Performance Modelling and Scheduling of MapReduce Jobs in Cloud Environment

A DISSERTATION

submitted towards the partial fulfillment of the

requirements for the award of the degree of

INTEGRATED DUAL DEGREE

in

COMPUTER SCIENCE AND ENGINEERING
(With specialization in Information Technology)

Submitted by:

VISHAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE

ROORKEE - 247667, INDIA

May 2015

Candidate's Declaration

I hereby declare that the work, which is being presented in the dissertation titled "**Performance Modelling and Scheduling of MapReduce Jobs in Cloud Environment**" towards the fulfillment of the requirement for the award of the degree of **Integrated Dual Degree in Computer Science and Engineering** submitted in the Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Roorkee Uttarakhand (India) is an authentic record of my own work carried out during the period from July, 2013 to May, 2015, under the guidance of **Dr. Manoj Mishra, Professor**, Department of Computer Science and Engineering, IIT Roorkee.

The matter presented in this dissertation has not been submitted by me for the award of any other degree of this or any other institute.

Date:
Place: Roorkee (Vishal)

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date:
Place: Roorkee (Dr. Manoj Mishra)

Professor

Department of Computer Science and Engineering

IIT Roorkee

Abstract

Many business enterprise and government organizations are increasingly using MapReduce platform to process highly unstructured data for efficient large scale data processing such as spam detection, advanced data mining, data analytics and personalized advertisements. With increasing number of users per MapReduce cluster there is a need for Service Level Objective (SLO) based resource provisioning. With its virtue of infinite computing and storage resources, Cloud computing environments are becoming increasingly famous to host MapReduce clusters. With its cheap, highly elastic performance model, cloud environment can guarantee fulfilment of SLOs.

In this thesis we address the problem performance modelling and scheduling of MapReduce jobs in cloud environment. We also argue that a single, simple performance model of jobs is sufficient to achieve all the performance goals and SLOs. We propose a job profile and performance model. Based on this performance model we propose algorithms for two types of service level objectives: makespan minimization and deadline constraint. To demonstrate the usefulness of our techniques, we apply them to real life data and perform critical analysis.

Acknowledgements

This dissertation would not have been possible without the support of several people. Many thanks to my adviser, Prof. Manoj Mishra, who gave me direction and taught me how to do research. His wisdom, knowledge and commitment to the highest standards inspired and motivated me. I would like to acknowledge the important role of my thesis evaluation committee for giving me feedback. And finally, thanks to my parents and numerous friends for always offering their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud Computing is being considered as the 5th utility after gas, electricity, water and telephony. Cloud Computing delivers software (SaaS), platform (PaaS) and infrastructure (IaaS) as services based on a "pay-as-you-go" model to consumers. Today, many large scale data processing applications such as search engines, weather prediction services, social networking, and online map services run on cloud environment. Improving the performance of information systems for analysis, computation and storage has always been a research challenge.

A cloud computing environment is composed of two entities – cloud provider and cloud user. Cloud providers provide computing infrastructure through their massive datacenters and give resources to cloud users on a pay-as-you-go model basis. Cloud users request computing, storage and network resources from providers in order to run their software with varying loads. The interactions between them is shown in Figure 1.1.

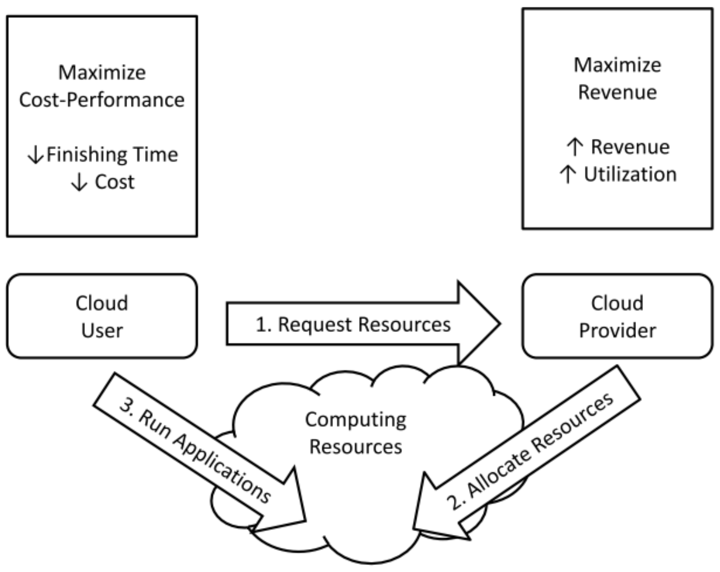

Figure 1.1 Cloud Usage Scenario

Cloud provider and cloud user are usually different parties with their own goals. Provider wants to generate maximum revenue with minimum investment and minimum failures. This involves optimum placement of Virtual Machines onto physical servers, along with taking care of VM interference problems. On the other hand, user aims at minimizing cost of operation. This involves proper resource identification (e.g. type of VMs to be leased) and effective scheduling of tasks onto those VMs.

There is no information sharing between these two parties. As a result, optimum resource allocation is very difficult to achieve. For instance, providers do not share the information about types of servers, network topology and exact locations of servers. Similarly, users do not share information about their workload, datasets. Since users do not have the complete knowledge of the available resources and resource performance, they cannot determine the resource requirement effectively. Similarly, provides do not have any knowledge about the type of application which is to be run on their servers, so they cannot multiplex their resources effectively. For instance, applications with complementary resource usage patterns can be assigned on same machine (a CPU intensive application and an I/O intensive application can be effectively multiplexed on same machine).

Performance models of jobs can be used to gain important incite of the working and critical parameters of the job. With an effective performance model, we can schedule jobs effectively and efficiently, and can ensure performance goals of the job are fulfilled. A single performance model can be used in different schedulers with different scheduling goals [1].

MapReduce is a software framework for "embarrassingly parallel and data-intensive tasks" is now used by most of the software giants such as Google, Yahoo!, and IBM [2]. It is one of the most popular frameworks running on cloud. MapReduce applications have a specific model, which makes their performance modelling easy. Furthermore, performance modelling and scheduling framework developed for MapReduce Applications can be easily extended to other data intensive frameworks.

Thus, our aim is to develop a performance model such that it encapsulates all the critical properties of jobs. This performance model can then be further used by scheduling frameworks to achieve performance goals and SLOs. We need to minimize the cost of operation for jobs running in cloud environment along with taking care of all the properties of cloud discussed above. We use MapReduce as the target application due to its simple model and large popularity.

1.2 Hypothesis

Our hypothesis is that only a single performance model of MapReduce environment running on Cloud is sufficient for enabling different service level objectives. This performance model is developed through a combination of measurement, simulation and analytical modelling, and is practical, novel and functional.

1. **Practicality**: The performance model developed for MapReduce environment is based on key performance characteristics measured from past job executions.
2. **Novelty**: The developed framework is compared against state of art frameworks, comparing and contrasting different aspects.
3. **Functionality**: The developed framework is used to achieve different service level objectives such as deadline constraint scheduling, and makespan optimization scