j} \mid \kappa^j)$ ([81]), which can be expressed in terms of the entropy of the prior minus the entropy of the posterior:

$$I(\theta^i; T^{i,j}, \kappa^j) = H(\theta^i) - H(\theta^i \mid T^{ij}, \kappa^j) \tag{3.8}$$

The entropy can be expressed in terms of our model as follows:

$$\begin{aligned} H(\theta^i) &= -\int_{\theta^i} p(\theta^i) \ln p(\theta^i) d\theta^i \\ H(\theta^i \mid T^{ij}) & \\ &= -\int_{T^{i,j}} p(T^{i,j}) \left(\int_{\theta^i} p(\theta^i \mid T^{i,j}, \kappa^j) \ln p(\theta^i \mid T^{i,j}, \kappa^j) d\theta^i \right) dT^{i,j} \end{aligned} \tag{3.9}$$

Optimal Task Inference

Since new applications continually enter the system, we do not know their resource profiles a priori. In this work, we only consider the CPU cycles and disk I/O requirements of applications, which are denoted by θ_c and θ_d. We do not have a functional form to represent the probability distribution, $p(\theta_c, \theta_d)$. The static model shown by Equation 3.2 suggests that θ_c and θ_d are linearly, negatively correlated to each other given an observation. Moreover, map tasks from a given job run same map function on similar amount of data, hence, the resource requirements of these tasks will also be similar. We therefore assume the random variables describing these profiles, θ_c and θ_d, to follow a multivariate Gaussian distribution. The uncertainty about the requirements can therefore be captured by a covariance matrix, $\Sigma_{\theta_c,\theta_d}$:

$$[\theta_c, \theta_d] \sim \mathcal{N}([\mu_{\theta_c}, \mu_{\theta_d}], \Sigma_{\theta_c,\theta_d}) \tag{3.10}$$

As we discussed eariler, T^j, is normally distributed around the value predicted by the task duration model given by Equation 3.2. The uncertainty is given by a standard deviation, σ_j, associated with the measurements in the machine:

$$T^j \sim \mathcal{N}\left(\frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j}, \sigma^j\right) \tag{3.11}$$

We choose Gaussian distribution to represent the random variables because it has a simple mathematical form. Using Gaussian distribution, the closed form of the posterior distribution can be easily derived. The closed form solution enables us to implement these calculations in Hadoop scheduler. In the absence of such closed form solution, a sampling based logic could be implemented, however, using a sampler in the scheduler will severely slow down the decision making process of the scheduler.

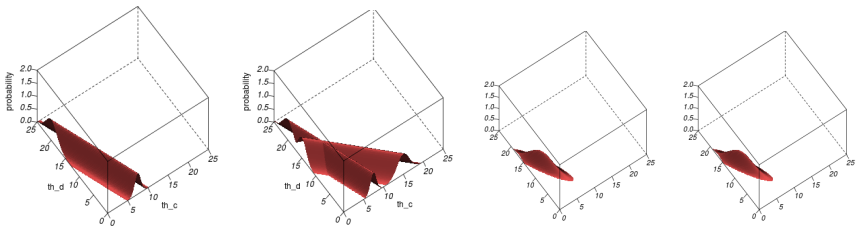
We assume that the machine capabilities, κ_c^j and κ_d^j, are learned using the offline approach described in the previous section. We treat them as point-valued constants. For simplification, we do not use the overhead parameter, Ω^j, in Equation 3.11. Rather, we assume that the learned values of Ω^j have been already subtracted from T^j.

Belief Updating

Given a prior belief about the requirements of a task, $P(\theta)$, and the observation of a task execution time, T^j, of the task on machine j, we can get an updated posterior task profile distribution via Bayes' rule (Equation 3.6). This requires a likelihood function to link observations to parameters. For the bivariate CPU and disk I/O usage example, the likelihood has this form:

$$p(T^j \mid \theta_c, \theta_d, \kappa_c^j, \kappa_d^j) = \frac{1}{\sqrt{2\pi}\sigma^j} \cdot \exp \frac{-\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j}\right)^2}{2{\sigma^j}^2} \tag{3.12}$$

We assume that every machine has a different, but constant observation standard deviation, σ^j, that is learned along with the machine capabilities κ^j. The likelihood is essentially a noisy line in parameter space that describes all of the possible mixtures of CPU



(a) Likelihood (b) Prior and Likelihood (c) Joint posterior (d) Joint posterior

Figure 3.3: Given an observed execution time, the likelihood function defines an uncertain line, representing possible mixtures of computation and disk I/O that would explain the observation. The joint probability of observations is a bivariate Gaussian.

and disk I/O usage profiles of the task that would explain the total observed execution time, T^j. A notional graphical example appears in Figure 3.3(a). Note that the density has been truncated at the boundaries of the positive quadrant as resource requirements cannot be negative.

We can get some insight into the form of the probability distribution function (PDF) of likelihood by considering its contours; that is, the set of points at which it takes the same value. To determine the shape of the contours, we use the following bivariate second order polynomial functional form in terms of θ_c and θ_d:

$$a_{20}{\theta_c}^2 + a_{10}\theta_c + a_{11}\theta_c\theta_d + a_{01}\theta_d + a_{02}{\theta_d}^2 + a_{00} = 0 \tag{3.13}$$

The shape of the above expression can be inferred by computing the determinant, Δ, which is given as the following:

$$\Delta = a_{11}^2 - 4a_{20}a_{02} \tag{3.14}$$

The contour,h, of the likelihood function shown in Equation 3.12 is given by the following:

$$\frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j}\right)^2}{{\sigma^j}^2} = \frac{1}{{\sigma^j}^2}\left(\left(\frac{\theta_c}{\kappa_c^j}\right)^2 + \left(\frac{\theta_d}{\kappa_d^j}\right)^2 - 2T^j\frac{\theta_c}{\kappa_c^j} - 2T^j\frac{\theta_d}{\kappa_d^j} + 2\frac{\theta_c}{\kappa_c^j}\frac{\theta_d}{\kappa_d^j} + {T^j}^2\right) = h \tag{3.15}$$

The determinant, Δ, of the above expression equals

$$\Delta = \frac{1}{{\sigma^j}^4}\left(\left(\frac{2}{\kappa_c^j}\frac{1}{\kappa_d^j}\right)^2 - 4\frac{1}{{\kappa_c^j}^2}\frac{1}{{\kappa_d^j}^2}\right) = 0 \tag{3.16}$$

Since the determinant is zero, the likelihood function is actually a (degenerate) parabola (in fact, a line) rather than an ellipse. Therefore, the likelihood function does not represent a bivariate Gaussian distribution. We refer to this distribution as a Gaussian tube, as it is uniform along the major axis and Gaussian across its minor axis (see Figure 3.3(a)).

Intuitively, this is because after only one observation, there is no information to distinguish which of the resource requirements contributed how much to the time it took to execute the task.

First Update

At the time of a Hadoop application submission, we do not get any internal information about application. This means we do not have any prior belief about the requirements of the application's tasks. We assume an uninformative prior, therefore the posterior distribution is just proportional to the likelihood:

$$p(\theta_c, \theta_d \mid T^j, \kappa) = \frac{1}{\sqrt{2\pi}\sigma^j} \cdot \exp \frac{-\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j}\right)^2}{2{\sigma^j}^2} \tag{3.17}$$

Similar to the likelihood function, the posterior is also the Gaussian tube in parameter space. This implies that there is an infinite number of equally likely explanations for a single observation—this can be thought of as a linear set of equations with two variables but only one equation. We will get a line in space no matter which machine we run the task on, so the results of the first update, by themselves, are not sufficient to guide machine selection.

Second Update

For the second update, we have a prior distribution and likelihood function both in the form of Gaussian tubes. These two are multiplied to obtain the density of the second posterior update. Let the first experiment be on machine j with capability κ^j, and let the observed time be T^j with variance σ^j. Let the second experiment be on machine k with capability κ^k, and let the observed time be T^k with variance σ^k. The resulting posterior distribution is as follows:

$$p(\theta_c, \theta_d \mid T^j, T^k, \kappa^j, \kappa^k) = \frac{1}{z} \cdot \exp - \left[\frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j}\right)^2}{2{\sigma^j}^2} + \frac{\left(T^k - \frac{\theta_c}{\kappa_c^k} - \frac{\theta_d}{\kappa_d^k}\right)^2}{2\,{\sigma^k}^2} \right] \tag{3.18}$$

Here, $z = 2\pi\sigma^j\sigma^k$. We verify that the posterior is bivariate Gaussian by expanding the argument of the exponential in Equation 3.18 and collecting the θ terms (see Equation 3.19).

The determinant, Δ of this contour equals

$$\Delta = -\frac{1}{(\sigma^j\sigma^k)^2}\left(\frac{1}{\kappa_c^j\kappa_d^k} - \frac{1}{\kappa_d^j\kappa_c^k}\right)^2 \tag{3.20}$$

When the determinant is negative, we have an ellipse [82, Chapter 10]. The determinant, can only be non-negative if $\frac{\kappa_c^j}{\kappa_d^j} = \frac{\kappa_c^k}{\kappa_d^k}$, in which case the determinant is zero and the tubes

$$\frac{{\theta_c}^2}{2}\left(\frac{1}{(\kappa_c^j\sigma^j)^2}+\frac{1}{(\kappa_c^k\sigma^k)^2}\right)+\frac{{\theta_d}^2}{2}\left(\frac{1}{(\kappa_d^j\sigma^j)^2}+\frac{1}{(\kappa_d^k\sigma^k)^2}\right)-\theta_c\left(\frac{T^j}{\kappa_c^j\sigma{j}^2}+\frac{T^k}{\kappa_c^k\sigma{k}^2}\right) \tag{3.19}$$
$$-\theta_d\left(\frac{T^j}{\kappa_d^j\sigma{j}^2}+\frac{T^k}{\kappa_d^k\sigma{k}^2}\right)+\theta_c\theta_d\left(\frac{1}{\kappa_c^j\kappa_d^j\sigma{j}^2}+\frac{1}{\kappa_c^k\kappa_d^k\sigma{k}^2}\right)+\frac{{T^j}^2}{2\sigma{j}^2}+\frac{{T^k}^2}{2\sigma{k}^2} \quad = \quad h$$

Figure 3.4: Level sets of the posterior distribution.

3

are parallel. Hence, as long as we choose machines with different capability ratios $\frac{\kappa_c}{\kappa_d}$, the intersection is an ellipse, and the distribution is bivariate normal. In higher dimensions, we can check the dot product of the normals of the planes representing possible solutions to test for parallelism.

We can recover the mean (μ_{θ_c,θ_d}) and covariance matrix ($\Sigma_{\theta_c,\theta_d}$) of the bivariate Gaussian distribution by identifying the origin and the rotation of the ellipse, as well as the length of its major and minor axes. We can determine the coefficients of terms involving θ_c and θ_d in Equation 3.19 in terms of the simple coefficients a_{nm} shown in Equation 3.13.

A well known decomposition relation allows us to then read off the mean and the inverse covariance matrix [83] as follows:

$$\begin{bmatrix} \mu_{\theta_c} \\ \mu_{\theta_d} \end{bmatrix} = \begin{bmatrix} \frac{a_{11}a_{01}-2a_{02}a_{10}}{4a_{20}a_{02}-a_{11}^2} \\ \frac{a_{11}a_{10}-2a_{20}a_{01}}{4a_{20}a_{02}-a_{11}^2} \end{bmatrix} \tag{3.21}$$

$$\Sigma^{-1}_{\theta_c\theta_d} = \begin{bmatrix} a_{20} & \frac{1}{2}a_{11} \\ \frac{1}{2}a_{11} & a_{02} \end{bmatrix} \tag{3.22}$$

Notice that the terms involving observation T^j in the expanded form in Equation 3.19, which were replaced by coefficients a_{10}, a_{01} and a_{00} in Equation 3.13, do not appear in the covariance matrix. The covariance matrix is therefore independent of the observations T^j and T^k. Therefore, the covariance matrix can be derived for a posterior distribution without actually running tasks on the machines. This implies that we can precompute the experimental schedule offline.

Third and Further Updates

For the third update, the prior will be a bivariate normal and the likelihood function a Gaussian tube. The general form of a bivariate normal distribution is given the following:

$$p(\theta_c,\theta_d) = \frac{1}{2\pi\sqrt{|\,\Sigma_\theta\,|}}\exp\left(-\frac{(\theta-\mu_\theta)^T\Sigma_\theta^{-1}(\theta-\mu_\theta)}{2}\right)$$

θ, μ_θ and Σ_θ can be calculated from Equation 3.21 and Equation 3.22:

$$\theta = \begin{bmatrix}\theta_c\\ \theta_d\end{bmatrix} \quad , \quad \mu_\theta = \begin{bmatrix}\mu_{\theta_c}\\ \mu_{\theta_d}\end{bmatrix} \quad \text{and} \quad \Sigma_\theta = \begin{bmatrix}\sigma^2_{\theta_c} & \rho\sigma_{\theta_c}\sigma_{\theta_d}\\ \rho\sigma_{\theta_c}\sigma_{\theta_d} & \sigma^2_{\theta_d}\end{bmatrix}$$

Here, μ_θ is the vector of means μ_{θ_c} and μ_{θ_d}. Σ_θ is the general form of covariance matrix, which includes individual variances σ_{θ_c} and σ_{θ_d}, and the correlation coefficient ρ. We will use the same likelihood distribution as we used in Equation 3.12. Given the prior and likelihood, the posterior can be derived as shown in Equation 3.23.

To determine the family of the above distribution, we substitute the mean vector and covariance matrix and then derive the expanded form of the distribution, as shown in Equation 3.24. As described in the previous section, one can show that the determinant of the result is negative for machines with non-identical capability ratios, and that the resulting family is a (elliptical) bivariate normal in that case. We have therefore reached a state where we have a closed form that can be updated repeatedly.

$$p(\theta_c,\theta_d \mid T^j,\kappa^j) = \frac{1}{(2\pi)^{\frac{3}{2}} \mid \Sigma_\theta \mid^{\frac{1}{2}} \sigma^j} \exp\left[\left(-\frac{1}{2}(\theta-\mu_\theta)^T\Sigma_\theta^{-1}(\theta-\mu_\theta)\right) - \frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_c}{\kappa_c^j}\right)^2}{2\sigma^{j2}}\right] \quad (3.23)$$

Figure 3.5: General form of the joint distribution of Elliptical Gaussian with Gaussian Tube.

Mutual Information Computation

We have shown that we can compute a posterior distribution over task parameters given observations of the execution time of a task on a machine. We must now consider the machine on which to execute the task to maximize information gain. In the previous section, we stated that the expected information gain for the next experiment can be computed as the mutual information between the distribution of the task requirements, θ, and the observed time sample T^j. The mutual information is simply the difference in entropy, which is determined by $H(\theta) - H(\theta \mid T^j,\kappa^j)$ (cf. Equation 3.8).

The entropy of a multivariate Gaussian distribution is proportional to the determinant of the covariance matrix ([81]). Therefore, the entropy of the prior is

$$H(\theta) = \frac{\ln 2\pi e \mid \Sigma_\theta \mid}{2} \quad (3.25)$$

where $\mid \Sigma_\theta \mid$ is the covariance of the prior. The entropy of the posterior, $p(\theta \mid T^j)$, is

$$-\int_\theta p(\theta \mid T^j,\kappa^j) \ln p(\theta \mid T^j,\kappa^j) d\theta = \frac{\ln 2\pi e \mid \Sigma_{\theta|T^j,\kappa^j} \mid}{2} \quad (3.26)$$

Applying Equation 3.26 to the definition of conditional entropy (Equation 3.9) gives us

$$H(\theta \mid T^j,\kappa^j) = \int_{T^j} p(T^j)\left(\frac{\ln 2\pi e \mid \Sigma_{\theta|T^j,\kappa^j} \mid}{2}\right) dT^j \quad (3.27)$$

where $\left|\Sigma_{\theta|T^j,\kappa^j}\right|$ denotes the determinant of the covariance matrix of the posterior distribution.

$$\frac{1}{2(1-\rho^2)}\left[\frac{(\theta_c-\mu_{\theta_c})^2}{\sigma^2_{\theta_c}}+\frac{(\theta_d-\mu_{\theta_d})^2}{\sigma^2_{\theta_d}}-\frac{2\rho(\theta_c-\mu_{\theta_c})(\theta_d-\mu_{\theta_d})}{\sigma_{\theta_c}\sigma_{\theta_d}}\right]+\frac{\left(T^j-\frac{\theta_c}{\kappa^j_c}-\frac{\theta_d}{\kappa^j_d}\right)^2}{2\sigma^{j2}} = h$$

$$\frac{{\theta_c}^2}{2}\left(\frac{1}{(\kappa^j_c\sigma^j)^2}+\frac{1}{\sigma^2_{\theta_c}(1-\rho^2)}\right)+\frac{{\theta_d}^2}{2}\left(\frac{1}{(\kappa^j_d\sigma^j)^2}+\frac{1}{\sigma^2_{\theta_d}(1-\rho^2)}\right)$$

$$-\theta_c\left(\frac{T^j}{\kappa^j_c\sigma^{j2}}+\frac{\mu_{\theta_c}}{\sigma^2_{\theta_c}(1-\rho^2)}-\frac{\rho\mu_{\theta_d}}{\sigma_{\theta_c}\sigma_{\theta_d}(1-\rho^2)}\right)$$

$$-\theta_d\left(\frac{T^j}{\kappa^j_d\sigma^{j2}}+\frac{\mu_{\theta_d}}{\sigma^2_{\theta_d}(1-\rho^2)}-\frac{\rho\mu_{\theta_c}}{\sigma_{\theta_c}\sigma_{\theta_d}(1-\rho^2)}\right)+\theta_c\theta_d\left(\frac{1}{\kappa^j_c\kappa^j_d\sigma^{j2}}-\frac{\rho}{\sigma_{\theta_c}\sigma_{\theta_d}(1-\rho^2)}\right)$$

$$+\frac{{T^j}^2}{2\sigma^{j2}}+\frac{{\mu_{\theta_c}}^2}{2\sigma^2_{\theta_c}(1-\rho^2)}+\frac{{\mu_{\theta_d}}^2}{2\sigma^2_{\theta_d}(1-\rho^2)}-\frac{\rho\mu_{\theta_d}\mu_{\theta_c}}{\sigma_{\theta_c}\sigma_{\theta_d}(1-\rho^2)}=h \tag{3.24}$$

Figure 3.6: Contour expression of the exponent term of the joint distribution shown in Equation 3.23

In Equation 3.22, we showed that the covariance matrix of the posterior distribution is independent of observation T^j and is therefore constant with respect to the integration in Equation 3.27. Then, all that remains in the integral is the prior, which integrates to one. We are left with the covariance term:

$$H(\theta \mid T^j,\kappa^j)=\frac{1}{2}\ln 2\pi e\left|\Sigma_{\theta|T^j,\kappa^j}\right| \tag{3.28}$$

First Experiment

We assume there is no prior knowledge about the task profile. The variance and entropy of the prior distribution are therefore unbounded, as expressed in $H(\theta)=|\Sigma_\theta|=\infty$. The posterior distribution after the first update has a linear tubular form. The overall variance and entropy are therefore still undefined: $\left|\Sigma_{\theta|T^j,\kappa^j}\right| = H(\theta \mid T^j,\kappa^j)=\infty$. The information gain determined by $H(\theta)-H(\theta \mid T^j,\kappa^j)$, is therefore undefined. Therefore, the first observation, by itself, does not independently give us information about which machine to run the task on first.

Second Experiment

After the second update, assuming that we have experimented on two machines, j and k, whose ratios of capabilities $(\frac{\kappa^j_c}{\kappa^j_d},\frac{\kappa^k_c}{\kappa^k_d})$ are distinct, the updated posterior follows a non-degenerate bivariate Gaussian distribution. At the beginning of the second experiment, the prior is still undefined and therefore, again, $H(\theta)=\infty$. Hence, the information gain

determined by $H(\theta) - H(\theta \mid T^j, T^k, \kappa^j, \kappa^k)$ will be maximized by minimizing the conditional entropy as provided by $H(\theta \mid T^j, T^k, \kappa^j, \kappa^k)$. As shown in Equation 3.28, the entropy is driven by the determinant of the covariance matrix. This determinant can be derived using the inverse covariance matrix $|\Sigma|^{-1}$:

$$|\Sigma| = \frac{1}{|\Sigma|^{-1}} \tag{3.29}$$

$$|\Sigma|^{-1} = -\frac{\Delta}{4} \tag{3.30}$$

From Equation 3.20, we can substitute in the expression for Δ:

$$\begin{aligned} H(\theta|T^j, T^k, \kappa^j, \kappa^k) &= \ln 2\pi e(-\frac{4}{\Delta}) \\ &= -\ln\left[\frac{1}{2\pi e}\frac{1}{(\sigma^j\sigma^k)^2}\underbrace{\left(\frac{1}{\kappa_c^j\kappa_d^k} - \frac{1}{\kappa_d^j\kappa_c^k}\right)^2}_{\text{squared term}}\right] \end{aligned} \tag{3.31}$$

Therefore, we can minimize the posterior entropy, $H(\theta \mid T^j)$, by simply maximizing the squared term in the discriminant in Equation 3.31. This term will be maximized when the difference of fractions is maximized. The difference is maximized when one is large and the other is small. Note that the denominator of each fraction consists of one term from each machine, but with different dimensions. The pattern that maximizes this is to maximize the difference between machines on each dimension (for example, one machine with a fast CPU and slow disk I/O, and another machine with a slow CPU and fast disk I/O).

Third and Subsequent Experiments

We can express the total information gain of a series of experiments $1, 2, \ldots, m$ as follows:

$$\begin{aligned} & I(\theta; T^{j^1}, T^{j^2}, \ldots, T^{j^m}, \kappa) \\ &= \sum_{k=0}^{m-1} H(\theta|T^{j^{1:k}}, \kappa^{1:k}) - H(\theta|T^{j^{1:k+1}}, \kappa^{1:k+1}) \\ &= H(\theta) - H(\theta|T^{j^{1:m}}, \kappa^{1:m}) \end{aligned} \tag{3.32}$$

As shown, the series telescopes as internal terms cancel. To maximize the information gain, we need to minimize the second term, which is the entropy of the posterior distribution conditioned on all experiments.

We can evaluate this using a method similar to the previous section. We obtain the entropy indirectly from the discriminant. The general form of the discriminant for three or more observations has a regular form. The entropy of the posterior conditioned on all

Application	μ_{θ_c}	μ_{θ_d}	$\lvert\Sigma_{\theta_c\theta_d}\rvert$	# of Tasks
Pi	24.00	6.30	0.0038	109
Random Writer	27.26	234.62	0.0061	28
Grep	15.82	8.10	0.0038	90
WordCount (1.5 GB)	43.50	22.50	0.00614	31
WordCount (15 GB)	138.05	206.40	0.00615	32
J_{IO}	5.60	96.46	0.0063	30

Table 3.2: Application resource profile measurements with variance and number of tasks executed

experiments has one term for each possible pairing of machines:

$$H(\theta|T^{j^{1:m}},\kappa^{1:m}) = \ln 2\pi e(-\frac{4}{\Delta}) \tag{3.33}$$
$$= -\ln\left[\frac{1}{2\pi e}\sum_{j\neq k}\frac{1}{(\sigma^j\sigma^k)^2}\left(\frac{1}{\kappa_c^j\kappa_d^k}-\frac{1}{\kappa_d^j\kappa_c^k}\right)^2\right]$$

This is going to be minimized when each of the squares is maximized. As in the previous case, the squares will be maximized when the machines' parameters in each pair are most dissimilar. This result does not depend on the observations, so we can plan the sequence of experiments before execution time.

We sample tasks until we get a determinant for the covariance matrix $|\Sigma_{\theta_c,\theta_d}| < 0.007$. We empirically observed that for all applications, we get stable values of μ_{θ_c} and μ_{θ_d}, when determinant of covariance matrix is less than 0.007. Table 3.2 summarizes resource requirements learned by the online inference mechanism for some of the Hadoop benchmark applications [84]. When we compare the 'Pi' application, which calculates digits of Pi, to RandomWriter, which writes bulk data, we see that the algorithm correctly recovers the fact that Pi is compute intensive (large μ_{θ_c}), whereas RandomWrite is disk intensive (large μ_{θ_d}). Other Hadoop applications show intermediate resource profiles as expected. The J_{IO} application will be described further in the experimental section. The '# of Tasks' column gives the number of tasks executed to reach the desired confidence.

3.2.2. EXPLOIT

Once the resource profile of an application is learned to sufficient accuracy, we switch from explore to exploit. The native Hadoop scheduler sorts task-to-machine pairs according to whether they are local (data for the task is available on the machine), on the same rack, or remote. We introduce our routine based on our task requirements estimation called "SelectBestapplication" to break ties within each of these tiers as shown in Algorithm 3.2.1. If we have two local applications, we would run the one most compati-

ble with the machine first.

Algorithm 3.2.1: THROUGHPUTSCHEDULER(Cluster, Request)

for each machine N ∈ Cluster

do
- applicationsWithLocalTasks ← N.GETAPPLICATIONSLOCAL(Request)
- applicationsWithRackTasks ← N.GETAPPLICATIONSRACK(Request)
- applicationsWithOffSwitchTasks ← N.GETAPPLICATIONSOFFSWITCH(Request)
- **if** Localapplications ≠ *NULL*
  - **then** J ← SELECTBESTAPPLICATION(Localapplications, N); ASSIGNTASKFORAPPLICATION(N, J)
- **else if** Rackapplications ≠ *NULL*
  - **then** J ← SELECTBESTAPPLICATION(Rackapplications, N); ASSIGNTASKFORAPPLICATION(N, J)
  - **else** J ← SELECTBESTAPPLICATION(OffSwitchapplications, N); ASSIGNTASKFORAPPLICATION(N, J)

Algorithm 3.2.2: SELECTBESTAPPLICATION(*machineN*, *Listofapplications*)

return $(\text{argmin}_{J \in \text{ListOfapplications}} \frac{\text{norm}(\theta_c^J)}{\text{norm}(\kappa_c^N)} + \frac{\text{norm}(\theta_d^J)}{\text{norm}(\kappa_d^N)})$

SelectBestapplication, shown in Algorithm 3.2.2, selects application J that minimizes a *score* for task completion on machine N. However, rather than using absolute values of θ_c, θ_d, κ_c and κ_d, we use the normalized value of these parameters to define the score. While absolute values represent expected time of completion, which can be measured in seconds, application selection based on these numbers would always favor short tasks over longer ones and fast machines over slower ones. For example, consider machines 1 and 3 in Table 3.1. In this case, machine3 is almost 7.5 times faster than machine1 in terms of CPU processing, but only 2.5 times faster in terms of disk I/O. Hence, intuitively, disk intensive applications are better scheduled on machine1, since the relatively higher CPU performance of machine3 is better used for CPU intensive applications (if there are any). To account for this relativity of optimal resource matching, we normalize both applications and machines to make their total requirements and capabilities sum to one for each resource x (here, $x \in \{c, d\}$):

$$\text{norm}(\theta_x^i) = \frac{\mu_{\theta_x^i}}{\sum_k \mu_{\theta_k^i}} \qquad \text{norm}(\kappa_x^j) = \frac{\kappa_x^j}{\sum_k \kappa_x^k}$$

3.3. EXPERIMENTAL RESULTS

To evaluate the performance of *ThroughputScheduler*, we conducted experiments on a five machine Hadoop cluster at PARC (see Table 3.1). The cluster contains two classes to machines, newer and older. The newer machines are 7.5 times faster in terms of CPU processing and 2.5 times faster in terms of disk I/O processing. All the experiments were repeated multiple times, and in each run same results were obtained. Therefore, we select results from one run to describe in the following subsections.

3.3.1. EVALUATION ON HETEROGENEOUS APPLICATIONS

We evaluate the performance of ThroguhputScheduler on applications with various resource requirements. Since the Hadoop benchmarks do not contain highly I/O intensive applications (cf. Table 3.2), we constructed our own I/O intensive Map-Reduce application, J_{IO}. J_{IO} reads 1.5 GB from HDFS, and writes files totaling 15 GB back to HDFS. This resembles the resource requirements of many expand-translate-load (ETL) applications used in big data applications to pre-process data using Map-Reduce and to write HBase, MongoDB, or other disk-backed databases. We learn J_{IO}'s resource profile using the application learner described in the Explore section. The learned resource requirement of J_{IO} is listed in Table 3.2. To evaluate ThroughputScheduler on drastically heterogeneous application profiles, we run J_{IO} along with the Hadoop benchmark Pi, which is a CPU intensive application. We compare the performance of ThroughputScheduler with FIFOScheduler and FairScheduler—for a single user, CapacityScheduler is no different from FIFOScheduler.

3.3.2. APPLICATION COMPLETION TIME

We first compare the performance of the proposed scheduler in terms of overall application completion time. In case of multiple applications, the overall application completion time is defined as the completion time of the application finishing last. In this experiment, we study the effect of heterogeneity between application resource requirements, which we can quantify as the ratio of disk I/O to CPU requirement of an application: $h = \frac{\theta_d}{\theta_c}$. In order to vary this quantity, we vary the I/O load of J_{IO} further by varying the replication factor of the cluster—the higher the replication factor, the higher the I/O load of an application. As we increase the replication factor, the job J_{IO} will write more number of data blocks on each machine. This impacts disk I/O intensive applications more than other applications.

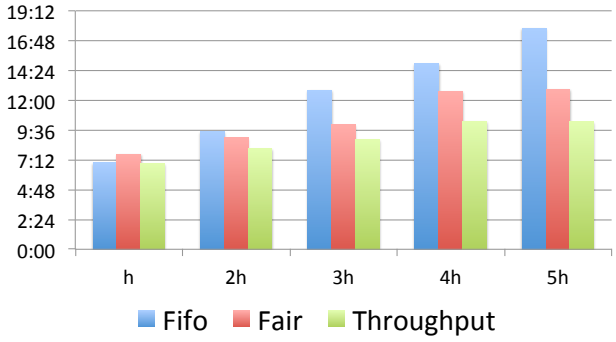


Figure 3.7: Overall application completion time in minutes (Y axis) on heterogeneous machines at PARC for different relative values of $h = \frac{\theta_d}{\theta_c}$. Disk load θ_d is increased by increasing the replication number.

These results show that ThroughputScheduler performs better than FIFOScheduler and FairScheduler in all cases. The relative performance increase of our scheduler increases as the heterogeneity of the two applications increase, as simulated by an in-

creased replication factor, which is up to 40 percent compared to FIFOScheduler, and up to 20 percent compared to FairScheduler. Note that both the FairScheduler and ThroughputScheduler benefit from higher replication as they can better take advantage of data locality. The improvements of ThroughputScheduler beyond FairScheduler are purely due to our improved matching of applications to computational resources.

application	FIFO	Fair	Throughput
Pi	9 sec	9 sec	6 sec
J_{IO}	2 min 15 sec	2 min	2 min 10 sec

Table 3.3: Comparison of average mapping time.

To better understand the source of this speed-up, we considered the average mapping time for each application (*throughput*). Table 3.3 summarizes these results and provides the explanation for the speed-up. As shown in the table, our scheduler improves the throughput of Pi by 33 percent, while maintaining the throughput of J_{IO} compared to the other schedulers. Since Pi has very many mapping tasks, these savings pay off for the overall time to completion.

3.3.3. PERFORMANCE ON BENCHMARK APPLICATIONS

To estimate the performance of ThroughputScheduler on realistic workloads, we also experimented with the existing Hadoop example applications. We ran the application combinations of concurrent applications as shown in Table 3.4.

$Comb_1$	Grep (15 GB) + Pi (1500 samples)
$Comb_2$	WordCount (15 GB) + Pi (1500 samples)
$Comb_3$	WordCount (15 GB) + Grep (15 GB)

Table 3.4: Application combination.

The performance comparison in terms of application completion time is presented in Figure 3.8.

For these workloads, ThroughputScheduler performs better than either of the other two in all cases. For $Comb_2$, the application completion time is reduced by 30 percent compared to FIFOScheduler. For $Comb_3$, all three schedulers perform similarly because both applications are CPU intensive (cf. Table 3.2).

Application Combination	FIFO	Fair	Throughput
$Comb_1$	210s	224s	214s
$Comb_2$	440s	319s	310s
$Comb_3$	225s	262s	214s

Table 3.5: Completion time of application combinations on a homogeneous cluster.

3.3.4. PERFORMANCE ON HOMOGENEOUS CLUSTER

We ran additional experiments on a set of homogeneous cluster machines to ensure such a setup would not cause ThroughputScheduler to produce inferior performance. Results

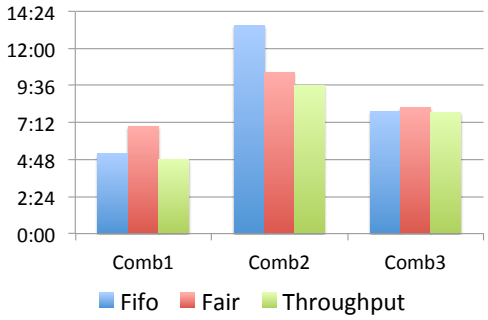


Figure 3.8: Application completion time in minutes (Y axis) of combinations of Hadoop example applications.

shown in Table 3.5 empirically demonstrate the effectiveness of ThroughputScheduler on homogeneous clusters. The improved performance suggests that if application resource requirements and machine capabilities are taken into account, then there is room for the performance improvement on homogeneous clusters as well.

3.4. SUMMARY

ThroughputScheduler represents a unique method of scheduling jobs on heterogeneous Hadoop clusters using active learning. The framework learns both server capabilities and job task parameters autonomously. The resulting model can be used to optimize allocation of tasks to servers, and thereby, reduce overall execution time (and power consumption). Initial results confirm that ThroughputScheduler performs better than the default Hadoop schedulers for heterogeneous clusters and does not negatively impact performance, even on homogeneous clusters.

In this chapter, apart from the optimal scheduling, we also provide an online approach to learn the application resource requirements by exploring heterogeneous clusters. Our online learning approach constructs a schedule of task assignments to machines that maximizes information gain about the resource requirements of tasks. This schedule depends only on the capability parameters of the available machines and can hence be computed offline before any experiments or tasks are executed.

We have assumed that machine performance is stationary. If we allow machine performance to be an estimated parameter, we could contemplate diagnosing suboptimal performance issues using a similar model. This is particularly relevant for Hadoop deployments on cloud infrastructure and other uses of virtual machines where performance can be less predictable, and many aspects of the system state are hidden and need to be diagnosed or inferred from observations.

We have shown that the Bayesian experimental design paradigm leads to an elegant closed-form solution to estimate the expected information gain of executing a task on a machine, and that the optimal experimental schedule can be precomputed offline. In addition, we have shown that there is a simple method for updating the posterior distribution over task parameters given the observation. Together, these results constitute a thought-provoking first step towards learning the resource requirements of applications

by exploiting the natural heterogeneity in the cluster. Our approach allows us to learn resource requirements without the extra privileges (root access) over the cluster.

Despite the performance improvement by ThroughputScheduler, there are still a few limitations with the scheduling policy. In order to learn the resource requirements, ThroughputScheduler needs a few heterogeneous machines in the cluster. If a cluster is homogeneous, then the resource requirements can not be learned, and therefore, scheduling policy can not be implemented. Additionally, ThroughputScheduler does not take into account the real-time load on various resources of machines while making the scheduling decision.

4

DARA: DYNAMICALLY ADAPTING, RESOURCE AWARE SCHEDULER

In the last chapter, we introduced the ThroughputScheduler for heterogeneous clusters, which takes application resource requirements and machine capabilities into account. Compared to existing Hadoop schedulers, ThroughputScheduler is a major improvement to Hadoop scheduling for two reasons. First, it relaxes the non-realistic homogeneity assumption, and second, it's a resource aware scheduler: The ThroughputScheduler assigns tasks to those machines that can most efficiently satisfy the resource requirements of those tasks.

We have experimentally shown the benefits of using ThroughputScheduler, it can still be improved upon. For example, ThroughputScheduler does not consider the real-time loads on the machine resources while assigning tasks to machines. Rather, it implicitly assumes that all the machines have identical loads on all the resources. Loads, however, have a direct impact on the performance of applications. For instance, CPU cores will be very busy if they are already running many CPU intensive tasks. In that case, running additional CPU tasks will further load the CPU cores, which might slow down all the CPU intensive tasks on that machine. Therefore, it is critical to consider real-time loads on machines' resources when scheduling tasks.

The Hadoop community is aware of limitations with the existing Hadoop schedulers. The latest version of Hadoop, YARN, takes into account the resource capabilities of machines, and therefore can deal with heterogeneous clusters up to a certain point. YARN pretends that every task requires a fixed amount of memory (generally, 1 GB) and a fixed number of CPU cores (generally, 1 CPU core). Tasks are allocated to every machine until the total memory is full or all the CPU cores are busy. As we discussed in Chapter 2, YARN uses the following equation to determine the number of parallel tasks on a machine, also denoted by the number of containers:

$$\text{Total number of Containers} = \min\left\{\frac{\text{TotalMemory}}{\text{ContainerMemorySize}}, \#\text{Cores}\right\} \tag{4.1}$$

Here, ContainerMemorySize denotes the memory assigned to a container. Each container executes exactly one task. This scheduling however might lead to an inefficient utilization of available resources. For instance, let's assume machine *M* has 16 CPU cores and 2 GB of memory. Let's suppose the scheduler assumes that every container uses 1 GB of memory; in other words, the container memory size is 1 GB. Hence, machine *M* will run 2 tasks in parallel. If CPU intensive applications are submitted, then the CPU cores of machine *M* will be underutilized. Therefore, to efficiently utilize the available resources, it is critical to consider the actual resource usage of tasks. In summary, we observe the following shortcomings of YARN:

4

- Fixed container size in terms of CPU core and memory.
- Similar to ThroughputScheduler, YARN also fails to take into account the real-time load when assigning tasks to machines.

To overcome the shortcomings of the scheduling policies mentioned in this chapter, we introduce the Dynamically Adapting, Resource Aware (DARA) Scheduler for Hadoop. DARA improves the throughput on both homogenous and heterogeneous clusters. DARA extends YARN so it takes into account actual machine resource loads, machine capabilities and the resource requirements of tasks while assigning tasks on machines. To generate the scheduling policies, DARA uses throughput as the objective function, where throughput is defined in terms of number of tasks completed per unit time.

Just like the ThroughputScheduler, DARA also uses a performance model to generate scheduling policies, which is known as a *dynamic model*. The dynamic model predicts task completion time in terms of task resource requirements, machine capabilities and loads on the resources of each machine. We learn the parameters of the dynamic task completion model automatically on data obtained from offline experiments. The parameters are related to resource capabilities of machines. The dynamic model allows us to solve the problem of determining the optimal combinations of tasks that maximizes the throughput of the cluster. Figure 4.1 notionally describes the intuition behind DARA's efficiency gains.

In Figure 4.1, the axes denote the available resources: CPU (X axis) and memory (Y axis). The two rectangles along the axes (blue and orange) are the representation of two machines (nodes) with different machine capabilities. As shown with the rectangles, machine1 (Node1) has more memory compared to machine2 (Node2), and machine2 has more processing power than machine1. Let's assume there are tasks from two applications, A and B, to be scheduled on machine1 and machine2. Tasks from application A use more memory and less processing power than tasks from application B. An arrow in the plot represents the resource usage of a task. The job of the scheduler is to keep the arrows in the rectangle until they reach either the memory or CPU edge of the rectangle. In other words, the scheduler's goal is to keep adding tasks to a machine, as long as memory or CPU are not over subscribed. The existing Hadoop schedulers (CapacityScheduler and FairScheduler) do not take into account the actual resource requirements of tasks and

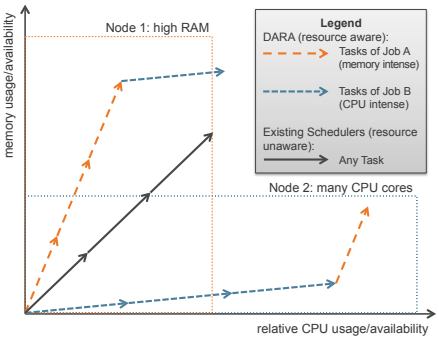


Figure 4.1: Existing schedulers are unaware of resource requirements of tasks and hence cannot use this information to make scheduling decisions. DARA reaches better task-to-node allocations by mixing tasks with varying resource requirements, which leads to better utilization of resources.

instead allocate the same amount of resources (for example, 1 CPU core and 1 GB memory). This kind of allocation is represented by black arrows in Figure 4.1. Only three tasks could be added to machine1 because all the available CPU resources are used, however, there was still free memory left. Similarly, the default Hadoop scheduler can only fit two tasks to machine2.

On the other hand, DARA takes into account both the resource usage and available resources to fit the maximum number of tasks. DARA adds four tasks to both machines. It adds three memory intensive tasks (orange arrow) to machine1, and to keep overall memory load less than the total memory, it adds a CPU intensive task (blue arrow). Similarly, DARA can fit four tasks to machine2. In summary, by assigning tasks on machines based on the resource requirements and load, DARA can run three more tasks in parallel than existing Hadoop schedulers can without oversubscribing the resources.

Intuitively, we expect performance gains from DARA due to a better utilization of concurrent resources (for example, disk I/O plus CPU, multiple CPU cores) while carefully avoiding over-subscription. We monitor the loads on various resources on machines and execute tasks that are suitable for those particular loads. For instance, mixing CPU-intensive tasks with disk-intensive tasks will achieve better throughput than putting CPU-intensive tasks with other CPU-intensive tasks. Running CPU-intensive tasks will increase CPU load on a machine and, therefore, to get better performance we should add disk-intensive task that will not impose additional CPU load on a machine [74–76]. Likewise, DARA can account for heterogeneity in available cluster resources, for example, by automatically assigning disk-intense tasks to nodes with fast solid-state disks, if available and possible, without reducing benefits of data-locality.

As we will show, DARA improves throughput over existing Hadoop schedulers up to 50 percent compared to CapacityScheduler and 55 percent compared to FairScheduler on a homogeneous cluster. DARA also speeds up the workload execution as much as 1.5 times compared to CapacityScheduler and 1.55 compared to FairScheduler.

In the next section, we define throughput and then introduce our model for task completion time. Following that, we formulate DARA's scheduling policy as the result of an optimization problem that maximizes throughput. Finally, we present our empiri-

cal results and conclusions.

4.1. PRELIMINARIES

In this section, we provide a general framework that we use to implement the DARA scheduler, and we also define the performance metric (throughput). We first define the terminologies used in the framework.

4.1.1. CLUSTER

A cluster $\mathscr{C}$ is a set of n machines, $\mathbf{M} = \{M^1, M^2, \ldots, M^n\}$, that running a set of l applications, $\mathbf{A} = \{A^1, A^2, \ldots, A^l\}$. The goal of a scheduler is to optimize some objective function. In our work, we choose throughput as our objective function. The description of throughput is given in the following section.

4.1.2. THROUGHPUT

Throughput can be understood as the rate at which a cluster is finishing the given work. Naively, throughput could be defined as:

$$\text{Throughput} = \frac{\text{Total work completed in time window } T^w}{T^w} \tag{4.2}$$

In an Hadoop environment, total work is defined as all the applications that need to be processed by the cluster. An application is divided into number of tasks. The total work can, therefore, be seen as the number of completed tasks processed by the cluster in a certain time window T^w. The total work, however, cannot be simply calculated by adding the number of tasks of each application completed by the cluster in the time window T^w. The following example illustrates that counting all tasks from all application does not give the actual work done.

Example 4.1.1. Depending on the resource requirements of applications, tasks from various applications can have different task completion times. For instance, let's assume two tasks, $task^1$ and $task^2$, which belong to applications A^1 and A^2. A^1 has twice as much CPU demand as A^2 and A^1 reads and writes three time more bytes than A^2. Under similar conditions, on a given machine, task completion time of $task^1$ will always be higher than $task^2$ because $task^1$ uses more CPU time than $task^2$. In simple words, we can say that $task^1$ is bigger than $task^2$. As a result, for a given time window, a cluster will complete more smaller tasks than bigger tasks. Completing more smaller tasks doesn't necessarily imply that the total work is also finishing at the higher rate because the total work needs to be done is the sum of both bigger and smaller tasks. Smaller tasks will be finished faster, but it might happen that only a small fraction of the total work is completed by the cluster in a certain interval.

The naive definition of the total work done does not take the *size* of a task into account; hence, maximizing total work favors smaller applications (or tasks) over larger ones. Instead, we define the total work in terms of the normalized number of completed tasks, where normalization is done based on a notion of *size* of tasks. We define the size λ_j of a map task belonging to application A^j as the time to finish the task on a reference

machine where no other tasks are running in the background. If there are l applications running on a n machine cluster, then the following seems to be an adequate measure of how much work is getting done by this task:

$$\text{Total work completed in time window } T^w = \sum_{i=1}^{n} \sum_{j=1}^{l} \lambda_j y_{ij} \tag{4.3}$$

where y_{ij} is the total number of tasks of application A^j completed on machine M^i in the time window T^w. In our definition, the total work done by the cluster is determined by the sum of all the work finished by each machine in T^w. The work done by a machine can be estimated by adding the effective number of tasks of each application that is running in parallel in T^w.

To estimate the throughput, we also need an appropriate method to define T^w. The definition has a direct impact on the scheduling policies. A scheduling policy might not be efficient if T^w is not defined properly. The following examples illustrate the importance of defining T^w appropriately.

Example 4.1.2. Consider three applications, A^1, A^2 and A^3 that are running concurrently. All tasks of these applications have identical size and same resource requirements. Each application has a different number of tasks: A^1 has 1000 tasks, A^2 has 5000 tasks, and A^3 has 20000 tasks. At any time, a new application can be submitted to the cluster. One possible way to define T^w is by interpreting it as the time to complete the longest application. In our case, A^3 is the longest application, therefore, T^w would be equal to the completion time of A^3. However, it might very well happen that before the completion of A^3, a new application, A^4 is submitted which has a greater number of tasks than A^3. In that case, T^w would be equal to the completion time of A^4. △

The above example highlights that there is no unique way to define the completion time of the longest application. Similarly, due the possibility of the application submission at any random time, there is no unique way to define the completion time of the smallest application, too. Therefore, we define T^w as the duration between submission and completion of applications. We start the clock as soon as a new application is submitted or an existing one finishes. We stop the clock and measure the time as soon as any other new application is submitted or completed. For a given time window, T^w, the throughput of the cluster can be defined as

$$\tau(\mathscr{C}, \mathbf{Y}) = \sum_{i=1}^{n} \sum_{j=1}^{l} \frac{\lambda_j \cdot y_{ij}}{T^w} \tag{4.4}$$

Here, $\tau(\mathscr{C}, \mathbf{Y})$ denotes the throughput of the cluster $\mathscr{C}$ and $\mathbf{Y}$ is represented by the following matrix:

$$\mathbf{Y} = \left[y_{ij} \right]_{\substack{i \in [1,n] \\ j \in [1,l]}} \tag{4.5}$$

Equation 4.4 shows that the total throughput of the cluster is determined by adding the throughput of each machine. The value y_{ij} depends on how many parallel tasks of application A^j are running on machine M^i. The number of parallel tasks from an application on a machine is determined by the number containers assigned to the application

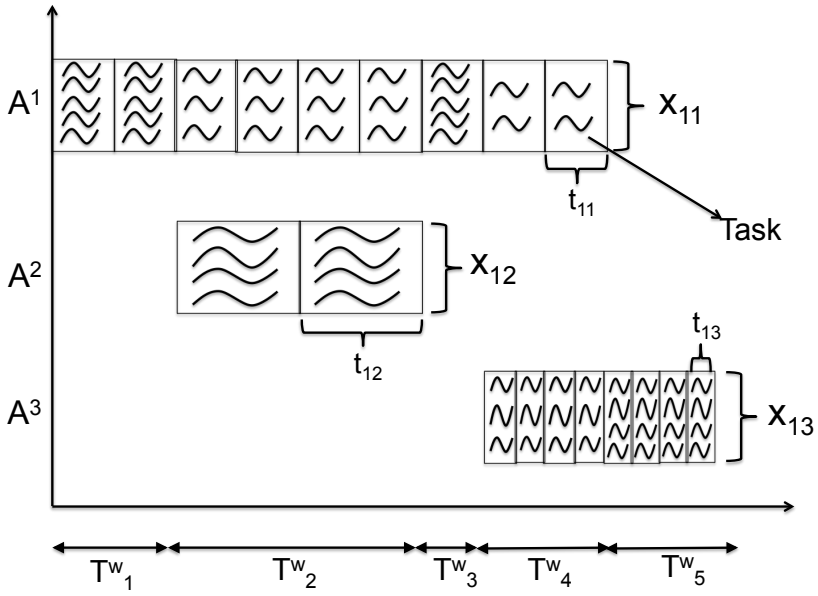


Figure 4.2: The timeline of the execution of three applications on a machine M^1. The X axis denotes the time and the Y axis denotes the application. The timeline shows various T^ws. x_{1j} denotes the number of parallel tasks of application A^j on machine M^1. The average task completion of those parallel tasks is represented by t_{1j}.

on that machine. Let's assume x_{ij} denotes the number of containers of application A^j to be run on machine M^i. In this chapter, we improve the cluster throughput by determining the optimal values for x_{ij}. We denote ρ_{ij} as how many times x_{ij} parallel tasks from application A^j are executed concurrently on machine M^i to run total y_{ij} tasks. The value of ρ_{ij} can be estimated by

$$\rho_{ij} = \frac{y_{ij}}{x_{ij}} \tag{4.6}$$

For example, if 4 containers are assigned to an application on certain machine, then to execute 100 tasks of that application, approximately 25 times that many tasks will be executed in parallel.

If x_{ij} containers of application A^j are running on machine M^i, we represent the average task completion as t_{ij}. The time windows T^w can be estimated by Equation 4.7:

$$T^w = \rho_{ij} t_{ij} \tag{4.7}$$

To illustrate the formulation presented so far, we present the following example, which is graphically depicted in Figure 4.2.

Example 4.1.3. Let's assume three applications, A^1, A^2, and A^3, that are submitted to a cluster at different points in time. Figure 4.2 shows the timeline of these applications

on machine M^1. Other cluster machines will also have analogous timelines. We assume these applications are identical in terms of their size but have varying resource requirements. Application A^1 is submitted first and 5 tasks of A^1 are executed in parallel on M^1 ($x_{11} = 5$). The average task completion time of these parallel tasks is denoted by t_{11}. Each task is represented by a thread. Later, applications A^2 and A^3 are submitted to the cluster. During the entire lifetime of these three applications, we observe 5 time windows, $T_1^w, \ldots, T_5^w$. For each time window, the throughput can be estimated by dividing the number of completed tasks by the time window. For instance, in T_2^w, the machine processes 12 tasks of A^1 ($y_{11} = 12$) and 8 tasks of A^2 ($y_{12} = 8$). In total, 20 tasks are completed by M^1 in the time window T_2^w. During the time window T_2^w, three tasks of application A^1 ($x_{11} = 3$) and 4 tasks of A^2 ($x_{12} = 4$) are executed in parallel. Therefore, from Equation 4.6 we can derive that $\rho_{11} = 4$ and $\rho_{12} = 2$. Hence, by using Equation 4.7, T_2^w can be estimated in terms of t_{11} and t_{12} as follows:

$$T_2^w = 4t_{11} = 2t_{12} \tag{4.8}$$

△

Combining Equations 4.4, 4.6 and 4.7 gives:

$$\bar{\tau}(\mathscr{C}, \mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{l} \frac{\lambda_j \cdot x_{ij}}{t_{ij}} \tag{4.9}$$

Here, $\mathbf{X}$ is the container assignment matrix

$$\mathbf{X} = \left[x_{ij}\right]_{\substack{i \in [1,n] \\ j \in [1,l]}} \tag{4.10}$$

Equation 4.9 implies that if there are more parallel tasks executed on a machine, then the machine will complete more tasks simultaneously, but each task might take longer to complete because the load on the machine would be higher. Therefore, we need a scheduling policy that can maintain a balance between parallelism and loads on a machine such that the throughput is maximal.

Hence, our scheduling goal can be concisely, albeit abstractly, stated as computing:

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmax}}\, \bar{\tau}(\mathscr{C}, \mathbf{X}) \tag{4.11}$$

The definition of throughput is provided in terms of hindsight measurements. Hence, in order to maximize throughput by making intelligent scheduling decisions, we need to be able to *predict* throughput. We hence need to develop a model of throughput that depends on the decisions made in scheduling. In the next section, we will show how we can automatically learn the parameters of the model to predict the individual task completion times t_{ij}. We will then describe how we can maximize the predicted throughput while respecting data locality. As mentioned in the Chapter 2, data locality is a crucial feature of Hadoop. Our scheduler needs to optimize only within the flexibility provided by the redundant storage of data on multiple nodes, and optimize only in terms of the assignment to those machines.

4.2. LOAD DYNAMIC MODEL

In this section, we describe the steps to learn the task completion time model in terms of task resource requirements, machine capabilities and the current load on machines. Unlike the static model, the dynamic model is not purely a process-based model. We learn the parameters of model from the data. In the previous chapter, we have already introduced the concept of the resource requirements and machine capabilities. In this chapter, we also introduce the load on machines. Load on a machine depends on the characteristics of tasks being executed by the machine. In order to derive a realistic model of task completion time, we use a notion of load on existing resources.

We denote the *load* on machine M^i by a tuple $L^i = \langle L^i_1, L^i_2, \ldots, L^i_N \rangle$, where L^i_k is the load on resource k of machine M^i. In our work, we consider loads on two resources of a machine: L^i_c, or CPU load; L^i_d, or disk load. We will now present an approach to estimate these loads given the set of tasks running on M^i, and some measurable resource usages of these tasks. Before explaining our modeling approach, we first describe the data set and our methodology to generate the data set.

4.2.1. GATHERING DATA

In the load dynamic modeling method, our goal is to learn parameters of a model that predicts task completion time in terms task of resource usage, machine capabilities and the loads on machines. We use data driven approach to learn the parameters. The data set consists of task completion time of various tasks executed on machines under various load. The load on a machine is determined by how many tasks are running in parallel on the machine and what are their resource usage. Different kinds of tasks impose different kinds of load. To generate different kinds of loads, we run various combinations of tasks in parallel on the machine. These parallel tasks may have similar or different resource utilization. To generate the data, we run tasks from various Hadoop benchmark applications. We selected the benchmark applications because they cover the spectrum of applications that we use in our Hadoop cluster. For instance, we execute a certain number of Pi tasks in parallel to generate one kind of load and then we execute certain number of Pi and $WordCount$ tasks in parallel to generate another kind of load. We run various combinations of tasks to cover a broader spectrum of load values. We use applications Pi, $WordCount$, $Sort$, $Pi+WordCount$, $Pi+Sort$ and $Pi+WordCount$ to generate the data. Here, $Pi+WordCount$ means that tasks from Pi and $WordCount$ are being executed in parallel.

INSIGHTS FROM DATA

To gain a deeper insight in the data, we first present the characteristics and configurations of applications executed on machines. Table 4.1 shows the list of applications we run to generate the data, along with the number of task configurations we use to generate various loads.

We collect average map task completion time for every application in each configuration. Each configuration imposes a certain amount of CPU, IO and memory load on a machine. To analyze the impact of loads on machines, we present and discuss the average task completion times of a few applications for different load settings.

Application	CPU Time (Sec)	RAM (MB)	Disk I/O (MB)	Task Configurations
Pi	10	230	5	2 to 16
Sort	5	280	500	2 to 16
WordCount	30	300	140	2 to 16
Pi + WordCount	6/30	230/300	5/140	2 to 16
Pi + Sort	6/5	230/280	5/500	2 to 16
WordCount + Sort	30/5	300/280	140/500	2 to 16

Table 4.1: Per task resource requirements of Hadoop benchmark applications. In the cases of $a+b$ applications, the resource requirements are presented in the a, b order.

The data plotted in Figure 4.3 shows the impact of the number of parallel tasks on the map task completion. For each configuration, a certain number of tasks from the application Pi are executed to impose a certain load. The plots show that as the number of parallel tasks increases. In the beginning, the task completion time remains nearly unchanged, and subsequently, the task completion time increases nonlinearly. Table 4.1 shows that Pi tasks perform a very small amounts of I/O activity, hence, Pi tasks mainly exploit machine CPU. Therefore, it is assumed that each Pi task adds extra load on CPU cores of a machine. For fewer numbers of parallel tasks, multiple cores share the CPU load and process tasks in parallel; there, we observe that initial load doesn't increase the task completion significantly. Later on, for higher numbers of parallel tasks, each core is overloaded with work, which causes a non-linear increment of task completion time.

In this example, we only showed the impact of CPU load on task completion time. In our next example, we discuss the impact of CPU and I/O load on task completion. To demonstrate the impact of multiple resources, we present the task completion time data of applications Pi and $Sort$.

The data in Figure 4.4 show that as the number of Pi and $Sort$ tasks increase, the task completion time for both applications increases slowly in beginning and subsequently it increases faster. We also observe adding more Pi tasks does not have significant impact on $Sort$ task completion time; similarly, adding $Sort$ tasks does not increase task completion time Pi task significantly. Pi is a CPU intensive application and $Sort$ is an I/O intensive. Therefore, Pi tasks do not have significant impact on the performance of $Sort$ tasks and vice versa.

These examples demonstrate that as we add more tasks on a machine, the completion of those tasks increases because the load on those machine increases. Each task adds a certain amount of load on each resource of the machine. For example, Pi tasks impose more load on a CPU then other resources, and $Sort$ task puts more pressure on a disk. In order to determine the impact of load on task completion time, we first need a formulation to estimate the load. The next section describe our approach to estimate load on various resources of a machine.

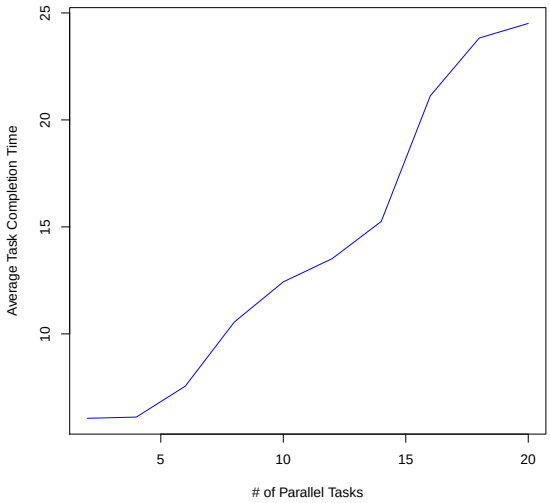
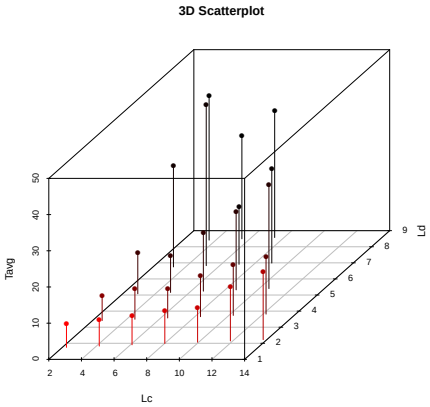
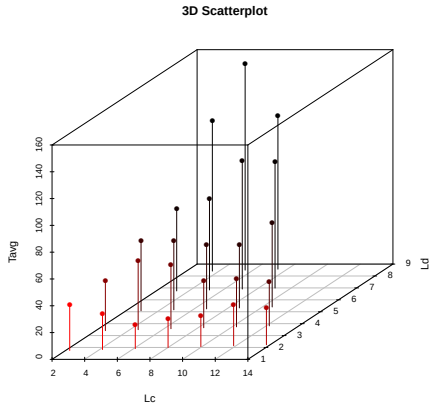


Figure 4.3: Average Map task completion time under various configuration



(a) Task Completion time for Pi application



(b) Task Completion time for $Sort$ application

Figure 4.4: Task completion data for Pi and $Sort$ applications. The X axis (L_c) represents the CPU load and the Y axis (L_d) represents the disk load.

4.2.2. DETERMINING LOAD

Every task imposes a certain load on the CPU, memory and disk of a machine, depending on the CPU, memory and disk requirements of the task. As we discussed earlier, map tasks first read files, then process the read content based on their map function, and finally write data back to the local disk. Reading from and writing to the filesystem constitutes the total disk load imposed by the task. In this work, we are only considering CPU, disk and memory loads on a machine. Estimating memory load is straightforward. A certain amount of memory is used for the entire execution of a task. If there are multiple tasks running in the parallel on the machine, then the total memory load can be determined by adding the memory used by all those tasks.

On the other hand, computing CPU and disk I/O load is not as straightforward as estimating memory load because, during the execution, we assume that a task is either exploiting CPU or disk. Therefore, it is not easy to determine how much load is imposed by the task on CPU and disk I/O. To estimate the CPU and disk I/O load, we take into account the CPU time and the bytes used by the task. Both CPU time and bytes used are measured from the Linux kernel of the machine where tasks are executed. To measure resource usage from Linux kernel, root access privileges over the cluster machines might be needed. The root access is not necessarily granted for every cluster. In the case when the root access is not granted, the ThroughputScheduler described in the previous chapter can be used, otherwise DARA scheduler can be used. The CPU time determines the time spent by the CPU to finish the task. We assume that the remaining time is spent by the task to perform the disk activity. Therefore, we assume that the load imposed on CPU cores of a machine is proportional to the CPU time of the task divided by the overall task completion time, and the amount of disk I/O load is proportional to the number of bytes read and written divided by completion time.

MODEL BASED APPROACH

For a task of application A^j, we can use the following entities as a proxy for CPU and disk loads:

$$R_c^j = \frac{c^j}{c^j + d^j} \qquad R_d^j = \frac{d^j}{c^j + d^j}$$

Here, c^j and d^j are the CPU time and total bytes written and read by a task. For example, applications Pi and $Sort$ have the same CPU seconds. However, $Sort$ has much more disk activity than Pi (cf. Table 4.1) and as a result, its R_d value is much higher than its R_c value. This reflects the fact that $Sort$ is a disk I/O intense application and Pi is a CPU intense application.

If l applications are running concurrently on a machine M^i, then the total loads on M^i can be computed as:

$$L_c^i = \sum_{j=1}^{j=l} R_c^j x_{ij} \qquad L_d^i = \sum_{j=1}^{j=l} R_d^j x_{ij}$$

Here, x_{ij} denotes the number of containers assigned to application A^j to machine M^i.

4.2.3. LEARNING THE MODEL

In this work, our intuition is to derive a task completion model on a given machine, under a given task assignment policy, and knowing only the overall resource requirements of the task (CPU time and disk I/O in bytes). The task assignment policy will determine the CPU and I/O load on every machine because the policy decides that how many tasks from each application will run on every machine.

We use the following function as our inductive bias for the model to be learned:

$$t_{ij} = \lambda^j + c^j \alpha_c^i (L_c^i)^2 + d^j \alpha_d^i (L_d^i)^2 \tag{4.12}$$

Here, t_{ij} denotes the predicted task completion time. It must be noted that we do not include memory load in the model, rather, we assume that task assignment policy always makes sure that the used memory is always less than total memory. In case of over-subscrption of the memory, we assume that the performance of machines become undeterministic and therefore, cannot be formulated in a form of an analytical equation.

In the model, c^j and d^j are the CPU time and bytes written and read by tasks of A^j. α_c^i and α_d^i capture the cpu and disk I/O capabilities of machine M^i. L_c^i and L_d^i represent the current load on these resources, as a result of other tasks running on the machine. c^j and d^j, are the only two values that need to be measured at runtime, since they are application specific, and we cannot know all applications ever submitted to the cluster ahead of time. L_c^i and L_d^i, can be computed at runtime based on the scheduler's knowledge of which tasks have already been assigned and are currently running on the machine. Hence, the parameters that need to be learned are: α_c^i, α_d^i. We search for values of parameters that minimize the Root Mean Squared Error (RMSE) between the values predicated by the model and the actual observed values.

The model in Equation 4.12 uses the size of the application, λ^i, as the intercept, which determines the execution time of a task under minimal load. The model reflects the fact that as the loads, L_c^i and L_d^i increase, tasks start experiencing a slowdown. Empirically, we observed a super-linear increment in task completion with respect to L_c^i and L_d^i, therefore, we model the increase in completion time using $(L_c^i)^2$ and $(L_d^i)^2$.

Figure 4.5 gives a flavor of the training data and fitted model. It shows the values of the average completion time, T_{avg}, for map tasks of Pi running on a node with eight CPU cores and one disk for various loads. These loads, L_c and L_d, were a result of different combinations of Pi and $Sort$ applications running on the node. Red dots in Figure 4.5 shows the empirical T_{avg} value of Pi on the node. These values increase more rapidly with L_c compared to L_d. This is because Pi is a CPU intensive application. Also note that the plot reflects the eight CPU cores, as this is roughly where the times start increasing in the L_c dimension.

We evaluate the accuracy of the model in terms of Root Mean Squared Error (RMSE). We measure 30 percent RMSE for the model we learn in this chapter. We intend to further improve the accuracy of the model and therefore, we develop a data driven model which further improves the accuracy. We do not consider a linear functional form for the model because a linear form would imply that the task completion time linearly increases with the load. This kind of linear behavior suggests that the throughput remains unchanged for varying load values. Moreover, we could also use a more complex functional form to improve the accuracy of the model, but in that case, we could not get the analytical

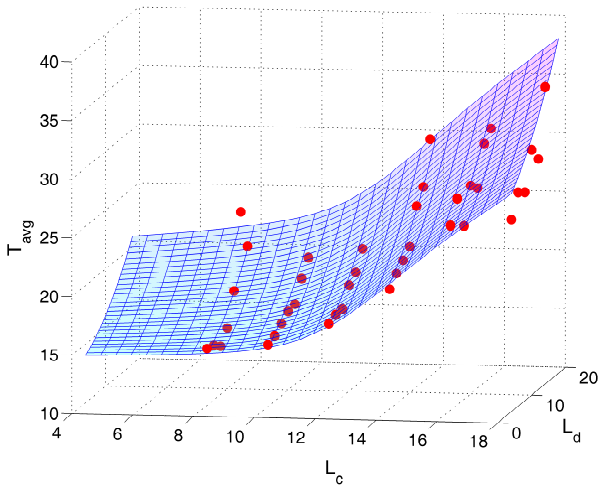


Figure 4.5: Training data and fitted model of task completion time for Pi on a node with eight CPU cores. The x- and y-axes, L_c and L_d, represent CPU and disk I/O load, respectively. The z-axis shows the average task completion time. Red dots show the empirical T_{avg} value, while the blue surface represents the trained model.

form for the scheduling policy. In the absence of the analytical form, a sampler-based approached can be used to generate scheduling policy. Implement the scheduler using a sampler could severely impact the performance of the scheduler.

4.3. THROUGHPUT MAXIMIZATION

To implement the scheduling policy, we use Equation 4.12 to predicted the individual task completion time t_{ij}. Equarion 4.12 can be expressed as the following function ρ, which takes θ^j, π^i, L^i, and λ^j as inputs:

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmax}} \sum_i \sum_j \frac{\lambda^j x_{ij}}{\rho(\theta^j, \pi^i, L^i, \lambda^j)} \tag{4.13}$$

where $\theta^j = (c^j, d^j)$.

As we discussed earlier, the total cluster throughput is a summation of the throughput of all the machines in a cluster. To simplify matters, we assume that the throughput of a machine is independent from all throughput of all other machines. This is not entirely accurate given a set of tasks to be allocated, as allocating a task to one machine means that it will not be allocated to a different machine. But the effect of this dependence seems minor and vanishes for large numbers of tasks being allocated. With this simplification, Equation 4.13 can be rewritten as:

$$\mathbf{X}^* = \sum_i \underset{\mathbf{X}}{\operatorname{argmax}} \sum_j \frac{\lambda^j x_{ij}}{\rho(\theta^i, \pi^j, L^j, \lambda^j)} \tag{4.14}$$

In Equation 4.14, we break down the maximization of cluster throughput into the maximization of throughput of individual machines. This allows the scheduler to maximize each machines throughput separately by allocating optimal numbers of containers to applications with different relative resource loads. Hence, the overall optimization problem can stated as:

$$\mathbf{X}^* = \sum_i \mathbf{x}_i^* \tag{4.15}$$

where $\mathbf{x}_i^*$ denotes the vector of containers to allocate for each application on machine M^i in order to locally maximize throughput on this machine:

$$\mathbf{x}_i^* = \underset{\mathbf{x}_i}{\operatorname{argmax}} \sum_i \frac{\lambda^j x_{ij}}{\rho(\theta^j, \pi^i, L^i, \lambda^j)} \tag{4.16}$$

Given the trained model derived in Equation 4.12, we can use Equation 4.12 directly to formulate the scheduling used by DARA, and which we describe programmatically in the next section.

4.3.1. IMPLEMENTATION

We implemented the DARA scheduler in Hadoop version 2 (YARN) by extending the existing FairScheduler. The implementation can be succinctly described by its three main functions, ADDAPPLICATION, REMOVEAPPLICATION and DARASCHEDULE. Resource capabilities (π^M) are estimated for every machine M in the cluster before starting the cluster for production.

In the production phase, when an application, A^{New}, arrives, one of the map tasks from the application is executed on a node, and its overall resource requirements, θ^{New}, (CPU time and total number of bytes written and read) are obtained by the NodeManager from the operating system on the node. This information is sent to the ResourceManager where it is received by DARA. The list of all applications **A** and their resource requirements, Θ, as well as the current assignment, **X**, are known from the records kept by the scheduler.

The function, COMPUTEOPTIMALASSIGNMENT, maximizes the expression shown in Equation 4.16 by finding the optimal number of containers for each application for a given machine M. This function takes as input the resource requirements of applications, Θ, and the capabilities, π^M, of the machine.

To avoid the over-subscription of memory, this method only explores the space of combinations of tasks where the sum of the peak memory usages is less than total available memory on the nodes. Hence, the returned assignment never oversubscribes memory. The returned list of the optimal assignments for this node is stored as a global variable (at least in this pseudo-code) and is here designated as $\mathbf{x}_M^*$. Recall that this assignment is a simple list of numbers, one for each application running on the cluster, indicating the best combination of tasks to run of these applications in order to maximize throughput.

The ADDAPPLICATION function runs when a new application is submitted to the clus-

ter:

Algorithm 4.3.1: ADDAPPLICATION($A^{New}, Cluster$)

$\theta^{New} \leftarrow$ GETRESOURCEREQ(A^{New})
for each machine $M \in$ Cluster
do {
  $\pi^M \leftarrow$ M.GETRESOURCECAPABILITIES()
  $\mathbf{A} \leftarrow$ M.GETRUNNINGAPPS()
  $\Theta \leftarrow$ GETRESOURCEREQ($\mathbf{A}$)
  $\mathbf{A} \leftarrow \mathbf{A} + A^{New}$
  $\Theta \leftarrow \Theta + \theta^{New}$
  $\mathbf{x}^*_{\mathbf{M}} \leftarrow$ COMPUTEOPTIMALASSIGNMENT($\Theta, \pi^M, \mathbf{x_M}$)

In the current implementation, we use exhaustive search to implement the function, $ComputeOptimalAssignment$. At the completion of an application, A^{Done}, Θ is updated by removing A^{Done} from the list of running applications, $\mathbf{A}$. The optimal assignment for the remaining applications is updated by calling the optimization function again.

Algorithm 4.3.2: REMOVEAPPLICATION($A^{Done}, Cluster$)

for each machine $M \in$ Cluster
do {
  $\pi^M \leftarrow$ M.GETRESOURCECAPABILITIES()
  $\mathbf{A} \leftarrow$ M.GETRUNNINGAPPS()
  $\mathbf{A} \leftarrow \mathbf{A} - A^{Done}$
  $\Theta \leftarrow$ GETRESOURCEREQ($\mathbf{A}$)
  $\mathbf{x}^*_{\mathbf{M}} \leftarrow$ COMPUTEOPTIMALASSIGNMENT($\Theta, \pi^M, \mathbf{x_M}$)

The optimal assignment corresponds to the set of (orange and blue) vectors we combined in Figure 4.1 to reach the location in the resource load space where resources are optimally utilized, and hence, throughput is greatest.

Once the optimal number of containers for every running application on every node is determined, they can be exploited to make scheduling decisions. The native Hadoop scheduler sorts task/machine pairs according to whether they are local (data for the task is available on the machine), on the same rack, or remote. We introduce our routine based on our task requirements estimation called BESTAPPTOADD to break ties within each of these tiers as shown in Algorithm 4.3.3. Intuitively, if we have two local apps, we

would run the one most compatible with the machine first.

Algorithm 4.3.3: DARASCHEDULE(Cluster, Request)

for each machine $M \in$ Cluster
do
- **AppsWithLocalTasks** $\leftarrow$ M.GETLOCALAPPS(Request)
- **AppsWithRackTasks** $\leftarrow$ M.GETRACKAPPS(Request)
- **AppsWithOffSwitchTasks** $\leftarrow$ M.GETOFFSWITCHAPPS(Request)
- **if AppsWithLocalTasks** $\neq NULL$
  - **then**
    - $A^{best} \leftarrow$ BESTAPPTOADD(**AppsWithLocalTasks**, M)
    - ASSIGNTASKFORAPP(M, A^{best})
- **else if AppsWithRackTasks** $\neq NULL$
  - **then**
    - $A^{best} \leftarrow$ BESTAPPTOADD(**AppsWithRackTasks**, M)
    - ASSIGNTASKFORAPP(M, A^{best})
  - **else**
    - $A^{best} \leftarrow$ BESTAPPTOADD(**AppsWithOffSwitchTasks**, M)
    - ASSIGNTASKFORAPP(M, A^{best})

Algorithm 4.3.4: BESTAPPTOADD(**A**, M)

return (argmax$_{A \in \mathbf{A}}$ $x^*_{A,M} - x_{A,M}$)

In BESTAPPTOADD, $x^*_{A,M}$ denotes the optimal number of containers to allocate to application A on node M and $x_{A,M}$ is the number of containers node N has currently allocated for application A. The term $x^*_{A,M} - x_{A,M}$ computes the current under-allocation for the application on this node compared to the optimal allocation. When selecting tasks to run on a node, the scheduler uses this number to determine the application A whose actual assignment is lowest compared to its optimal assignment. Hence, by adding tasks of this application we are getting closer to the optimal assignment. Note that our scheduler, like others, never removes running tasks from a node to achieve the optimal assignment. Hence, we do not consider over-allocations in this function.

4.4. EMPIRICAL RESULTS

To evaluate the performance of DARA, we conducted experiments on our six machine Hadoop cluster. Each machine has 8 physical CPU cores, 12 GB of RAM, and runs CentOS 5.6. We compare the performance of DARA Scheduler against CapacityScheduler [37] and FairScheduler [38].

FairScheduler and CapacityScheduler are the schedulers underlying resource-aware big data platforms such as Mesos [85]. Mesos fairly shares cluster resources among different frameworks such as Hadoop and MPI. Mesos implements Dominant Resource Fairness (DRF) [86] to fairly allocate CPU and memory among different users. It assumes a priori knowledge about the resource requirements of jobs, unlike DARA, which automatically infers it. Another reason why comparing DARA to Mesos is not appropriate is that Mesos maximizes fairness rather than throughput, so a comparison would not be fair. All the experiments were repeated multiple times, and in each run same results were obtained. Therefore, we select results from one run to describe in the following subsections.

4.4.1. CONTAINER ALLOCATION

To investigate the container allocation scheme of DARA, various combinations of Hadoop benchmark applications were run on the cluster.

DARA determines the optimal number of containers per application for every combination. Table 4.2 shows these numbers for one of the nodes in the cluster for a few example workloads. All six nodes in the cluster are identical, therefore, the optimal number of containers to allocate is the same for all nodes of the cluster. In the previous chapter, we demonstrated that any heterogeneity in cluster resources can actually be exploited for improved performance as well, and DARA would be able to exploit this seamlessly, too.

	Allocation per Application			
Workload	Pi	Sort	WordCount	RandomWriter
Pi	12	-	-	-
Sort	-	7	-	-
WordCount	-	-	14	-
RandomWriter	-	-	-	3
AggWC	-	-	-	-
Pi + Sort	12	5	-	-
Pi + WordCount	9	-	7	-
Sort + WordCount	-	0	-	-

Table 4.2: Optimal number of containers as computed by DARA for every application and their combinations.

The table shows that DARA allocates the highest number of concurrent containers when only running WordCount alone. This is because WordCount uses a good mix of CPU and disk I/O. On the other hand, RandomWriter generates lots of disk I/O load but does not need CPU for very long. Due to the limited bandwidth of disks, the number of containers that maximize throughput when running RandomWriter alone is very small. For the combination of Sort and WordCount, Sort gets no containers at first because running WordCount will keep both CPU and disk I/O busy.

Note that Pi is so CPU intense and Sort is so disk intense, that the optimal number of Pi containers to run does not change when adding a Sort application. Hence, intuitively, the Sort tasks are being processed "for free."

4.4.2. WORKLOAD DESIGN

Due to unavailability of production workloads, we construct various synthetic workloads based on publicly available Hadoop traces from Facebook [87].

Facebook has published traces of its clusters in order for other to be able to simulate real workloads from Facebook Hadoop clusters. A workflow is defined as a set of MapReduce jobs that are submitted to the cluster in certain intervals. However, Facebook did not publish the job specific resource requirements, nor did they provide the actual jobs (MapReduce programs). Instead, they suggest constructing the simulated workloads purely based on I/O operations that read and write a specific number of bytes to and from disk. This is not sufficient for our purposes. In order to evaluate the effectiveness of our scheduling, we require heterogeneous workflows with varying disk I/O, memory, and CPU requirements. Therefore, we only used the submission intervals from

the Facebook traces, but constructed our own jobs. The workflows we constructed contain between one and five MapReduce jobs, randomly selected from a pool of standard MapReduce jobs. The Hadoop distribution includes benchmark MapReduce jobs. These have different kinds of resource requirements (CPU and I/O), and were therefore, a good pool to choose from. Table 4.3 describes the composition of each workload.

Workload	Job Composition
WL1	Pi
WL2	Sort
WL3	WordCount
WL4	RandomWriter
WL5	Pi, WordCount
WL6	Pi, Sort
WL7	Sort, WordCount
WL8	WordCount, Sort, AggWordCount
WL9	Pi, WordCount, Sort
WL10	AggWordCount, Pi, Pi
WL11	WordCount, Pi, Sort
WL12	Sort, AggWordCount, WordCount, WordCount
WL13	AggWordCount, AggWordCount, Sort, Pi
WL14	Pi, Sort, WordCount, AggWordCount, WordCount
WL15	AggWordCount, WordCount, Sort, WordCount, AggWordCount
WL16	Pi, Pi, Pi, WordCount, WordCount

Table 4.3: Composition of workload that are used compare performance of DARA against Fair and Capacity scheduler.

For each workload, jobs are submitted in the order they appear in Table 4.3. In order to simulate the submission intervals between two jobs in accordance with the Facebook traces, we randomly draw samples from the populations of submission intervals provided in Facebook Hadoop traces.

Figure 4.6 shows a histogram of the arrival intervals of jobs in Facebook's traces.

For our experiments we construct 16 synthetic workloads, and every workload contains one or more MapReduce applications from Hadoop benchmark examples.

4.4.3. WORKLOAD SPEEDUP

We compared the performance of DARA against CapacityScheduler and FairScheduler in terms of speedup of workload execution on the cluster. Speedup gained by DARA is measured by dividing time to complete workload using FairScheduler and CapacityScheduler by time to complete workload using DARA. Workload completion time is defined as the time from the beginning of execution until the completion the application finishing last. Speedup results are shown in Figure 4.7.

Our experimental results demonstrate that DARA speeds up execution of all the workloads compared to CapacityScheduler and FairScheduler. Even though DARA is designed to optimize the execution of map tasks only, we can see the speedup of overall MapReduce applications. The improved speedup shows that DARA is assigning tasks to machines that efficiently satisfy the the task resource requirements.

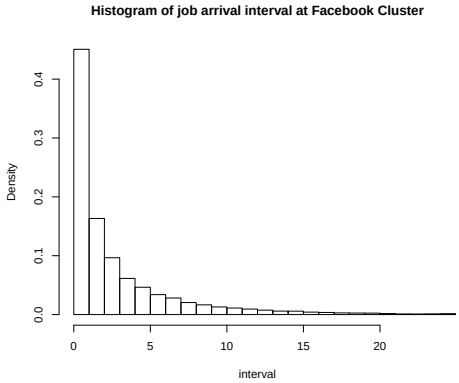


Figure 4.6: Histograms of time interval (in seconds) between two application submissions at Facebook Hadoop cluster.

4.4.4. CLUSTER THROUGHPUT

To further evaluate the container allocation policy of DARA, we compare the performance of DARA against CapacityScheduler and FairScheduler in terms of throughput.

The cluster throughput is measured using Equation 4.4. The time variable used in Equation 4.4 is measured as the time to complete a workload. In case of multiple applications in a workload, the completion time of the application finishing last is used in the throughput measurement. Results are shown in Figure 4.8.

The results show that DARA delivers higher throughput than than FairSchedulers and CapacitySchedulers. For applications that are CPU intensive, DARA assigns more containers than for I/O intensive applications. For the latter, it turns out that the optimal number of containers is significantly less than the number of CPU cores. Also, the other rule of thumb is to assign containers based on the amount of RAM divided by 1 GB, but this would not assign the optimal number of containers in terms of throughput. DARA dynamically adapts its container allocation to the resource requirements of the mix of applications running at any one time, and as a result achieves higher throughput.

4.4.5. RESOURCE UTILIZATION

Another way to understand the improved throughput achieved by DARA is to consider resource utilization. To illustrate this, we ran Pi and Sort applications together on the cluster and monitored the CPU usage while both the applications are active.

Recall that Pi is a CPU intensive application and Sort is an I/O intensive application. Therefore, both CPU and disk I/O are exploited during the execution of applications. Figures 4.9 and 4.10 show the CPU utilization for the DARA scheduler and FairScheduler.

DARA is exploiting CPU much more efficiently than FairScheduler. For DARA, the average user CPU usage is around 35 percent and waiting for I/O is around 20 percent (Figure 4.9). On the other hand, for FairScheduler with the average user CPU utilization around 17 percent and waiting for I/O is around 20 percent (Figure 4.10). These results

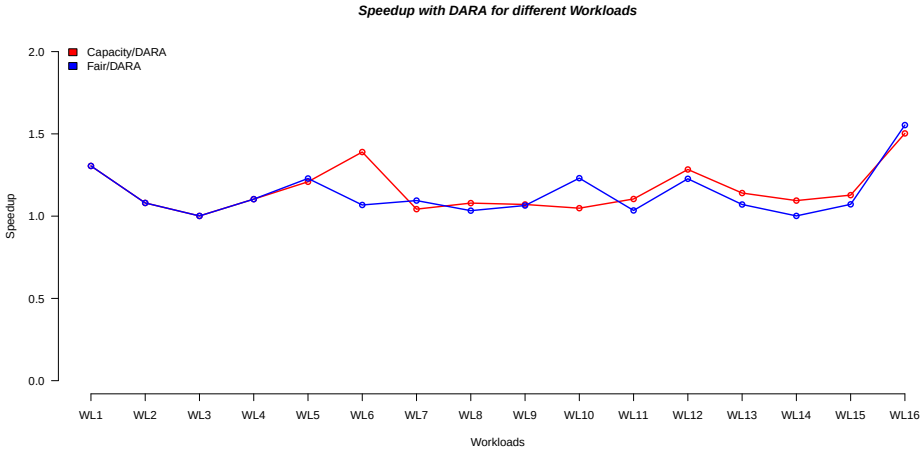


Figure 4.7: Speedup gained by running workloads with DARA compared to FairScheduler and CapacityScheduler. For our 16 workloads, DARA provides average speedup of 1.14 compared to FairScheduler and 1.16 compared to CapacityScheduler. X axis indicates which workload was used for the experiment.

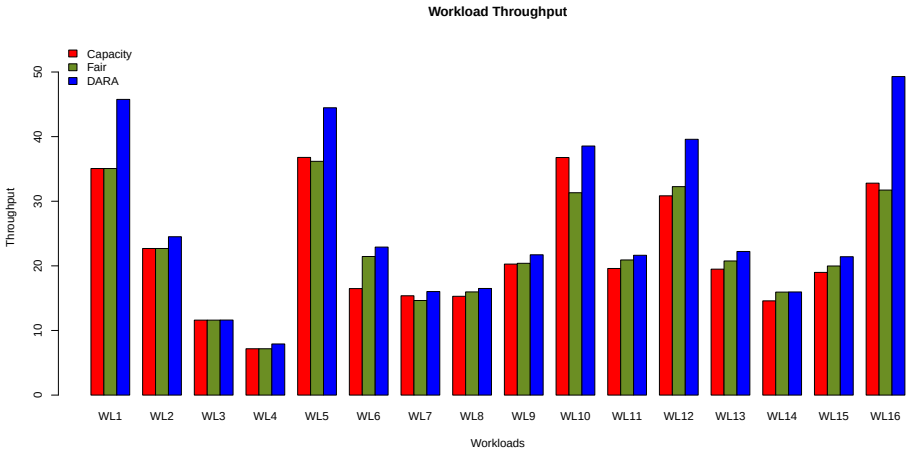


Figure 4.8: Throughput comparison on Hadoop benchmarks. For our 16 workloads, DARA increases the average throughput by 16% compared to CapacityScheduler and 14% compared to FairScheduler.

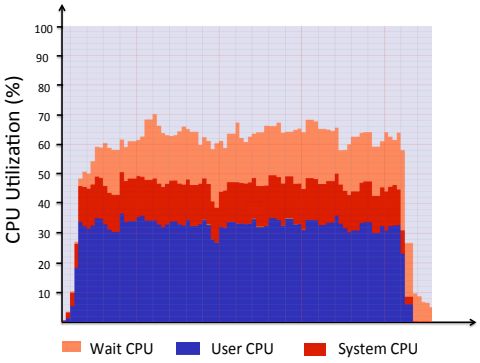


Figure 4.9: CPU Utilization of the cluster when Pi+Sort are in parallel executed by DARA

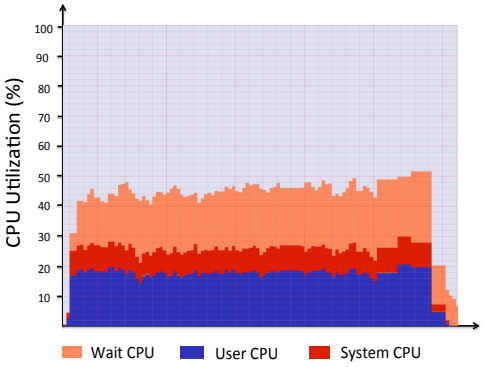


Figure 4.10: CPU Utilization of the cluster when Pi+Sort are executed by FairScheduler

demonstrate that DARA efficiently exploits multiple resources by allocating containers more intelligently based on its automatically inferred knowledge about their resource requirements. In this example, intuitively, DARA combines tasks a way that when one application is waiting for I/O, it can be exploited to do the additional processing.

DARA is carefully designed to improve throughput without oversubscribing the available memory of a node. To validate this design feature, we monitor memory used by every node of the cluster. Figure 4.11 shows the memory usage of a node in the cluster during the execution of all the workloads with DARA. This observations shows that the actual memory used is significantly under the total memory on the node. Similar memory usage is reported from the other nodes in cluster.

4.5. CONCLUSION

DARA represents a novel approach for scheduling jobs on Hadoop clusters to maximize throughput. The framework dynamically determines the optimal number of containers to run for each application and node. Unlike previous schedulers, DARA uses an estimate of the actual resource requirements of running applications together with re-

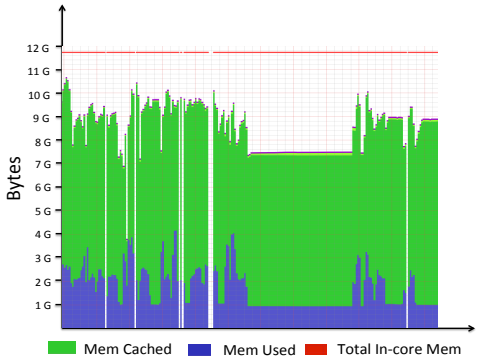


Figure 4.11: Memory usage of a node in the cluster while running different workloads using DARA.

source capabilities of nodes to optimize for throughput. To the best of our knowledge, none of the existing (published) schedulers consider CPU and disk I/O use when making scheduling decisions in online manner.

Our main contributions are twofold. First, we present a model of task completion time that can be trained offline to account for node-specific capabilities, which we then parameterize online to account for application specific resource requirements, based solely on the actual overall resource consumption of the first map task. Second, we present an algorithm that can use this model to make scheduling decisions that improve throughput. Empirical results confirm that DARA scheduler performs better than existing Hadoop schedulers in terms of throughput, speedup, and resource utilization. While DARA is specifically for Hadoop, the underlying idea and overarching approach generalizes to other frameworks of distributed computing as well.

5

IDENTIFYING PERFORMANCE PROBLEMS IN HETEROGENEOUS HADOOP CLUSTERS

So far in this thesis, we have covered the scheduling aspects of the Hadoop CMS. As we discussed in our goals in Chapter 1, the other important component of the CMS is monitoring and diagnosis. High throughput is necessary for the businesses of the companies that depend on Hadoop. A cluster can have a higher throughput if all the machines are producing output at a higher rate. If the machines start failing or slowing down then the cluster productivity might be severely affected. No matter how good or costly the machines are, in practice they can abruptly fail because of unexpected faults. In such scenarios, it is important to take certain actions such that the cluster production can be maintained at a certain level. Therefore, from the production point of view, it is important to keep monitoring the performance of machines and to identify faults in machines that might cause slowdown or failure.

In Chapter 2, we introduced two types of faults, *hard faults* and *soft faults*. CPU failure, disk failure, network failure can be seen as hard faults, resulting in failing applications. Soft faults, on the other hand, don't cause application failure on machines; however, applications complete at a lower than usual rate. Limping hardware, unnecessary background processes, or poor task scheduling resulting in overloaded machines, can all be seen as soft faults. These faults create resource congestions, such as CPU or I/O or network congestion, resulting in slowdowns. The heartbeat protocol works on the idea that if a machine is alive, it will send the heartbeat message. But in the presence of soft faults, a machine can still send the heartbeat message. In spite of the presence of soft faults in machines, as long as the machine is alive it can send the heartbeats. As a result, hard faults can be detected by Hadoop heartbeat protocol, however, it fails to detect soft faults. Compared to hard faults, the causes and impacts of the soft faults is a little more complex to understand.

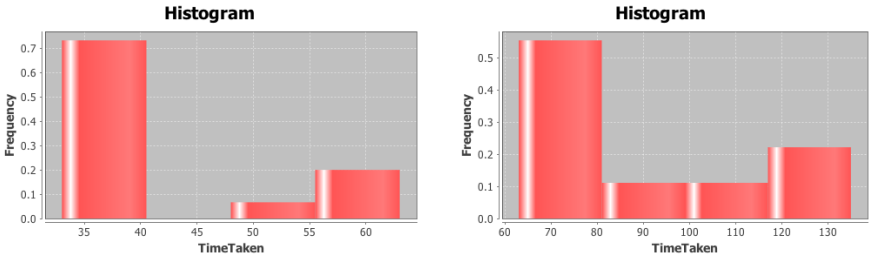
One of the well-known solutions to identify soft faults is the peer-similarity [65] [11], which compares the performance of map tasks from a given application on cluster machines to find the machine(s) with soft faults. Tasks from the applications are executed on each machine of the cluster, and task completion times on each machine are observed. Tasks completion times are compared among machines, and machines on which tasks are taking longer to finish are suspected to be faulty. The peer-similarity approach works well in practice, but has many limitations. The approach relies on the assumption that the cluster is homogeneous, meaning that all machines have the same hardware and software configuration. As we discussed in Chapter 3, this assumption is not always true in production clusters. Moreover, the peer-similarity approach is limited to identifying faulty machines; it can not detect which resources are affected by the faults. To implement a resource-aware CMS, however, it is important to monitor performance of each resource in every machine of the cluster.

In this chapter, we investigate the problem of identifying resource specific soft faults in heterogeneous Hadoop clusters. To address this problem, we derive two data-driven approaches for identifying the performance problems that are applicable to the heterogeneous clusters. In our first method, we propose an approach to detect resource specific soft faults in machines [88]. We present a very simple probabilistic model-based approach to detect such soft faults. To build the probabilistic model, we assume that the task resource requirements and machine capabilities are already known. The performance of a machine is measured in terms of slowdown of a machine. A slower machine will process tasks at a lower rate compared to faster machines.

This probabilistic model-based approach has a few shortcomings. First, it assumes that resource requirements of tasks and resource capabilities of machines are known in advance. In many practical cases, both of these parameters are generally unknown. Second, the approach works in offline mode, which means that while detecting the faults, the cluster needs to stop processing applications. Therefore, this approach can reduce overall cluster productivity.

In our next approach, we improve the previous probabilistic model-based approach [89], where we relax the assumptions that the requirements and performance metrics are known. Rather, we propose an unsupervised learning-based approach to learn unknown parameters. To make our approach appealing, we develop the monitoring tool that works in online mode; therefore, soft-faults can be detected without stopping production. Our approach continuously estimates the machine capabilities in terms of the slowdown of every machine, and slowdown is measured for every kind of resource (CPU or disk) separately. Hence, our solution is able to explain what kinds of resources are affected in the presence of faults in real time. While continuously monitoring the slowdown of machines, a radical increment in slowdown indicates the presence of a fault. This is accomplished without *any* additional input from the user or the cluster administrator.

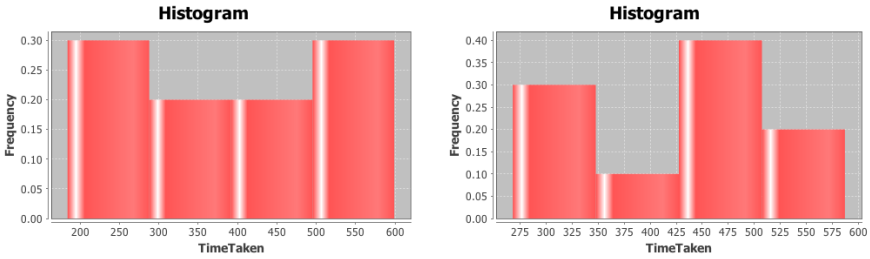
In this chapter, we only develop methodologies to find soft faults in Hadoop clusters. We believe that hard faults can be efficiently diagnosed using the Hadoop heartbeat protocol. In the next section, we describe the nature and impact of soft faults on the cluster.



(a) Time distribution for `WordCount` without CPU hogging

(b) Time distribution for `WordCount` with CPU hogging

Figure 5.1: CPU intensive MapReduce jobs such as `WordCount` are strongly affected in their completion time by over-subscription of the CPU (note the number of seconds shown on the x-axis).



(a) Time distribution of `RandomWriter` without CPU hogging

(b) Time distribution of `RandomWriter` with CPU hogging

Figure 5.2: MapReduce jobs that are not as CPU intensive, such as `RandomWriter`, are much less strongly affected in their completion time by over-subscription of the CPU.

5.1. SOFT FAULTS

We consider soft faults in the form of resource contention on the cluster machines. There are numerous ways for resource contention, such as any hardware fault, limping hardware, oversubscription of resources due to unknown background processes. Resource contention also could be the result of bad scheduling. For the purpose of understanding the impact of these resource problems, we simulate two types of soft faults: CPU hogging and disk I/O hogging. To simulate the CPU hogging, we run multiple infinite while loops to keep all the CPU cores busy. [1] As a result, tasks of CPU intensive applications took much longer to complete, as shown in Figure 5.1. On the other hand, disk I/O intensive jobs, such as the `RandomWriter`, are not affected nearly as strongly, as can be seen in Figure 5.2.

These experiments suggest that if a certain resource has performance problems, then it mainly impacts the performance of the applications that are mostly exploiting that particular resource. In the next section, we present our first approach to identify soft

[1] To simulate the disk hogging, we read and write random data from the disks.

faults, which is based on the kinds of resources used by applications.

5.2. RESOURCE CLASSIFICATION BASED APPROACH

This approach for determining performance problems of a heterogeneous Hadoop cluster consists of two steps. First, we learn a performance model of every machine in the cluster for each class of applications. Application classes are created based on what kinds of resources are primarily used by applications. In the second step, these performance models are used to estimate how long a given new task should take on a given machine. Intuitively, if a new task takes much longer than predicted by the model, the machine is likely to be suffering from a fault. By comparing the impact such a fault has on the completion time for tasks of different classes, meaning different resource profiles, it is possible to diagnose the kind of fault in the sense of determining which resource is affected (for example, CPU, disk I/O).

5.2.1. ASSUMPTIONS

Our approach makes the following assumptions:

1. The resource profile, in terms of CPU, and I/O of a given task, is known in advance.
2. For each application, there is one resource that is the bottleneck which dominates the resource requirements for all map tasks of the application. For instance, a CPU intensive application is only marginally affected by disk I/O contention.
3. The class-specific, relative-task completion time is machine independent. This means that for two applications, A and B, of the same class, if B takes a factor of X longer than A on one machine, then this factor will also apply to the completion times for A and B on another machine. This is true even if the two machines have different configurations. We will further elaborate on this and provide empirical evidence for this assumption in Section 5.2.4.
4. The completion times for tasks executing on the same machine are independent of each other.
5. The distribution of completion times for the tasks of a specific application on a specific machine follows a Gaussian distribution. This means two things: repeating the exact same task multiple times would lead to a Gaussian distribution, and also the execution time for individual tasks of the same application (processing distinct 64MB blocks of the same data set using the same algorithm) follows this distribution.

   Our use of a Gaussian distribution in this chapter is a first, pragmatic choice. In principle, a distribution that does not extend into the negative values would be a better choice for this application.

Given these assumptions, our approach is captured by Figure 5.3. We can infer from the completion time of applications J1 and J2 on machine M1 to the completion time of J2 on M2 once the completion time for J1 on M2 is known. This is possible based on Assumption 3.

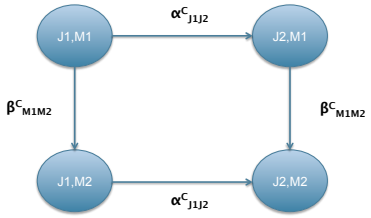


Figure 5.3: The *complexity coefficient* α describes the relative hardness or complexity of an application compared to other applications, and is assumed to be machine independent. Likewise, β is a coefficient that captures the relative performance of one machine compared to another. It reflects the relative hardware configuration of the machines. Both of these coefficients are specific to a class of application, denoted C.

5.2.2. CLASSES

Our diagnosis approach relies on an understanding of the resource profile of a task. Every task will make use of system resources in different ways. Some applications are more CPU intensive, while others read and write from and to disk more than others. In this chapter, as stated in Assumption 2, we assume that for each application there is a single resource that forms the bottleneck. We only consider two classes: CPU intensive applications and disk I/O intensive applications. The assumption states that the intersection between these two classes is empty. In the rest of the chapter, we will denote the class of applications whose bottleneck is resource R as C_R. Hence, we will be talking about two classes: C_{CPU} and C_{IO}.

In our approach, we run our diagnosis for every class and in the end, combine these diagnoses to recommend the specific root cause for a slowdown. For example, if we observe that on a particular machine only CPU intensive applications take longer than predicted by the model, this suggests that the machine is suffering from CPU over-subscription, which could be caused, for example, by some background operating system or zombie process.

BASE MODEL OF CLASS

In our approach, we assume that initially, for instance, during first installation of the cluster, a set of prototypical "*base*" applications can be run on each machine. We assume that one such application for each *class* of applications is run on each machine, where there is one class for each dimension in the configuration space (for instance, CPU speed, disk I/O speed, or amount of RAM). This is discussed further in the next section.

For the two classes we are considering in this chapter, C_{CPU} and C_{IO}, `WordCount` and `RandomWriter` are selected as base applications, respectively. The model for another application of a class is determined relative to the base application model. Given application J belonging to class C_R, we define $\alpha_J^{C_R}$, the *complexity coefficient* of application J with respect to resource R, as the factor of how much longer J takes to compute relative to the base application for C_R. Assumption 3 states that this factor is equal for every machine. That means that if, for instance, the original `WordCount` task on machine 1 takes 60 seconds, and another application that is also CPU intense takes 120 seconds on the same machine, then we assume that the same factor of 2 would be observed on other

machines as well. Empirical justification and the procedure for estimating application complexity coefficients is described in Section 5.2.4.

5.2.3. BEHAVIORAL MODEL CONSTRUCTION

To learn the models, we gather time samples of mapping tasks of every application from the log files generated by Hadoop. We collect the system logs produced by Hadoop's native logging feature on the master node. Subsequently, we parse these log files to collect timing samples for every machine in the cluster. Each entry in the log can be treated as an event and is marked with a time stamp. An event is used to determine when a task is started and when it finishes on a specific machine. The duration of mapping tasks is computed by subtracting those time stamps. We denote the average duration of all mapping tasks belonging to application J on machine M by $t_{J,M}$.

MODEL LEARNING

It is assumed that mapping task durations of the same application are distributed normally for any specific machine. The model for each application class C_R is hence described by a mean $\hat{\mu}_{J,R}$ and variance $\hat{\sigma}_{J,R}$. For the base applications of each class, we can easily learn these parameters from the samples gathered during the initial execution of base applications.

For each class C_R, we run the base application of the class on the Hadoop cluster and collect samples $t_{J,M,i}$ for every machine M. Recall that Hadoop schedules the execution of the tasks belonging to an application onto the available data machines according to where the data to be processed is stored. Since the data is distributed roughly uniformly, this process produces a number of samples for each machine. We assume that during this initial learning, no faults occur on the cluster.

For an application J, the mean and variance for machine M can be estimated using the standard maximum likelihood estimator over the samples collected for tasks belonging to the base application of C_R:

$$\hat{\mu}_{J,M} = \frac{1}{N_{J,M}} \sum_i t_{J,M,i} \tag{5.1}$$

$$\hat{\sigma}^2_{J,M} = \frac{1}{N_{J,M}} \sum_i (t_{J,M,i} - \hat{\mu}_{J,M})^2 \tag{5.2}$$

Here, $N_{J,M}$ is the total number of samples collected for machine M and application J. In our experiments, we use `WordCount` as the base application for class C_{CPU}, and `RandomWriter` for C_{IO}. We introduce the shorthand $J_{CPU} = WordCount$ and $J_{IO} = RandomWriter$. In the rest of the chapter, we refer to $\hat{\mu}_{J_{CPU},M}$ and $\hat{\sigma}^2_{J_{CPU},M}$ as the model parameters of the base application for C_{CPU}, and $\hat{\mu}_{J_{IO},M}$ and $\hat{\sigma}^2_{J_{IO},M}$, respectively, for C_{IO}.

5.2.4. ESTIMATING APPLICATION COMPLEXITY

Assumption 3 states that the relative complexity of an application compared to its base application does not depend on the machine it is executed on. To verify this assumption, we conducted an experiment with two new MapReduce applications: J'_{CPU} and J'_{IO}. The first term, J'_{CPU}, belongs to class C_{CPU}, and second term, J'_{IO}, belongs to C_{IO}. For every

machine M in the Hadoop cluster, we computed $\alpha^{CPU}_{J'_{CPU},M}$ and $\alpha^{IO}_{J'_{IO},M}$ as:

$$\alpha^{CPU}_{J'_{CPU},M} = \frac{\hat{\mu}_{J'_{CPU},M}}{\hat{\mu}_{CPU,M}} \tag{5.3}$$

$$\alpha^{IO}_{J'_{IO},M} = \frac{\hat{\mu}_{J'_{IO},M}}{\hat{\mu}_{IO,M}} \tag{5.4}$$

The above equations use the ratio of the means of task completion times to estimate αs, because using the means reduces out the measurements noise.

Figure 5.4: The values for complexity coefficient α for applications belonging to C_{CPU} (`WordCount`) and C_{IO} (`RandomWriter`). Note that even though the machines are heterogeneous, the relative complexity of an application is comparable between machines.

Figure 5.4 shows the results. The figure shows that the complexity coefficient for each class is fairly similar across machines, providing empirical justification for Assumption 3.

5.2.5. DIAGNOSIS

Given the duration of mapping tasks belonging to a previously unseen application $J2$ on a number of machines $\{M_i\}_i$, we want to determine whether any of the machines is having a fault and what the nature of the fault is. We do this by estimating for each machine how long each task should take under normal circumstances, and then use this information to determine a likelihood for a new observation (task duration) to indicate abnormality of the machine. The idea for predicting the duration is that we can estimate the model, meaning, the distribution of task duration, for a new task executing on a specific machine by scaling the base model of the class of the application for this machine by the complexity coefficient. The complexity coefficient, in turn, can be estimated from all other machines that have already run tasks of this application.

PREDICTING APPLICATION COMPLETION TIME

When diagnosing machine M_k for an application of class C_R, to get a better estimate of the complexity coefficient, every machine's α value for this application will be used

except M_k's. Hence, $\alpha^R_{J,*}$ can be estimated as follows:

$$\alpha^R_{J,*} = \frac{1}{N-1} \sum_{i \neq k} \alpha^R_{J,M_i} \tag{5.5}$$

Here, N is the number of machines in the cluster that ran mapping tasks for this application.

On machine M_k, the model for application $J2$ can be inferred from the model for M_k for applications of the respective class. Given the estimated complexity coefficient $\alpha^R_{J,*}$ for application $J2$ belonging to class C_R, the mean and variance for $J2$ on M_k can be estimated as:

$$\hat{\mu}^*_{J2,M_k} = \alpha^R_{J2,*} \cdot \hat{\mu}_{R,M_k} \tag{5.6}$$

$$\hat{\sigma}^*_{J2,M_k} = \alpha^R_{J2,*} \cdot \hat{\sigma}_{R,M_k} \tag{5.7}$$

Recall that the approximate model of an application on a machine, characterized by these two parameters, describes the (approximate) distribution of durations of tasks of this application in this machine. Therefore, the approximate model can be used for diagnosis by computing the *relative likelihood* for an observed duration x of a task using the probability density function (pdf) of the underlying distribution; in this case, the pdf of the normal distribution is:

$$f(x; \hat{\mu}, \hat{\sigma}) = \frac{1}{\hat{\sigma}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\hat{\mu}}{\hat{\sigma}})^2} \tag{5.8}$$

If a machine is suffering from over-subscription of a resource that is extensively used by the task, then this resource over-subscription will slow down the task and make the observed task completion time (duration) less likely. Hence, we consider a machine M_k to potentially have an over-subscription of resource R, if for an application J'_R of class C_R, $f(t_{J'_R,M_k}; \hat{\mu}^*_{J'_R,M_k}, \hat{\sigma}^*_{J'_R,M_k})$ is the least compared to the average relative likelihood for the durations observed for tasks of J'_R on all other machines. That is when:

$$f(t_{J'_R,M_k}; \hat{\mu}^*_{J'_R,M_k}, \hat{\sigma}^*_{J'_R,M_k}) \ll \frac{1}{N-1} \sum_{i \neq k} f(t_{J'_R,M_i}; \hat{\mu}^*_{J'_R,M_i}, \hat{\sigma}^*_{J'_R,M_i})$$

5.2.6. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our diagnosis approach, we collected task durations from the log files of our 14-machine Hadoop cluster for two different instances of the `WordCount` and `RandomWriter` applications. The 14 machines are quite heterogeneous in their hardware configuration since these machines were purchased at different times and for different original purposes. This heterogeneity is reflected in Figure 3.1.

Following the described approach, we divided the experiments into two phases: a learning phase and a diagnosis phase. In the learning phase, we learned the model for the two base applications, one for `WordCount` and one for `RandomWriter`, for each machine in the cluster. During this phase, we made sure that every machine was functioning flawlessly.

During the diagnosis phase, we simulated two types of resource contention: disk I/O contention on machine 10 and CPU contention on machine 13. These faults were simulated by running additional programs on the machines that made excessive use of these resources ("hogging"). For CPU hogging, we over-subscribed each CPU core on the machine by a factor of two by running one extra program per core that ran an infinite loop with a basic arithmetic operation inside. Hence, the overall load per CPU core was around 2.0 when the Hadoop applications were executing. For disk hogging, we ran a program that repeatedly wrote large files to the disks used by HDFS.

With these fault simulations in place, we ran a different `WordCount` application and a different `RandomWriter` application. These applications differed from the applications used as base applications in that they repeated certain sub-tasks multiple times. This was to ensure these applications have roughly the same resource profile as the base applications but at the same time take noticeably longer. As can be seen in Figure 5.4, these applications took roughly twice as long as their respective base application.

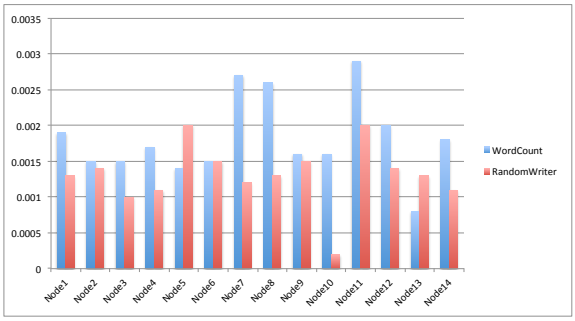


Figure 5.5: Results of the diagnosis for an instance of the `WordCount` and an instance of the `RandomWriter` application. On machine10 (Node10), we injected disk I/O contention. On machine13 (Node13), we injected CPU contention. The y-axis shows the relative likelihood for observed task durations.

RESULTS

Figure 5.5 summarizes the results of our experiment. On the x-axis we show the machine number. For each machine, two relative likelihood values for the observed average task durations are shown: one for the new `WordCount` application and one for the new `RandomWriter` application.

As can be seen, the relative likelihood for the observed task durations of the second `WordCount` application on machine13 is the lowest (almost zero) compared to other machines. Similarly, the relative likelihood for the observed task durations of the second `RandomWriter` application on machine 10 is lowest. This suggests that it is indeed possible to use the presented approach to identify abnormally behaving machines in a heterogeneous Hadoop cluster. Any machine whose average task duration has a relative likelihood that is lowest compared to the average likelihood over other machines is a candidate diagnosis. This is because, according to the model, it is unlikely to observe such average task durations on a machine that is behaving normally. Applied to the results shown in the figure, this approach would correctly identify machines 10 and 13 as being abnormal.

Furthermore, note that the performance of machine 13 on the `RandomWriter` is not noticeably reduced by the presence of CPU contention, and likewise, the effect of disk I/O contention on machine 10 does not impact the completion time of the CPU intense `WordCount` application to a noticeable degree. This leads us to believe that this approach is indeed capable of determining the type of fault occurring on a machine, at least to the level of detail of which resource is affected by the fault. Applied to our results, this approach would correctly determine that machine 10 is suffering from disk I/O contention, and machine 13 is experiencing CPU contention.

All the experiments were repeated multiple times, and in each run same results were obtained. Therefore, we select results from one run to describe in this section.

The results demonstrate that the probabilistic model-based approach successfully uncovered soft faults on a heterogeneous Hadoop cluster. However, this approach makes lots of assumptions, such as the resource requirements are known in advance, and there is only one resource that can cause bottlenecks for one application. In practice, these assumptions might not be true. In many cases, when a programmer or user submits applications to the cluster, the resource usage of the application might not be available. Similarly, in many cases, it might happen that more than one resources causes the bottleneck. In these cases, our approach can not be used to diagnose faults. Additionally, this approach works in offline mode, which can cause lower productivity. Therefore, in the next section, we present our next approach in which we relax these assumptions while identifying performance problems in heterogeneous Hadoop clusters. Moreover, our next approach is also online in nature, and therefore, it is not required to halt production to identify the faults.

5.3. CONTINUOUS MONITORING APPROACH

Now we describe the methodology to implement the monitoring tool to detect resource specific performance anomalies in a machine. Our monitoring approach is based on the static model of task completion time that we defined in terms of *task requirements* and *machine performance*. Unfortunately, these requirements and performance are not directly observable and there is no simple way to predict task execution times given only machine hardware specifications and task source code. Instead, we adopt a learning approach where machine parameters are continuously learned from observed task completion times using our model.

As we defined in Chapter 3, the static model predicts the execution time of a task on a machine given the task resource requirements and the performance of the machine. The task resource requirements are represented by a vector, $\theta = [\theta_1, \theta_2, \ldots, \theta_N]$, where each component represents a resource requirement of a certain type (for example, CPU, disk I/O, network I/O). The performance of the machine is measured by the slowdown, which is described by a corresponding vector, $\gamma = [\gamma_1, \gamma_2, \ldots, \gamma_N]$. It must be noted that the slow parameter γ is the inverse of machine capability κ that we defined in Chapter 3. The total time $T^{i,j}$ to process a task of application i on machine j is the sum of the times of each resource requirement and the respective machine performance:

$$T^{i,j} = \sum_k \theta_k^i \gamma_k^j + \Omega^j \tag{5.9}$$

Here, Ω^j represents the fixed overhead to start the task on the machine. We assume that

every application imposes the same amount of overhead on a given machine. As we discussed eariler in this work, we consider a two dimensional model in which $\gamma = [\gamma_c, \gamma_d]$ represents computation and disk I/O machine performance. Here, $\theta = [\theta_c, \theta_d]$ represents the corresponding task requirements. The task duration model will be as follows:

$$T^{i,j} = \theta_c^i \gamma_c^j + \theta_d^i \gamma_d^j + \Omega^j. \tag{5.10}$$

In order to monitor the state of every machine, we use observations of map execution times $T^{i,j}$ to infer the values of γ_c^j and γ_d^j. For a specific machine, a significant increase in γ_c^j or γ_d^j highlights a performance problem (machine has become slower) such as a runaway process or disk contention. This will allow an administrator to identify the root cause of cluster performance problems.

5.3.1. MONITORING MODULE

We adopt a Bayesian perspective in which we start with a prior distribution over the parameters of the model $P(\gamma_c, \gamma_d, \theta_c, \theta_d)$ and update these using observations of mapping times $\{T^{i,j}\}_1^N$ where i is a task index and j is a machine on which it was run to get the posterior $P(\gamma_c, \gamma_d, \theta_c, \theta_d | \{T^{i,j}\}_1^N)$.

We assume that the observed execution times $T^{i,j}$ are normally distributed around the value predicted by the task duration model of Equation 5.10. The uncertainty is given by a standard deviation σ_j associated with machine j:

$$T^{i,j} \sim \mathcal{N}\left(\theta_c^i \gamma_c^j + \theta_d^i \gamma_d^j + \Omega^j, \sigma^j\right) \tag{5.11}$$

Given the likelihood function for observed time samples based on parameter values, the posterior distribution of the slowdown of a machine, and the resource profile of an application can be derived using Bayes rule. For our model with only CPU and disk I/O resources, the likelihood function has the following form:

$$\begin{aligned} & p(T^{i,j} \mid \theta_c^i, \theta_d^i, \gamma_c^j, \gamma_d^j, \sigma^j) \\ = \; & \frac{1}{\sqrt{2\pi}\sigma^j} \cdot \exp \frac{\left(T^{i,j} - \theta_c^i \gamma_c^j - \theta_d^i \gamma_d^j - \Omega^j\right)^2}{2{\sigma^j}^2} \end{aligned} \tag{5.12}$$

We do not know the exact functional form of the prior $P(\gamma_c, \gamma_d, \theta_c, \theta_d)$ nor the posterior $P(\gamma_c, \gamma_d, \theta_c, \theta_d | \{T^{i,j}\}_{i,j})$, but we do know that all of the variables are correlated by the observations. We propose an approximate decomposition in terms of the product of two bivariate normal distributions, which we optimize using a heuristic re-estimation procedure. This approximation is motivated by the observation that the individual slowdown parameters γ_c and γ_d are linearly, negatively correlated given an observation because they enter into the likelihood linearly and represent a sum that explains the time taken. The intuition is that execution time can be explained by a fast CPU and slow disk I/O or fast disk I/O and slow CPU. We therefore employ a bivariate normal with full covariance matrix to represent the joint probability over the slowdown parameters. A similar argument motivates the use of a second bivariate normal to represent the symmetric linear correlation in the resource requirements parameters of the task. We use bivariate

Gaussian distribution because, due to their simple mathematical form, we can derive the analytical expression of posterior distribution:

$$P(\gamma_c,\gamma_d,\theta_c,\theta_d|\{T^{i,j}\}_{i,j}) = P(\gamma_c,\gamma_d|\{T^{i,j}\}_{i,j})P(\theta_c,\theta_d|\{T^{i,j}\}_{i,j})$$

This factored joint is approximated by iteratively updating each component distribution using Algorithm 5.3.1. We initialize the slowdown parameters $\gamma_c^{j,t}$ and $\gamma_d^{j,t}$ from offline experiments. The algorithm then computes the posterior distribution of the resource profile of every application in the cluster for the given time sample and prior values of machine performance measure. In this step, the mean values of the slowdown parameters $\bar{\gamma}_c^{j,t}$ and $\bar{\gamma}_d^{j,t}$ are used to determine the task requirements $p(\theta_c^{i,t},\theta_d^{i,t})$. Similarly, in the second step, machine slowdown parameters are updated based on the posterior distribution of the resource profile. The updated $p(\gamma_c^{j,t+1},\gamma_d^{j,t+1})$ is used as the prior distribution for further iterations.

5

Algorithm 5.3.1: MONITORCLUSTER(T^{ij})

for each Iteration t ∈ Total Iterations

do $\begin{cases} p(\theta_c^{i,t},\theta_d^{i,t} \mid \{T^{i,j},\bar{\gamma}_c^{j,t},\bar{\gamma}_d^{j,t}\}_j) \\ p(\gamma_c^{j,t+1},\gamma_d^{j,t+1} \mid \{T^{i,j},\bar{\theta}_c^{i,t},\bar{\theta}_d^{i,t}\}_i) \end{cases}$

There are two major challenges to implementing the proposed heuristic. First, we need to derive the marginal distributions $p(\gamma_c,\gamma_d)$ and $p(\theta_c,\theta_d)$. Second, we need to explain how the performance parameters γ_c and γ_d are initialized.

5.3.2. UPDATING MARGINALS

In this section, we derive the update for the marginal distribution of task requirements $p(\theta_c,\theta_d)$. The task model shown in Equation 5.10 is symmetric in terms of machine slowdown and requirement parameters, so the same form of update can be used for the machine performance parameters $p(\gamma_c,\gamma_d)$. To derive the posterior marginal distribution $p(\theta_c,\theta_d)$, we treat θ_c and θ_d as random variables in Equation 5.10. These random variables are assumed to follow bivariate Gaussian distribution. The uncertainty about the slowdown is therefore captured by a covariance matrix $\Sigma_{\theta_c^i,\theta_d^i}$:

$$[\theta_c^i,\theta_d^i] \sim \mathcal{N}([\mu_{\theta_c^i},\mu_{\theta_d^i}],\Sigma_{\theta_c^i,\theta_d^i}) \tag{5.13}$$

We assume that the observed execution time $T^{i,j}$ is normally distributed as described in Equation 5.12. For the derivation, we substitute the expected values of $\bar{\gamma}_c^j$ and $\bar{\gamma}_d^j$ for the parameters. When an application is first submitted, we assume that the resource requirements for its tasks are completely unknown. Assuming an uninformative prior, the posterior distribution after the first observation is just proportional to the likelihood:

$$p(\theta_c^i,\theta_d^i \mid T^{i,j}) = \frac{1}{\sqrt{2\pi}\sigma^j} \cdot \exp \frac{\left(T^{i,j} - \theta_c^i\bar{\gamma}_c^j - \theta_d^i\bar{\gamma}_d^j - \Omega^j\right)^2}{2{\sigma^j}^2}$$

For the second and subsequent updates, we have a definite prior distribution and likelihood function. These two are multiplied to obtain the density of the second posterior update. Let the first experiment be on machine j with slowdown γ^j, and let the observed time be $T^{i,j}$. Let the second experiment be on machine k with slowdown γ^k, and let the observed time be $T^{i,k}$. The resulting posterior distribution is as follows:

$$p(\theta_c^i, \theta_d^i \mid T^{i,j}, T^{i,k}) =$$
$$\frac{1}{\sqrt{2\pi}} \cdot \exp\left[\frac{\left(T^{i,j} - \theta_c^i \bar{\gamma}_c^j - \theta_d^i \bar{\gamma}_d^j - \Omega^j\right)^2}{2\sigma^{j2}} + \frac{\left(T^{i,k} - \theta_c^i \bar{\gamma}_c^k - \theta_d^i \bar{\gamma}_d^k - \Omega^j\right)^2}{2\,\sigma^{k2}} \right]$$

With every time sample we can recover the mean $\mu_{\theta_c^i, \theta_d^i}$ and covariance matrix $\Sigma_{\theta_c^i, \theta_d^i}$ by using the property of bivariate Gaussian distributions. Expanding the exponent of Equation 5.14, and collecting the θ_c^i and θ_d^i terms, gives us a conic section in standard form:

$$a_{20}{\theta_c^i}^2 + a_{10}\theta_c^i + a_{11}\theta_c^i\theta_d^i + a_{01}\theta_d^i + a_{02}{\theta_d^i}^2 + a_{00} = 0$$

As we discussed in Chapter 3, there is a transformation to map between the coefficients of a conic in standard form and the parameters of a Gaussian distribution [83]. The mean and covariance of the distribution with the same elliptical form is given by the following equation:

$$\begin{bmatrix} \mu_{\theta_c^i} \\ \mu_{\theta_d^i} \end{bmatrix} = \begin{bmatrix} \frac{a_{11}a_{01} - 2a_{02}a_{10}}{4a_{20}a_{02} - a_{11}^2} \\ \frac{a_{11}a_{10} - 2a_{20}a_{01}}{4a_{20}a_{02} - a_{11}^2} \end{bmatrix} \tag{5.14}$$

$$\Sigma^{-1}_{\theta_c^i \theta_d^i} = \begin{bmatrix} a_{20} & \frac{1}{2}a_{11} \\ \frac{1}{2}a_{11} & a_{02} \end{bmatrix} \tag{5.15}$$

For every new time sample, we compute coefficients a_{nm} of Equation 5.14. These coefficients determine the updated value of $\mu_{\theta_c^i}$, $\mu_{\theta_d^i}$ and $\Sigma^i_{\theta_c, \theta_d}$.

5.3.3. INITIALIZE MACHINE SLOWDOWN PARAMETERS

The initial values of machine slowdown are estimated by executing probe applications offline. Since the time we measure is the only dimension with fixed units, the value of the parameters is undetermined. We determine the parameters of the system by choosing a unit map task to define a baseline. The unit map task has an empty map function, and it does not read or write from or to HDFS.

The compute and disk task requirements, θ_c and θ_d respectively, are both zero; therefore, Equation 5.10 allows us to estimate Ω. Multiple executions are averaged to create an accurate point estimate. Note that Ω includes some computation and disk I/O that occur during start up.

One could imagine attempting to isolate the remaining parameters in the same fashion, however, it is difficult to construct an application with zero computation or zero disk I/O. Instead, we construct applications with two different levels of resource usage defined by a fixed ratio η.

Let's assume we aim to determine γ_c. First, we run an application $J_c^1 = \langle \theta_c, \theta_d \rangle$ with fixed disk requirement θ_d (J_c^1 might be a application which simply reads an input file and

processes the text in the file). We compute the average execution time of this application on each machine. According to our task model the average mapping time for every machine j can be given as follows:

$$T^{1,j} = \theta_c \gamma_c^j + \theta_d \gamma_d^j + \Omega^j \tag{5.16}$$

Next, we run an application J_c^η, which reads the same input, but the processing is multiplied by η compared to J_c^1. Therefore, the resource requirements of J_c^η can be given as $J_c^\eta = \langle \eta\theta_c, \theta_d \rangle$. The average mapping time for every machine can be given as follows:

$$T^{\eta,j} = \eta\theta_c \gamma_c^j + \theta_d \gamma_d^j + \Omega^j \tag{5.17}$$

We solve for $\theta_d \gamma_d$ in Equations 5.16 and 5.17, set them equal, and solve for γ_c^j to get the initial slowdown value:

$$\gamma_c^j = \frac{T^{\eta,j} - T^{1,j}}{\theta_c(\eta - 1)} \tag{5.18}$$

This equation gives us γ_c^j in terms of a ratio. To make it absolute, we arbitrarily choose one machine as the reference machine. We set $\gamma_c^1 = 1$ and $\gamma_d^1 = 1$ and then solve Equation 5.18 for θ_c. Once we have the task requirements θ_c in terms of the base units for machine 1, we can use this application requirement to solve for the machine slowdown on all the other machines. Similarly, we estimate γ_d. To avoid network communication while learning machine slowdowns, we set the number of reducers to zero and set the replication factor to one. Table 5.1 gives an example of computed machine slowdown parameters for an eight-machine cluster of heterogeneous machines.

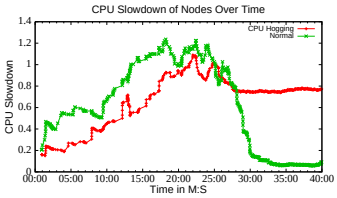
machine	γ_c	γ_d	Ω
machine1	1	1	45
machine2	1	1	45
machine3	0.1	0.33	5.3
machine4	0.1	0.33	5.3
machine5	0.1	0.25	4.8
machine6	0.1	0.33	5.3
machine7	0.1	0.33	5.3
machine8	0.1	0.33	5.3

Table 5.1: Machine Slowdown and Overhead

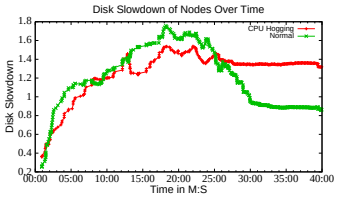
5.3.4. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our monitoring tool, we execute two MapReduce jobs, `Pi` and `TestDFSIO`, on the eight-node Hadoop cluster while continuously estimating γ_c and γ_d for every node. `Pi` calculates digits of Pi and starts 2000 mapping tasks on the cluster. `TestDFSIO` writes 4 TB of files on the nodes of the cluster. The nodes in the cluster are heterogeneous in their hardware configuration since these machines were purchased at different times and for different original purposes. This heterogeneity is reflected in Table 5.1.

We conduct two experiments to demonstrate detection of CPU contention and disk I/O contention. First, we illustrate the nominal case. At the time zero, we start both the

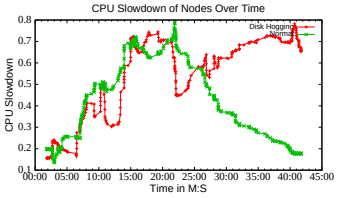


(a) Slowdown in CPU (γ_c) with CPU hogging

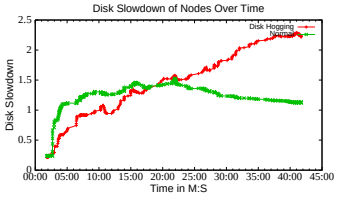


(b) Slowdown in Disk (γ_d) with CPU hogging

Figure 5.6: CPU hogging is injected in machine7. The slowdowns for γ_c and γ_d are plotted as the normal condition (green) and CPU hogging (red). Job `TestDFSIO` finishes at 25:00.



(a) Slowdown in CPU (γ_c) with Disk hogging



(b) Slowdown in Disk (γ_d) with Disk hogging

Figure 5.7: Disk hogging is injected in machine3. The slowdowns for γ_c and γ_d are plotted as the normal condition (green) and Disk hogging (Red). Job `TestDFSIO` finishes at 25:00

MapReduce jobs `Pi` and `TestDFSIO`. In Figures, 5.6(a) and 5.6(b), and 5.7(a) and 5.7(b), the green line shows the increase in CPU usage. The line increases slowly as it takes a number of updates before the priors on the γ parameters are overcome by the data. At time 25:00, `TestDFSIO` completes and CPU slowdown falls.

To demonstrate detection of CPU contention, we repeat the experiment but introduce a third CPU hogging task on machine7 at time 10:00. Red lines in Figure 5.6(a), demonstrate that the monitoring system can detect the presence of a CPU hogging job. The CPU saturates and does not show additional load until `TestDFSIO` completes at time 25:00. Unlike the nominal case, the CPU continues to be loaded by the hogging task. A similar effect can be seen for disk I/O in Figure 5.6(b)), however, the effect on disk I/O is smaller as the problem is due to CPU hogging, not disk I/O.

In Figure 5.7(a) and Figure 5.7(b), we can see that the monitoring system has more trouble isolating disk hogging processes. We start with the nominal behavior shown by the green line. Again, we start the `Pi` and `TestDFSIO` jobs. Once the `TestDFSIO` job completes at time 25:00, the CPU behavior returns to low levels.

To demonstrate the detection of disk hogging, we repeat the experiment but introduce a disk hogging task on machine3 at time 10:00. Again, due to resource saturation, we do not see an immediate effect. However, once `TestDFSIO` completes at time 25:00, we see that both the CPU and disks stay busy. While we can see that there is a fault

present, we cannot clearly distinguish between a CPU and disk fault in this case. There was no impact of CPU or disk I/O hogging on the slowdown on machines other than machine7 or machine3 for these experiments. All the experiments were repeated multiple times, and in each run same results were obtained. Therefore, we select results from one run to describe in this section.

5.4. CONCLUSIONS

We have presented two approaches for diagnosing performance issues in heterogeneous Hadoop clusters. The approaches extend the existing diagnosis approaches for Hadoop clusters where the peer-similarity assumption does not hold. Further, the diagnosis is extended to distinguish between different types of faults. In our first approach, we build a probabilistic model of task completion time of applications on every machine in the cluster. To find the find machines, we compare the current performance of applications against their learnt performance.

To empirically validate our diagnosis approach, we simulated soft faults in a Hadoop cluster consisting of 14 machines, running two different MapReduce jobs with different resource profiles. The preliminary results presented in this chapter suggest that the proposed approach is viable and able to achieve the intended goals of a) identifying machines on which resource contention is occurring, and b) determining the resource which is over-subscribed by considering the relative impact of faults on the completion time for jobs with different resource requirements. This approach has two drawbacks. First, it's an offline approach, and therefore, it can result in a lower production. Second, the approach assumes that resource profiles of applications are known, and this assumption might not be always true.

Therefore, we present a continuous state estimation approach for heterogeneous Hadoop clusters to detect performance issues on server nodes. This approach works in real time and does not require any specification of task requirements or server performance, but learns these parameters automatically by exploiting heterogeneity in the cluster. Using our monitoring tool, a system administrator can not only discover underperforming machines, but can also infer which resource in the node is lowering the performance. The chapter presents a novel and simple iterative heuristic to approximate the joint distribution in terms of marginals.

To empirically validate our diagnosis approach, we simulated soft faults in a Hadoop cluster consisting of eight nodes, running two different MapReduce jobs with different resource profiles simultaneously. The preliminary results presented in this chapter suggest that the proposed approach is viable and able to achieve the intended goals of a) identifying machines on which intermittent resource contention is occurring, and b) determining the resource which is over-subscribed by considering the relative impact of faults on the completion time for jobs with different resource requirements. While our demonstration uses the Hadoop system, the our approach is applicable to other frameworks of distributed computing as well.

6

A Pervasive Approach to Scheduler Design

For high-throughput systems, such as Hadoop, an efficient scheduler is critical to maintain high production. In the previous chapter, we described the importance of efficient schedulers and showed the performance gains from our Hadoop cluster, where the scheduler either maximizes throughput or minimizes task completion time. To achieve the best possible production from a cluster, the available resources need to be exploited efficiently. In order to use the resources efficiently, a few characteristics of the resource must be known to the scheduler, such as how many resources are available, how busy they are, and how well they are performing. We define the features of machines, such as how well each resource is performing, as the state of a machine. The scheduler uses the current beliefs about the state of machines for an optimal task assignment.

In realistic applications, there are several factors that might hamper a precise estimation of the state of a machine. For example, in a cluster consisting of several machines, a machine might either be a physical machine or a virtual machine (VM) running on an unknown physical machine. Limping hardware, unspecified background processes, or poor task scheduling results in overloaded machines [90]. These problems create resource congestions such as CPU or I/O or network congestion, resulting in the lower performance of one or more resources. In the case of Virtual Machines (VMs), this problem is magnified because they are dependent on a physical machine shared with other competing applications that may degrade their performance. Moreover, VM migrations may also result in unexpected and abrupt performance changes. This volatile performance makes it very challenging for the scheduler to precisely estimate machines' state.

Along with the precise estimation task, it is also essential for a scheduler to be completely autonomous in order to make clusters self-adaptive. An efficient self-adaptive scheduler automatically detects the changes in states and updates them. As soon as changes in a machine's performance are detected, a common response at the data center is that the machine stops running the production workloads, and a software agent is started (automatically or manually) that runs a diagnostic workload on the machine.

Even though this approach estimates new states very precisely, it comes at a price. Stopping machines can have serious negative impact on the overall cluster production. Therefore, we need a framework that updates the beliefs about the machine state and also maintains a certain production level. To this end, we apply Pervasive Diagnosis.

Pervasive Diagnosis [25] is a framework that identifies faulty components during production without necessarily stopping production. It enables higher production by scheduling production workload in systems to gain maximum diagnostic information. In Pervasive Diagnosis, if the scheduler is uncertain whether or not a component is faulty, it generates a production plan that uses such a suspicious component. Depending on the outcome of an execution plan (pass/fail), the state of the components is inferred. Pervasive Diagnosis obtains a higher long-run productivity than a decoupled combination of production and diagnosis. Unfortunately, this framework has been only studied and developed for binary systems, where a component can be either faulty or healthy and plans can either pass or fail. In Chapter 5, we discussed a methodology to estimate the state of machine capabilities by running actual production jobs on the cluster. This approach maximizes information gain to efficiently learn the machine capabilities. However, the approach does not guarantee that a certain performance will be maintained during the estimation of states.

In this chapter, our primary goal is to develop a framework for schedulers that automatically generates policies that maximizes the production. We assume that a system can be in *steady* mode, where the system states remain steady, or the system can be in *uncertain* mode, where system states are varying. In the steady mode, the productive scheduling policies are generated using the methodologies we derived in Chapter 4. In the uncertain mode, we extend the Pervasive Diagnosis framework to the continuous domain to estimate the state of machines without stopping them from production. States of a machine are modeled as continuous variables because machine states represent the effective capabilities of the machines. Therefore, in our work we model machine states as random variables. Unlike Pervasive Diagnosis, a machine (component) may have more than one state variable. Every machine has a state variable corresponding to every resource in the machine. In this chapter, we consider CPU speed and I/O bandwidth as our state variables per machine. If we schedule a workload on a machine and observe a slow performance, then we cannot precisely say which resource is actually causing the slowdown. Therefore, the problem of estimating state is much more challenging than binary Pervasive Diagnosis.

To estimate the state of a system, Pervasive Diagnosis implements a diagnostic policy that maximizes information gain. However, the most informative policy might not be very productive. Systems such as Hadoop are developed for large volume production. Therefore, in our framework, to estimate the states we implement a policy that has maximum expected production and also maximizes future production gain. Expected production implies how much production workload is finished while estimating the states. Future production depends on how system states are estimated, because a more accurate estimate implies that the future scheduling policy will more optimally assign workloads to machines. To select such a policy in a decision theoretic setting, we present a cost function in terms of instant expected production gain from running the policy and expected future production after running the policy. Such a production deci-

sion theoretical framework is useful for systems such as Hadoop because if the system is in steady state where the system dynamics are not changing, then the framework would always run the most productive policy. In the case where there is uncertainty about the system state, then framework runs policies that will give us enough information about the system to maximize the future production. As part of our contributions, we introduce a decision theoretic support to the Pervasive Diagnosis framework to make it more efficient. We empirically evaluate the performance of our framework under steady conditions and uncertain conditions. In both the cases, our framework enables maximum production from the cluster. Under the steady conditions, we observe throughput improvement up to 18 percent. Under varying conditions, we gain throughput by seven times compared to a scheduler that doesn't update the beliefs in the machine states.

6.1. OPTIMIZATION FRAMEWORK

As we stated earlier, we assume that the system can either be in a steady mode or uncertain mode. During the production, the system can transition between the stable mode and the uncertain mode. At any time, an efficient scheduler will always keep the production maximum, whether the system is stable or uncertain. In our decision theoretic setting, a scheduler generates policies to maximize production. Here we treat production as our reward function. Let's assume for cluster $\mathscr{C}$, the system states are denoted by Π and the policy is represented by $\mathbf{X}$. State of each machine in the cluster, in terms of the performance of machines is interpreted as the state of the cluster. A policy decides that how many parallel containers should be assigned to an application on a machine. An optimal scheduling policy can be formalized as:

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmax}} \mathscr{R}(\Pi, \mathbf{X}) \tag{6.1}$$

Here, $\mathscr{R}$ denotes a reward function, which represents the productivity obtained by executing policy $\mathbf{X}$ on a cluster that has state Π. It must be noted that the reward function has both the cluster state and policy as input. If the cluster states are changing, the scheduler will automatically take these changes into account, and therefore, it will continuously generate policies to maximize the reward (production). It is computationally intensive to search an optimal policy; therefore to reduce the overhead, the optimal policies are only generated when changes in system states are observed.

Pervasive Diagnosis has addressed this type of problem. In the uncertain mode, Pervasive Diagnosis uses information criteria to generate policies that reveal maximum information about the system. However, there is no guarantee that the most informative policy is also the most productive policy. It might happen that the productivity significantly goes down by executing the informative policy. Moreover, it cannot be guaranteed that after running the informative policy the system will be productive enough to compensate for the possible production lost by running the informative policy. We argue that for systems, such as Hadoop, the productivity is the most critical metric. Lost production can seriously harm business revenues. Therefore, we develop a reward based decision theoretic framework that maintains the system production at the maximum possible level all the time. We aim to achieve this goal by deriving a reward function $\mathscr{R}(\Pi, \mathbf{X})$ that can be used to generate policies that always maximize production. To de-

fine, such a reward function, we need a formulation to estimate the production of the cluster.

6.1.1. PRODUCTION METRIC AND STATE ESTIMATION

In this section, we provide a formulation to estimate the expected value of production. Subsequently, we provide the methodology to update the cluster state. Expected value of production, and state estimation are used to formulate $\mathscr{R}(\Pi, \mathbf{X})$.

EXPECTED PRODUCTION

Production metric measures the expected performance of the system under a certain policy and for the given state of the system. We use throughput as our metric to measure the productivity of the cluster. In Chapter 4, we have defined the throughput. In this chapter, we will be using the same definition of throughput, which is given as:

$$\bar{\tau}(\Pi, \mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{l} \frac{\lambda_j \cdot x_{ij}}{t_{ij}^{\Pi,\mathbf{X}}} \tag{6.2}$$

As we described in Chapter 4, λ^j denotes the size of application A^j, and x_{ij} denotes the number of containers assigned to A^j on machine i. The value $t_{ij}^{\Pi,\mathbf{X}}$ denotes the average task completion time of A^j on machine i, for a given cluster state, Π, and a given policy, $\mathbf{X}$.

The dynamic model, proposed in Chapter 4, $f(\theta^j, \Pi^i, L^i, \lambda^j)$, is used to predict the average task completion time. Therefore, the task completion time can be estimated as the following:

$$t_{ij}^{\Pi,\mathbf{X}} = f(\theta^j, \Pi^i, L^i(\mathbf{X}), \lambda^j) \tag{6.3}$$

Here, θ^j denotes the resource requirements of application A^j, and Π^i denotes the state of machine M^i in terms of its resource capabilities. $L^i(\mathbf{X})$ represents the load on machine M^i which is running the policy, $\mathbf{X}$. Hence, Equation 6.2 can be expressed as:

$$\bar{\tau}(\Pi, \mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{l} \frac{\lambda_j \cdot x_{ij}}{f(\theta^j, \Pi^i, L^i(\mathbf{X}), \lambda^j)} \tag{6.4}$$

Here, $\bar{\tau}(\Pi, \mathbf{X})$ is defined as the production utility function. The expected production is given by:

$$E\big[\bar{\tau}(\Pi, \mathbf{X})\big] = \int_{\Pi} \bar{\tau}(\Pi, \mathbf{X})\, p(\Pi)\, d\Pi \tag{6.5}$$

Here, $p(\Pi)$ denotes the belief about the machine states of the cluster.

STATE ESTIMATION

Once a cluster starts running tasks using a policy, their completion times are used as observations to update the machine state beliefs. In our formulation, Π^t is a set of state variables at time t, which are unobservable. We use $\mathbf{O^t}$ to denote set of observations variables at time t. The task completion time is treated as the observation. A migration

event is responsible for abrupt state change of the cluster. If a cluster consists of VMs, then migration of a VM from one physical host to another might drastically change the state of the VM. Variable m^t is a boolean variable which denotes whether or not a migration event occurs at time t. The following Baysian formulation is used to determine the probability distribution of Π^{t+i+1}, which the state of the cluster after $t+i+1$ steps in future:

$$\begin{aligned} & P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) \\ = & \int_{\Pi^t \cdots \Pi^{t+i}} \prod_{j=0}^{j=i} P(\Pi^{t+j+1} \mid \Pi^{t+j}, m^{t+j}) P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) d\Pi^t \cdots d\Pi^{\mathbf{t+i}} \end{aligned} \tag{6.6}$$

To compute the above predictive distribution is calculated by the posterior distribution, $P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})$ and the Markov predictive model, $P(\Pi^{t+j+1} \mid \Pi^{t+j}, m^{t+j})$. To obtain the predictive model, we use the first order Markov assumption that the next state only depends on the previous state. Therefore we use,

$$\Pi^{t+i+1} = \Pi^{t+i} + \mathcal{V}_{m^{t+i}} \tag{6.7}$$

$\mathcal{V}_{m^{t+i}}$ is a vector which denotes the noise in the above Markov model. The noise models abrupt or gradual changes in the system. In case of abrupt changes $\mathcal{V}_{m^{t+i}}$, will have high values and therefore, the weight of older observations will be negligible compared to new observation in the state estimation shown in Eq 6.6. The lower value of $\mathcal{V}_{m^{t+i}}$ implies gradual changes and, in that case, all observations will be treated equal. We use the following to determine the noise at time $t+i$:

$$\mathcal{V}_{m^{t+i}} = \begin{cases} \mathcal{V}^{low}, & \text{if } m^{t+i} = 0 \\ \mathcal{V}^{high}, & \text{if } m^{t+i} = 1 \end{cases} \tag{6.8}$$

Here, $m^{t+i} = 1$ denotes that the VM has migrated to another physical host, and $m^{t+i} = 0$ denotes that the VM is still running on the same physical host. The value for m^{t+i} can be determined by the cluster migration policy. For our experiments we choose $\mathcal{V}_{high} = 1$ and $\mathcal{V}_{low} = 0.001$.

6.1.2. SEARCH OF MOST PRODUCTIVE POLICY

After deriving the methodologies to compute the expected production and the probability distribution over system states from observations, we now develop the reinforcement learning [91] based framework to search for the most productive policy. A policy will be "most productive" if it maximizes the expected production. Earlier we defined the reward function $\mathscr{R}$ that is used by the scheduler in search the most productive policy (Eq. 6.9). If the system is in a steady state (no migration), the $\mathscr{R}$ is equal to $\bar{\tau}(\Pi, \mathbf{X})$, which is estimated in Eq. 6.4. Hence, the search for a productive policy can be given as:

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmax}} E\left[\bar{\tau}(\Pi, \mathbf{X})\right] \tag{6.9}$$

When a system is in steady state and we precisely know the states, the scheduler finds the policy that maximizes expected production, which is throughput is our case. However, when there is uncertainty in the states of the system, then using the same machine state belief to find an optimal policy might harm the production. This is because it is likely that the actual state of the system might be different from the scheduler's beliefs about the system state. Hence, the system states need to be estimated precisely. We aim to use the Pervasive Diagnosis approach to estimate the system state. Unlike the traditional Pervasive Diagnosis approach, our framework does not use expected informative gain as the only expected reward. It also incorporates expected production in the reward. The reward should also include expected future production after estimating the system state. Let's assume that at time t, the system is in state Π^t and there is a high variance in system state belief. The scheduler wants to search for a policy $\mathbf{X}^t$ that maximizes the reward $\mathscr{R}$. Just like reinforcement learning, the reward can be broken into immediate reward, $\mathscr{R}^i$, and future reward, $\mathscr{R}^f$, which is given as the following:

$$\mathscr{R}(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) = \mathscr{R}^i(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) + \mathscr{R}^f(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) \tag{6.10}$$

$\mathscr{R}^i(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})$ is the immediate reward for executing a policy $\mathbf{X}^t$, given the past observations, $\mathbf{O}^{1:t-1}$, past migration events, $m^{1:t-1}$ and past policies $\mathbf{X}^{1:t-1}$. The immediate reward can be expressed as the following:

$$\mathscr{R}^i(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) = \underset{P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}{E\left[\bar{\tau}(\Pi^t, \mathbf{X}^t)\right]} \tag{6.11}$$

The immediate reward is determined by calculating the the immediate throughput, $E\left[\bar{\tau}(\Pi^t, \mathbf{X}^t)\right]$, which is gained by running policy, $\mathbf{X}^t$, under the cluster state, Π^t, and the probability distribution of the cluster state is determined by the posterior distribution, $P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})$.

Let $\mathscr{R}^f(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})$ be the total expected future reward for running new jobs after updating the model from observations. We approximate the future stream of value by assuming the future returns stay constant until a regime change event such as process migration or environment change. For each future state, Π^{t+i+1}, we search a policy, $\mathbf{X}^{t+i+1}$, that maximizes the future reward function. As shown in Equation 6.13, we use a discount factor γ to represent the probability of this change occurring at any time step.

$$\mathscr{R}^f(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) = \sum_{i=0}^{\infty} \gamma^i \underset{\substack{\mathbf{X}^{t+i+1} \\ P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}}{E\left[\operatorname{argmax} \bar{\tau}(\Pi^{t+i+1}, \mathbf{X}^{t+i+1})\right]} \tag{6.12}$$

To solve this reinforcement learning problem, we prepare a heuristic simplification assumption that the reward remains constant into the future. We assume that rewards for each step after $t+1$ in the future will be equal to the $t+1$ step. Hence, the overall future reward is given by the following:

$$\mathscr{R}^f(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) = \frac{1}{1-\gamma} E\Big[\underset{\substack{\mathbf{X}^{t+1} \\ P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}}{\operatorname{argmax}} \bar{\tau}(\Pi^{t+1}, \mathbf{X}^{t+1})\Big] \tag{6.13}$$

Therefore, total expected reward can be given as:

$$\mathscr{R}(\mathbf{X}^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1}) = \underset{P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}{E\big[\bar{\tau}(\Pi^t, \mathbf{X}^t)\big]} + \frac{1}{1-\gamma} E\Big[\underset{\substack{\mathbf{X}^{t+1} \\ P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}}{\operatorname{argmax}} \bar{\tau}(\Pi^{t+1}, \mathbf{X}^{t+1})\Big] \tag{6.14}$$

Hence, the most optimal policy at time, t, can be obtained from the following:

$$\mathbf{X}^{t*} = \underset{\mathbf{X}^t}{\operatorname{argmax}} \left(\underset{P(\Pi^t \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}{E\big[\bar{\tau}(\Pi^t, \mathbf{X}^t)\big]} + \frac{1}{1-\gamma} E\Big[\underset{\substack{\mathbf{X}^{t+1} \\ P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})}}{\operatorname{argmax}} \bar{\tau}(\Pi^{t+1}, \mathbf{X}^{t+1})\Big] \right) \tag{6.15}$$

Here, $P(\Pi^{t+i+1} \mid \mathbf{O}^{1:t-1}, m^{1:t-1}, \mathbf{X}^{1:t-1})$ can be evaluated using Equation 6.6. To evaluate the expression derived in Equation 6.15, multiple nested samplers need to be implemented. Also, for every sample, we will have to search for the policy $\mathbf{X}^{t+1}$ that maximizes the expected throughput. Implementing such a compute intensive sampler in the scheduler can significantly reduce the overall performance of the system.

We therefore approximate the utility gained by using an improved model in the future by a linear multiple β of the information gain associated with the experimental policy $\mathbf{X}^t$, as shown in Equation 6.16:

$$\mathscr{R}^f(\Pi^t, \mathbf{X}^t) = \beta \cdot IG(\mathbf{X}^t \mid \Pi^t) \tag{6.16}$$

Let's assume that U is a utility function which estimates the information gain. Information theory tells us that the expected KL divergence can be used to determine the information gain [81]. Therefore, we select the expected KL divergence as the utility function, which is given as the following:

$$\begin{aligned} U(\mathbf{O}^t, \mathbf{m}^t, \mathbf{X}^t) &= \int_{\mathbf{O}} D_{KL}\Big[p(\Pi^{\mathbf{t+1}} \mid \mathbf{O}^t, \mathbf{m}^t, \mathbf{X}^t) \,||\, p(\Pi^t)\Big]\, d\mathbf{O}^t \\ &= \int_{\mathbf{O}} \int_{\Pi^{t+1}} p(\Pi^{\mathbf{t+1}} \mid \mathbf{O}^t, \mathbf{m}^t, \mathbf{X}^t) \\ &\qquad \ln \frac{p(\Pi^{\mathbf{t+1}} \mid \mathbf{O}^t, \mathbf{m}^t, \mathbf{X}^t)}{p(\Pi^t)}\, d\Pi^{\mathbf{t+1}}\; p(\mathbf{O}^t \mid \mathbf{X}^t, \mathbf{m}^t)\, d\mathbf{O} \end{aligned} \tag{6.17}$$

A sampling based solution [80] is implemented to estimate the above expression. This policy is suboptimal as it may learn more than is necessary to make effective decisions, but should optimize the schedule to learn about the parameters we are most uncertain about. We propose to empirically learn the value of β. However, for time being, we use $\beta = 0$ and $\beta = 1$ for our experiments. In the future, we would like to develop an empirical methodology to learn β.

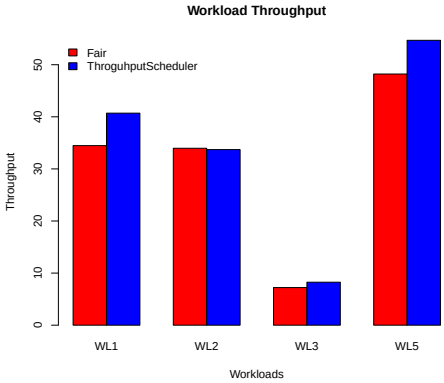


Figure 6.1: Comparison of ThroughputScheduler with Capacity Scheduler in terms of throughput. We observe throughput improvement up to 18 perecent under steady mode.

6.2. EMPIRICAL RESULTS

To empirically demonstrate the performance of our framework, we conduct experiments on our six-node Hadoop cluster. We implement our framework in the Hadoop scheduler, and we call it *ThroughputScheduler*. Each node has 8 physical CPU cores, 12 GB of RAM, and runs CentOS 5.6. We use Hadoop benchmark examples as workloads. The primary goal of our framework is to maximize total production (throughput) under stable and uncertain states. Therefore, we run experiments to demonstrate the effectiveness of our framework in stable state and uncertain state. All the experiments were repeated multiple times, and in each run same results were obtained. Therefore, we select results from one run to describe in the following subsections.

6.2.1. PERFORMANCE UNDER STABLE STATE

To evaluate the performance of ThroughputScheduler in the stable state, we run various Hadoop workloads and measure cluster throughput for each workload. Since it is common to submit multiple workloads in parallel, we also run workloads in combinations and measure throughput for each combination. We compare the performance of ThroughputScheduler against state-of-art CapacityScheduler [37] and FairScheduler [38].

The results show that ThroughputScheduler delivers higher throughput than Fair and Capacity schedulers by assigning tasks to servers that maximize the production.

6.2.2. PERFORMANCE UNDER UNCERTAINTY

To evaluate the performance of ThroughputScheduler under uncertainty, we shut down a few CPU cores of a server while running the workload. Shutting down cores during the workload execution also simulates the VM migration effect. We run this experiment using ThroughputScheduler, with and without the Pervasive Diagnosis framework. Without the Pervasive Diagnosis framework, ThroughputScheduler assumes no changes in

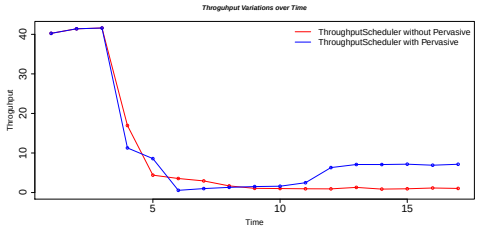


Figure 6.2: Performance of ThroughputScheduler with and without Pervasive Diagnosis framework. $\beta = 0$ is used for this experiment under an uncertain model. Time on X axis is in terms of minutes.

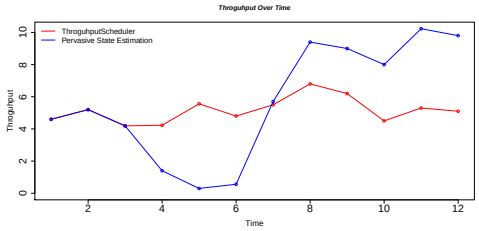


Figure 6.3: Performance of ThroughputScheduler with and without Pervasive Diagnosis framework. $\beta = 1$ is used for this experiment. Time on X axis is in terms of minutes.

the believed machine states. Therefore, it uses the initial beliefs about the states during the entire execution of workloads. On the other hand, with the framework, ThroughputScheduler estimates the new belief states of machines as soon as a change is detected and derives a policy that maximizes the throughput for new beliefs.

Figure 6.2 shows that the throughput drops from 40 to around 1 due to core shutdown. Without the Pervasive Diagnosis state estimation framework, ThroughputScheduler keeps running at low throughput. On the other hand, with the Pervasive Diagnosis framework, the throughput goes down in beginning but the schedulers update the beliefs of machine states and raises the throughput almost 7 times. It must be noted that we achieved such a throughput gain for a certain application and under a certain setting. At the same time, it should also be noted that the Hadoop clusters are long running systems, and therefore 7 times gain for one case shows the potential benefits of our approach.

In the second experiment, we do not inject any artificial slowdown in any of the servers. While running the workloads on the cluster, the scheduler detects that one of the servers has lower throughput than other servers. This is a suspicious observation because it is believed that all the servers were identical. Less throughput indicates that the server state is different from other servers. The server might be performing slower to process this kind of workload or it has some hardware issues or it might be running some unknown processes. To get more information about the server, ThroughputScheduler with Pervasive Diagnosis runs the workloads according to an informative policy by setting $\beta = 1$. Results of the experiment are shown in Figure 6.3.

Results show that with the Pervasive Diagnosis framework, throughput of the server

goes down as it runs informative tasks to determine the updated beliefs of machine states. However, after the update, the throughput increases and gets better than ThroughputScheduler without Pervasive Diagnosis.

In both the experiments under uncertainty, we demonstrate the performance of our framework on only one server rather than the entire cluster. We must recall that throughput of the cluster is a sum of throughputs of all the servers. Therefore, any improvement in one server's throughput directly improves the throughput of the cluster.

6.3. CONCLUSIONS

We extended a Pervasive approach for estimating system machine state beliefs to continuous domains using a Pervasive Diagnosis framework. To estimate the system belief in machine states, our framework uses a decision theoretic approach to determine an optimal policy that maximizes immediate and future production, weighing the cost in production of diagnostic workloads against future gains. We derive an objective function to implement the decision theoretic scheme. Whether there is uncertainty about the system machine state beliefs, or the beliefs are known precisely, the framework will generate optimal policies that maximizes the production.

Unlike Pervasive Diagnosis, our framework can handle systems that are described by multiple, continuous state variables. We implemented pervasive state estimation for the problem of scheduling in Hadoop to maximize the production of Hadoop clusters. We define throughput as a metric to measure cluster production. For Hadoop clusters, the system CPU congestion and I/O bandwidth of the servers are used as state variables. Empirically, we confirm that our extended Hadoop scheduler improves the cluster throughput by up to 18 perecent under stable conditions. More importantly, we also demonstrate that when server capabilities change, our framework successfully detects the change, updates its belief by running informative workloads, and as a result can provide throughput that can be seven times higher compared to a scheduler that does not update its belief of machine states.

7

CONCLUSION

Hadoop is one the most famous tools used to run the big data applications. The applications are written in the form of MapReduce programs, and they work on the data that is stored in a Hadoop Distributed File System (HDFS). From a revenue point of view, it is important to efficiently run applications. The performance of those applications on a Hadoop cluster depends on how efficiently cluster resources are being utilized. Efficient use of cluster implies better throughput, improved SLAs, and improved (gained) capacity from the cluster. Gaining extra capacity provides the ability to run more applications on the given cluster. Therefore, this dissertation addresses the problem of designing the Cluster Management System (CMS) for Hadoop clusters to use resources more efficiently. In the previous chapters, we mention that our CMS design has two primary components: *scheduler* and *fault monitor*. We propose two designs of schedulers, and both of them are resource-aware schedulers. The monitoring unit not only monitors the performance of machines, but it also monitors the resource-specific performance of every machine. In this thesis, our work revolves around these two components, scheduling and monitoring. In each chapter, we address the issues related to these components. Our contributions in the thesis can summarized as follows:

- Generally, CMS for various distributed systems assume a homogeneous cluster where machines are identical to each other in terms of their resource capabilities. The homogeneity assumption makes the CMS's job easier in terms of scheduling and monitoring. For instance, if all machines are identical then simple scheduling policies, even round-robin, can be used for scheduling. However, the homogeneity assumption is not true in practice. Machines in the clusters are collected from different generations, and therefore clusters naturally become heterogeneous. Additionally, the performance of machines will degrade over the time, and performance-different machines will degrade differently. In such heterogeneous cluster environment, scheduling and monitoring becomes challenging. Therefore, it makes sense to enable CMS to handle heterogeneous clusters as well. We study the problem of implementing the scheduler and monitoring tool for clusters

that can be both homogeneous and heterogeneous. To extend CMS capabilities to heterogeneous clusters, we include the resource capabilities (CPU cycles, disk I/O and memory) of machines in the CMS implementation.

- To efficiently implement the CMS scheduler, resource requirements of workloads need to be known. Workloads are submitted to clusters by users, and the submitted workloads are in the form of executable files. Therefore, in practice, the resource requirements of every new incoming workload is unknown in advance. We also do not expect that the programmers will provide the resource usage of applications while submitting them. Profiling tools can be used to derive these unknown resource requirements. Profiling tools run workloads in certain environments to estimate the resource requirements of workloads. The profiling tools are practically useful, however, in order to use them, the applications need to be executed in an isolated environment. While the applications are running in the environment, they cannot be executed on the actual production cluster. Therefore, using profiling tools can severely reduce the cluster productivity. In this thesis, we aim to derive learning approaches, which determine the resource usage of applications without stopping the production. Our first approach exploits the resource heterogeneity to learn the resource requirements. The approach executes the applications' tasks on the actual heterogeneous production cluster, and the observed task completion times are used to estimate the resource requirements. The second approach simply measures the resource usage of machines while running tasks on them to profile the applications. Neither of these approaches stop production at any point for the learning.

- To efficiently assign workloads to machines, the scheduler must also know the resource-specific capability of every machine. Additionally, determining the machine resource capabilities is important to monitoring the performance of the cluster. We refer these capabilities as machine states. Such machine states can change suddenly due to the faults in the machines, or over time, the state can change gradually due to performance degradation. Executing workloads of a certain kind also impact the machine state dynamically. For example, if there are CPU-intensive workloads on a machine, then the effective CPU capability of a machine will be lower. The existing CMS tools can only find out if a machine is performing better or slower than other machines. Those tools fail to capture resource-specific performance states, therefore they can not exactly identify if a slowdown in a machine is because of CPU capability or disk I/O. Those tools also can not capture the dynamic machine states. Therefore, in this thesis, we address these issues by proposing tools that can estimate resource-specific dynamic machine states. Just like learning about the resource requirements, the method to learn the machine capabilities also does not require machines to stop production. The production tasks of different resource requirements are executed on machines, and based on the task completion times, the machine capabilities per resource is estimated.

- In this thesis, both the learning modules to determine resource requirements and machine capabilities are implemented in the Hadoop scheduler. As the final contribution of this thesis, we propose a scheduler design that runs in two modes,

learning and optimal scheduling. The learning mode, which is also known as explore, includes learning resource requirements and machine capabilities. After learning the parameters with a certain precision, the scheduler switches to the optimal scheduling mode, which also known as the exploit. The scheduler has different objective functions in each mode, and generates scheduling policies that maximizes a specific objective function. In the explore mode, the information gain about the parameters is the objective function. In the exploit phase, the productivity gain is the objective function. In this work, we define productivity in terms of either task completion time or throughput. Due to two different definitions of productivity, we propose two schedulers in this thesis. The first scheduler, ThroughputScheduler, tries to minimize task completion time in heterogeneous Hadoop clusters by running tasks on machines that can most efficiently satisfy the tasks' resource requirements. ThroughputScheduler has a drawback that it does not consider real-time load on various resources of machines while making the scheduling decisions. To overcome this limitation, we propose DARA scheduler, which maximizes the throughput of the clusters, and also considers real-time loads while generating the scheduling plans.

- It must be noted that the scheduler never stops production — either it's in explore phase or exploit phase. However, it might very well happen that in exploit phase the production is lower, because the most informative scheduling policy might not be the most productive policy. In this thesis, the ultimate goal of the CMS is to maximize the overall productivity. Therefore, the scheduler automatically decides how much to learn such that the overall production is maximized. If the parameters are not learnt precisely, then in the exploit phase the production might not be the most optimal. On the other hand, if the scheduler spends lots of time in learning, then the overall production might be lower. Hence, the scheduler should find a balance between the explore and exploit phase. To find the balance, the scheduler uses the Pervasive Diagnosis based approach, where we define cluster productivity in terms of throughput and information gain.

7.1. FUTURE WORK

Inspired by the contributions we made in this thesis, there are many interesting open problems that are worth exploring. In the following, we suggest several recommendations for the future work:

- The proposed schedulers, ThroughputScheduler and DARA scheduler use two performance models, static model and load-dynamic model, to generate the scheduling policies. The static model is a process-based model and the load dynamic model is a data-driven. In this thesis, both of these models are not very accurate, and we do not provide a detailed discussion about the errors in the models. To make sure that the scheduler is running fast, our goal was to derive an analytical expression for the objective function that can be easily optimized. Therefore, we did not use any complicated models that could improve the accuracy. However, we believe that a more accurate model will improve performance of the scheduler

in terms of the cluster productivity. In the future, it would be interesting to develop more accurate models to implement the scheduler. It might not be possible to derive the analytical expression for the objective functions with more accurate models, which can complicate the scheduler implementation. We recommend that sampling-based approaches can be employed in such scenarios.

- In this thesis, we always use the synthetic workload for experimental evaluations, because we do not have any production workload. The synthetic workload is generated using the Hadoop benchmark applications. Although these benchmark applications cover a broader range of applications, it would be very useful to know how much improvement is observed on the production workload.

- Apart from the non-production workload, the other limitation with our experimental setup is the smaller cluster size. Due to limited resources, we used a cluster size of up to 6-8 machines for the experimental evaluations. However, a typical Hadoop cluster has hundreds to thousands of machines. The proposed CMS design extends the existing Hadoop schedulers, which maximizes performance of each machine by estimating how many tasks to run on each machine. In practice, most of the cluster machines are similar to each other. Even in heterogeneous clusters, there will be some sub-groups of homogeneous machines. Therefore, the task assignment calculation results for one machine can be reused by all the other similar machines. In theory, our CMS design can be scaled up to any number of machines, as long as Hadoop native schedulers are scalable. However, we do not have any experimental evidence to support that argument. It would be very interesting to evaluate the performance of our CMS on bigger clusters.

- In our work, we compare the performance of our schedulers against the existing Hadoop schedulers, FairScheduler and CapacityScheduler. These two schedulers are part of the open source Hadoop distribution and is widely used in production by various organizations. Hadoop is an open source project, and contributes are continuously improving the various aspects of Hadoop, including scheduling. Most of the improvements in Hadoop scheduling policies are already part of these two schedulers. Therefore, comparing the performance of our schedulers against these schedulers implies the comparison against the latest Hadoop version. However, there are other open source projects which emerged from Hadoop, such as Spark, which has its own scheduling and resource management modules. In the future, it would be interesting to compare performance of our CMS with other Hadoop related projects, such as Spark.

- This dissertation proposes a CMS solution for Hadoop, but the design methods developed in this work can be extended to other distributed-computing frameworks as well. In future work, we would like to improve the CMS aspects of distributed systems, such as Grid Computing using our CMS methodologies.

SUMMARY

In recent years, we have seen a major shift in computing systems: data volumes are growing very fast, but hardware capabilities to store, process, and transfer the massive data are not speeding up at the same rate. Today, data are generated from a variety of sources, such as social networking websites, business transactions, banking sectors, etc. These data are valuable and contain lots of vital information if they are analyzed efficiently. The processing capabilities of single machines, however, are not sufficient enough, which makes it harder to use them for data analysis. As a result, most web companies, but also the traditional business organizations, research labs, and universities, are scaling out their major computational frameworks to clusters of thousands of machines. To find the hidden and interesting insights from the data, in addition to simple queries, also complex machine learning algorithms and graphs processing are becoming a common choice in many areas. Nowadays, the problem to collect, store and analyze these data is called the *Big Data* problem.

Due to various hardware limitations, the traditional large scale distributed computing platforms, such as SQL, grid computing, and volunteer computing are not suited for big data. To solve the big data problem, Hadoop has emerged as the most useful framework. Hadoop Distributed File System (HDFS) is used to store a tremendous amount of data in cluster machines in a distributed manner. Once the data is stored in HDFS, the MapReduce framework is used to analyze the data. Programs to analyze data are written in Java based MapReduce framework and to execute on the Hadoop clusters. To reduce the network bottleneck, Hadoop sends computation to data, rather than the traditional approach where data is sent to machine to process.

To ensure the better performance, it is important for Hadoop clusters to have an efficient scheduler component and fault resilient component. These components can be collectively defined as the Cluster Management System (CMS) of Hadoop. Although many organizations have widely used Hadoop, its current CMS is still not efficient enough to optimally run applications on Hadoop. For example, it fails to identify the resources required by the applications and resource capabilities provided by the cluster while distributing applications on machines. Therefore, it might happen that resources are used sub-optimally, which leads to an overall lower productivity of the cluster. Additionally, the heartbeat protocol used by Hadoop to identify faulty machines can only detect machines that have stopped working. The protocol cannot detect faults in the machines that only might slow down the performance of machines. Such faults include hogging of resources and limping hardware. Moreover, Hadoop assumes that the cluster is homogeneous, which means that cluster machines are identical regarding their resources. However, in practice, often Hadoop clusters are quite heterogeneous.

In this thesis, we study the challenges with the design of the existing Hadoop CMS, specifically concerning resource management. We propose a CMS design to utilize the cluster resources efficiently which optimally runs applications on the cluster. The two

important features of our CMS design are the autonomous behavior and self-adaptation. The intuition behind these features is to ensure a high productivity of the cluster. The cluster should be able to adapt automatically to the dynamic behaviors such as the changing resource requirements of applications and the varying capabilities of the machine resources. To implement the mentioned features, we design two main components of CMS: a scheduler and a fault monitor. The goal of the scheduler is to run applications on machines that can most efficiently satisfy the applications' resource requirements. To accomplish this goal, the application resource requirements need to be learned automatically and without causing the severe production loss. Therefore, our scheduler autonomously determines the resource requirements in an online way by scheduling the actual applications on the machines, such that the information gain about the resource requirements is maximum. We use two approaches to learning the resource requirements. The first approach uses the resource heterogeneity present in the cluster to implement the learning. The second approach learns the resource requirements by measuring the resource utilization of machines while running actual workload. Once the resource requirements are learned, the scheduler runs applications on machines such that the throughput of the cluster is maximum.

Once the applications are up and running efficiently, the other challenge for the CMS is to maintain the productivity of the cluster under the performance problems in machines. These performance problems can cause lower the machine performance, which leads to the lower production. The monitoring module of our CMS continuously estimates the state of each machine in the cluster. The state of a machine is characterized in terms of the performance of each resource of the machine, such as CPU, I/O, and memory. The machine states are estimated without stopping the cluster production by running the actual production workloads in a way such that cluster states can be learned. Once the cluster states are determined, they can be directly used by the cluster administrators to identify the problems in the cluster. Additionally, the information about the machine states can be provided as an input to the scheduler such that applications can be distributed efficiently to machines under the dynamic behaviors of machines. Unlike Hadoop's heartbeat protocol based fault detection mechanism, our monitoring tool successfully detects slowdowns in machines, both heterogeneous and homogenous.

We have implemented our CMS by extending the Hadoop scheduler and compared the performance of our scheduler with existing Hadoop schedulers. Our results demonstrate that the proposed CMS improves the performance in terms of throughput and application completion time. The proposed CMS improves performance of both homogeneous and heterogeneous clusters. Moreover, the CMS successfully uncovers the performance problems in the Hadoop cluster in real time. Such kinds of performance problems could not be detected by native Hadoop heartbeat based fault detection. In this work, we concentrated the CMS design for Hadoop; however, the proposed concepts and theories are not only limited to Hadoop, and can easily be extended to any distributed computing platform.

SAMENVATTING

In de afgelopen jaren was er een belangrijke verschuiving in computersystemen te zien: data volumes groeien zeer snel, terwijl de hardware-mogelijkheden om grote hoeveelheden gegevens op te slaan, te verwerken en te verplaatsen niet in het zelfde tempo meegroeien. Vandaag de dag worden gegevens gegenereerd uit verschillende bronnen, zoals social networking websites, zakelijke transacties, het bankwezen, et cetera. Deze gegevens zijn waardevol en bevatten veel essentiële informatie als ze efficiënt worden geanalyseerd. De verwerkingsmogelijkheden van afzonderlijke machines zijn echter niet voldoende, waardoor het moeilijker is om de gegevens te gebruiken voor gegevensanalyse. Als gevolg daarvan schalen de meeste webbedrijven—maar ook traditionele bedrijven, onderzoekslaboratoria en universiteiten—hun rekencapaciteit op tot clusters van duizenden machines. Om verborgen en interessante inzichten uit gegevens te halen, niet alleen eenvoudige antwoorden, worden complexe machine learning en graafalgoritmen steeds meer toegepast. Tegenwoordig wordt het probleem hoe deze gegevens te verzamelen, op te slaan en te analyseren het *Big Data* probleem genoemd.

Als gevolg van verschillende beperkingen van de hardware, zijn de traditionele large scale distributed computing-platforms, zoals SQL, grid computing en volunteer computing niet geschikt voor Big Data. In plaats daarvan is Hadoop het meest bruikbare platform gebleken. Het Hadoop Distributed File System (HDFS) wordt gebruikt om een enorme hoeveelheid data op een gedistribueerde manier in clustermachines op te slaan. Zodra de gegevens zijn opgeslagen in HDFS, wordt het op Java gebaseerde MapReduce-framework gebruikt om de gegevens te analyseren op de Hadoop-clusters. Om de belasting voor het netwerk te verminderen, stuurt Hadoop de berekening naar de gegevens in plaats van de traditionele aanpak waarbij de gegevens naar de machine worden verzonden ter verwerking.

Om de betere prestaties te garanderen, is het belangrijk dat Hadoop-clusters een efficiënte planningscomponent en foutherstellingscomponent hebben. Deze componenten kunnen gezamenlijk worden gedefinieerd als het Cluster Management System (CMS) van Hadoop. Alhoewel veel organisaties al Hadoop op grote schaal gebruiken, is het huidige CMS van Hadoop nog niet efficiënt genoeg om applicaties optimaal te laten lopen. Zo is het niet in staat om bij het verdelen van applicaties te bepalen welke middelen de applicaties nodig hebben, en welke beschikbaar zijn op het cluster. Daardoor kan het gebeuren dat de middelen suboptimaal worden ingezet, wat leidt tot een lagere totale productiviteit van het cluster. Daarnaast kan het hartslagprotocol, dat wordt gebruikt door Hadoop om defecte machines te identificeren, alleen machines opsporen die volledig zijn gestopt met werken. Het protocol kan geen fouten in de machines op te sporen die enkel de prestaties van de machine verminderen. Bovendien neemt Hadoop aan dat het cluster homogeen is, met dezelfde hoeveelheid middelen voor alle clustermachines. In de praktijk zijn er echter vaak grote verschillen tussen machines in hetzelfde Hadoop-cluster.

In dit proefschrift bestuderen we de nadelen van het ontwerp van het bestaande Hadoop CMS, in het bijzonder met betrekking tot resource management. We stellen een CMS-ontwerp voor dat applicaties op het cluster kan draaien met optimaal gebruik van de beschikbare middelen. De twee belangrijkste kenmerken van ons CMS-ontwerp zijn autonoom gedrag en zelf-adaptatie. De intuïtie achter deze kenmerken is het waarborgen van een hoge productiviteit van het cluster. Het cluster moet zich automatisch kunnen aanpassen aan het dynamische gedrag, zoals de benodigde middelen van applicaties en de variërende capaciteiten van de clustermachines. Om de genoemde functies te implementeren, ontwerpen we twee belangrijke onderdelen van een CMS: een planner en een storingsmonitor. Het doel van de planner is om applicaties te laten draaien op de machines die het meest efficiënt de beschikbare middelen ter beschikking kunnen stellen. Om dit doel te bereiken, moeten de benodigde middelen automatisch ingeschat kunnen worden, zonder dat daarbij ernstig productiviteitsverlies worden geleden. Daarom bepaalt onze planner 'online' de benodigde middelen, door te applicaties zo op machines te plannen dat de informatiewinst over de benodigde middelen maximaal is. We maken gebruik van twee benaderingen voor het leren van de benodigde middelen. De eerste benadering maakt gebruik van de heterogeniteit in het cluster om de benodigde middelen te achterhalen. De tweede benadering leert de benodigde middelen door de benutting van middelen te meten terwijl de applicatie wordt uitgevoerd. Zodra de benodigde middelen bekend zijn, voert de planner de applicaties zodanig uit dat de doorvoer van het cluster maximaal is.

Zodra de applicaties efficiënt werken, is de andere uitdaging voor het CMS om bij prestatieproblemen in machines de prestatie van het cluster toch hoog te houden. De bewakingsmodule van ons CMS bepaalt continu de status van elke machine in het cluster. De toestand van een machine wordt gekenmerkt in termen van de prestaties van elke bron van de machine, zoals de processor, I/O en geheugen. De toestand van de machines wordt geschat zonder de clusterproductie te stoppen, door de normale productiewerklast zodanig uit te voeren dat de toestand van het cluster bepaald kan worden. Zodra de clustertoestanden zijn bepaald, kunnen ze direct worden gebruikt door de clusterbeheerders om de problemen in de cluster identificeren. Bovendien kan de informatie over de toestand van verschillende machines door de planner worden gebruikt om applicaties efficiënt te kunnen verdelen, gegeven het dynamische gedrag van de machines. In tegenstelling tot het hartslagprotocol van Hadoop, kan onze monitoring tool met succes vertragingen in machines detecteren, zowel heterogeen als homogeen.

We hebben ons CMS geïmplementeerd door de Hadoop-planner uit te breiden en hebben de prestaties van onze planner vergeleken met de bestaande Hadoop-planners. Onze resultaten tonen aan dat het voorgestelde CMS de prestaties verbetert, zowel wat betreft de doorvoer als de doorlooptijd van applicaties. Het voorgestelde CMS verbetert de prestaties bij zowel homogene als heterogene clusters. Bovendien is het CMS in staat gebleken succesvol de prestatieproblemen van een Hadoop-cluster in real time te identificeren. Dergelijke vormen van prestatieproblemen konden niet worden gedetecteerd door het standaard, op een hartslag gebaseerde, foutdetectiemechanisme van Hadoop. In dit werk concentreren we ons op het CMS ontwerp voor Hadoop. De voorgestelde concepten en theorieën zijn echter niet beperkt tot Hadoop, en kunnen gemakkelijk worden toegepast bij andere gedistribueerde computerplatforms.

CURRICULUM VITÆ

Shekhar GUPTA

15-07-1986 Born in Bhopal, India.

EDUCATION

2011 - Present PhD. Computer Science
Delft University of Technology
Delft, The Netherlands
Thesis: Cluster Management System Design for Big Data Infrastructures
Promotor: Prof. dr. Cees Witteveen
Copromotor: Dr. Johan De Kleer

2009–2011 Masters in Computer Engineering
Delft University of Technology
Delft, The Netherlands

2005–2009 Undergraduate in Information and Communication Technology
Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar, Gujarat, India

WORK EXPERIENCE

2015 - Present Software Engineer
Pepperdata
Cupertino, California, USA

2011 - 2015 Visiting Researcher
Palo Alto Research Center
Palo Alto, California, USA

2010-2011 Research Assistant
Delft University of Technology
Delft, The Netherlands

PUBLICATIONS

- **Shekhar Gupta**, Rui Abreu, Arjan J.C. van Gemund and Johan de Kleer. Automatic Systems Diagnosis without Behavioral Models. In Proceedings of the *IEEE*

Aerospace Conference (AEROCONF 2014). Big Sky, Montana, USA, March 2014

- **Shekhar Gupta**, Christian Fritz, Bob Price, Johan de Kleer and Cees Witteveen. Continuous State Estimation for Heterogeneous Hadoop Clusters. Accepted in the *24th Int'l Workshop on the Principles of Diagnosis (DX'13)*. Jerusalem, Israel, October 2013

- **Shekhar Gupta**, Christian Fritz, Bob Price, Roger Hoover, Johan de Kleer and Cees Witteveen. ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters. In proceedings of the*USENIX: 10th International Conference on Autonomic Computing (ICAC-2013)*. SanJose, CA, USA, June 26 - 28, 2013

- **Shekhar Gupta**, Christian Fritz, Johan de Kleer and Cees Witteveen. Diagnosing Heterogeneous Hadoop Clusters. In Proceedings of the *23rd Int'l Workshop on the Principles of Diagnosis (DX'12)*. Great Malvern, U.K, August 2012

- **Shekhar Gupta**, Bob Price, Johan de Kleer, Nico Roos and Cees Witteveen. Exploiting Shared Resource Dependencies in Spectrum Based Plan Diagnosis. In Proceedings of the *Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*. Torronto, Canada, July 2012.

- **Shekhar Gupta**, Bob Price, Johan de Kleer, Nico Roos and Cees Witteveen. Extended Spectrum Based Plan Diagnosis for Plan-Repair. In Proceedings of the *23rd International Conference on Automated Planning and Scheduling (ICAPS 2012)*. Sao Paulo, Brazil, June 2012

- Arjan J.C. van Gemund, **Shekhar Gupta** and Rui Abreu. The ANTARES Approach to Automatic Systems Diagnosis. In Proceedings of the *22nd Int'l Workshop on the Principles of Diagnosis (DX'11)*. Munich, Germany, October 2011

- **Shekhar Gupta**, Arjan J.C. van Gemund and Rui Abreu. Probabilistic Error Propagation Modeling of Logic Circuits. In Proceedings of the *1st Workshop on Testing and Debugging (TeBug)*. Berlin, Germany, March 2011.

PATENTS

- System And Method For Efficient Task Scheduling In Heterogeneous, distributed Compute Infrastructures Via Pervasive Diagnosis. Application number: 20121082US01, patent pending.

- Frugal User Engagement Help Systems. Application number: 20140034US01, patent pending.

- Dynamically Adaptive, resource aware scheduler. Application number: 20140109US01, patent pending.

- Pervasive State Estimation and an Application To Distributed Systems. Application number: 20150706US01, patent pending.

REFERENCES

[1] R. Ascierto, *Dcim market size and forecast: onward and upward,* https://451research.com/report-short?entityId=75918&referrer=marketing.

[2] *Welcome to apache hadoop,* http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] *Facebook reports first quarter 2013 results,* http://investor.fb.com/releasedetail.cfm.

[4] *Twitter usage data,* https://about.twitter.com/company.

[5] *Amazon adds 30 million customers in the past year,* http://www.geekwire.com/2014/amazon-adds-30-million-customers-past-year/.

[6] *The los angeles police department is predicting and fighting crime with big data,* https://datafloq.com/read/los-angeles-police-department-predicts-fights-crim/279.

[7] *Patterns and predictions uses big data to predict veteran suicide risk,* http://www.cloudera.com/content/cloudera/en/resources/library/casestudy/patterns-and-predictions-uses-big-data-to-predict-veteran-suicide-risk-video.html.

[8] *Ibm big data,* https://www.ibm.com/developerworks/community/blogs/ibm-big-data/entry/october_18_2011_2_41_am1?lang=en ().

[9] *Apache hadoop nextgen mapreduce (yarn),* https://hadoop.apache.org ().

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The hadoop distributed file system,* in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10 (IEEE Computer Society, Washington, DC, USA, 2010) pp. 1–10.

[11] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters,* Commun. ACM **51**, 107 (2008).

[12] *Who uses hadoop?* http://wiki.apache.org/hadoop/PoweredBy ().

[13] W. Barth, *Nagios: System and Network Monitoring*, 2nd ed. (No Starch Press, San Francisco, CA, USA, 2008).

[14] M. L. Massie, B. N. Chun, and D. E. Culler, *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience,* Parallel Computing **30** (2004).

[15] P. Panfil, *Data center downtime study shines light on best practices to eliminating unplanned outages,* http://www.continuityinsights.com/articles/2013/12/data-center-downtime-study-shines-light-best-practices-eliminating-unplanned-o

[16] *Cluster systems management,* http://www-03.ibm.com/systems/power/software/csm/ ().

[17] R. Miller, *Pepperdata scores $15m for hadoop cluster management,* http://techcrunch.com/2015/04/16/pepperdata-scores-15m-for-hadoop-cluster-performance-management/.

[18] *Benefits of pepperdata,* http://pepperdata.com/benefits/.

[19] R. Recio, *What's the big thing about open daylight?* http://www.smartercomputingblog.com/smarter-computing/open-daylight/.

[20] H. Liu, *Host server cpu utilization in amazon ec2 cloud,* https://huanliu.wordpress.com/tag/amazon-ec2/.

[21] B. Schroeder and G. A. Gibson, *Understanding failures in petascale computers,* Journal of Physics: Conference Series **78** (2007).

[22] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, *Omega: Flexible, scalable schedulers for large compute clusters,* in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13 (ACM, New York, NY, USA, 2013) pp. 351–364.

[23] Y. Luo, *Cost-Based Automatic Recovery Policy in Data Centers*, Master's thesis, University of Waterloo, Canada (2011).

[24] *Downtime, outages and failures - understanding their true costs,* http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-co
html.

[25] L. Kuhn, B. Price, M. Do, J. Liu, R. Zhou, T. Schmidt, and J. de Kleer, *Pervasive diagnosis,* Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on **40**, 932 (2010).

[26] *A relational database overview,* https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html ().

[27] *relational database management system (rdbms) definition,* http://searchsqlserver.techtarget.com/definition/relational-database-management-system.

[28] T. White, *Hadoop: The Definitive Guide*, 1st ed. (O'Reilly Media, Inc., 2009).

[29] *Overview of oracle scheduler,* http://docs.oracle.com/cd/B28359_01/server.111/b28310/schedover001.html ().

[30] *Scheduling jobs with oracle scheduler,* https://docs.oracle.com/cd/E11882_01/server.112/e25494/scheduse.htm#ADMIN034.

[31] K. F. Bart Jacob, Michael Brown and N. Trivedi, *Introduction to Grid Computing* (IBM Redbooks publication, 2005).

[32] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, *Seti@home: An experiment in public-resource computing,* Commun. ACM **45**, 56 (2002).

[33] O. Nov, D. Anderson, and O. Arazy, *Volunteer computing: A model of the factors determining contribution to community-based scientific research,* in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10 (ACM, New York, NY, USA, 2010) pp. 741–750.

[34] D. P. Anderson, *Boinc: A system for public-resource computing and storage,* in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04 (IEEE Computer Society, Washington, DC, USA, 2004) pp. 4–10.

[35] *Celebrating diversity in volunteer computing,* in *Proceedings of the 42Nd Hawaii International Conference on System Sciences*, HICSS '09 (IEEE Computer Society, Washington, DC, USA, 2009) pp. 1–8.

[36] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The google file system,* (2003).

[37] *Hadoop mapreduce next generation - capacity scheduler,* `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[38] *Hadoop mapreduce next generation - fair scheduler,* `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, *Apache hadoop yarn: Yet another resource negotiator,* in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13 (ACM, New York, NY, USA, 2013) pp. 5:1–5:16.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: Cluster computing with working sets,* in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10 (USENIX Association, Berkeley, CA, USA, 2010) pp. 10–10.

[41] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, *Apache tez: A unifying framework for modeling and building data processing applications,* in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15 (ACM, New York, NY, USA, 2015) pp. 1357–1369.

[42] *Apache slider: Dynamic yarn applications,* `http://slider.incubator.apache.org/`.

[43] B. Zhang, F. Křikava, R. Rouvoy, and L. Seinturier, *Self-balancing job parallelism and throughput in hadoop,* in *Distributed Applications and Interoperable Systems: 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, edited by M. Jelasity and E. Kalyvianaki (Springer International Publishing, Cham, 2016) pp. 129–143.

[44] C. Li, H. Zhuang, K. Lu, M. Sun, J. Zhou, D. Dai, and X. Zhou, *An adaptive auto-configuration tool for hadoop,* in *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on* (2014) pp. 69–72.

[45] R. Singhal, M. Nambiar, H. Sukhwani, and K. Trivedi, *Performability comparison of lustre and hdfs for mr applications,* in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on* (2014) pp. 51–51.

[46] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, *Resource-aware adaptive scheduling for mapreduce clusters,* in *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'11 (Springer-Verlag, Berlin, Heidelberg, 2011) pp. 187–207.

[47] P. Lu, Y. C. Lee, and A. Zomaya, *Non-intrusive slot layering in hadoop,* in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (2013) pp. 253–260.

[48] S. Tang, B.-S. Lee, and B. He, *Dynamic slot allocation technique for mapreduce clusters,* in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on* (2013) pp. 1–8.

[49] B. Sharma, R. Prabhakar, S. Lim, M. Kandemir, and C. Das, *Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters,* in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012) pp. 1–8.

[50] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, *Large-scale cluster management at Google with Borg,* in *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[51] K. A. Kumar, V. K. Konishetty, K. Voruganti, and G. V. P. Rao, *Cash: context aware scheduler for hadoop,* in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12 (ACM, New York, NY, USA, 2012) pp. 52–61.

[52] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas, *Performance impact of resource contention in multicore systems,* in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010) pp. 1–12.

[53] S. Zhuravlev, S. Blagodurov, and A. Fedorova, *Addressing shared resource contention in multicore processors via scheduling,* in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV (ACM, New York, NY, USA, 2010) pp. 129–142.

[54] S. Blagodurov, S. Zhuravlev, and A. Fedorova, *Contention-aware scheduling on multicore systems,* ACM Trans. Comput. Syst. **28**, 8:1 (2010).

[55] H. Herodotou, *Hadoop performance models,* CoRR **abs/1106.0940** (2011).

[56] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, *Improving mapreduce performance in heterogeneous environments,* in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08 (USENIX Association, Berkeley, CA, USA, 2008) pp. 29–42.

[57] H. Chang, M. S. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee, *Scheduling in mapreduce-like systems for fast completion time.* in [57], pp. 3074–3082.

[58] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, *Improving mapreduce performance in heterogeneous environments,* in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08 (USENIX Association, Berkeley, CA, USA, 2008) pp. 29–42.

[59] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, *Towards characterizing cloud backend workloads: insights from Google compute clusters,* SIGMETRICS Perform. Eval. Rev. **37**, 34 (2010).

[60] A. Khan, X. Yan, S. Tao, and N. Anerousis, *Workload characterization and prediction in the cloud: A multiple time series approach,* in *NOMS* (IEEE, 2012) pp. 1287–1294.

[61] S. Pacheco-Sanchez, G. Casale, B. W. Scotney, S. I. McClean, G. P. Parr, and S. Dawson, *Markovian workload characterization for qos prediction in the cloud,* in *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011* (2011) pp. 147–154.

[62] L. Cherkasova and P. Phaal, *Session-based admission control: A mechanism for peak load management of commercial web sites,* IEEE Trans. Computers **51**, 669 (2002).

[63] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty, *File system workload analysis for large scale scientific computing applications,* in *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (2004) pp. 139–152.

[64] S. Aggarwal, S. Phadke, and M. A. Bhandarkar, *Characterization of hadoop jobs using unsupervised learning,* in *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings* (2010) pp. 748–753.

[65] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, *Kahuna: Problem diagnosis for mareduce-based cloud computing environments,* in *IEEE/IFIP Network Operations and Management Symposium* (2010).

[66] B. J. Mathiya and V. L. Desai, *Apache hadoop yarn parameter configuration challenges and optimization,* in *Soft-Computing and Networks Security (ICSNS), 2015 International Conference on* (2015) pp. 1–6.

[67] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, *X-trace: A pervasive network tracing framework,* in *In NSDI* (2007).

[68] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer, *Pinpoint: Problem determination in large, dynamic internet services,* in *In Proc. 2002 Intl. Conf. on Dependable Systems and Networks* (2002) pp. 595–604.

[69] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, *Visual, log-based causal tracing for performance debugging of mapreduce systems,* in *ICDCS* (2010) pp. 795–806.

[70] J. Tan, X. Pan, S. Kavulya, R. G, and P. Narasimhan, *Salsa: Analyzing logs as state machines 1,* .

[71] X. Pan, J. Tan, S. Kavulya, R. G, and P. Narasimhan, *Ganesha: Black-box fault diagnosis for MapReduce systems*, Tech. Rep. (2008).

[72] A. Konwniski and M. Zahari, *Finding the Elephent in the Data Center: Tracing Hadoop*, Tech. Rep. (2008).

[73] R. Zhang, S. Moyle, S. McKeever, and A. Bivens, *Performance problem localization in self-healing, service-oriented systems using bayesian networks,* in *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07 (ACM, 2007).

[74] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, *The impact of performance asymmetry in emerging multicore architectures,* in *In Proceedings of the 32nd Annual International Symposium on Computer Architecture* (2005) pp. 506–517.

[75] S. Ghiasi, T. Keller, and F. Rawson, *Scheduling for heterogeneous processors in server systems,* in *Proceedings of the 2nd conference on Computing frontiers*, CF '05 (ACM, New York, NY, USA, 2005) pp. 199–210.

[76] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, *Heterogeneous chip multiprocessors,* Computer **38**, 32 (2005).

[77] S. Gupta, C. Fritz, R. Price, R. Hoover, J. de Kleer, and C. Witteveen, *Throughputscheduler: learning to schedule on heterogeneous hadoop clusters,* in *Proceedings of the International Conference on Autonomic Computing (ICAC '13), June 26-28, 2013, San Jose, CA USA* (2013).

[78] S. Gupta, *An optimal task assignment policy and performance diagnosis strategy for heterogeneous hadoop cluster,* in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13 (AAAI Press, 2013) pp. 1664–1665.

[79] K. Chaloner and I. Verdinelli, *Bayesian experimental design: A review,* Statistical Science **10**, pp. 273 (1995).

[80] X. Huan and Y. M. Marzouk, *Simulation-based optimal bayesian experimental design for nonlinear systems.* J. Comput. Physics **232**, 288 (2013).

[81] T. M. Cover and J. A. Thomas, *Elements of information theory* (Wiley-Interscience, New York, NY, USA, 1991).

[82] R. Larson, *Precalculus with Limits*, 2nd ed. (Cengage Learning, 2013).

[83] J. Rayces, *General equation of the ellipse,* (1999).

[84] *Package org.apache.hadoop.examples,* `https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/package-summary.html` ().

[85] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, *Mesos: A platform for fine-grained resource sharing in the data center,* in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11 (USENIX Association, Berkeley, CA, USA, 2011) pp. 22–22.

[86] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, *Dominant resource fairness: Fair allocation of multiple resource types,* in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11 (USENIX Association, Berkeley, CA, USA, 2011) pp. 24–24.

[87] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, *The case for evaluating mapreduce performance using workload suites,* in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on* (2011) pp. 390–399.

[88] S. Gupta, C. Fritz, J. D. Kleer, and C. Witteveen, *Diagnosing heterogeneous hadoop clusters,* .

[89] B. P. J. d. K. Shekhar Gupta, Christian Fritz and C. Witteveen, *Continuous state estimation for heterogeneous hadoop clusters,* in *Proceedings 24rd International Workshop on the Principles of Diagnosis: DX-2013* (2013).

[90] E. Bortnikov, A. Frank, E. Hillel, and S. Rao, *Predicting execution bottlenecks in map-reduce clusters,* in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12 (USENIX Association, Berkeley, CA, USA, 2012) pp. 18–18.

[91] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. (MIT Press, Cambridge, MA, USA, 1998).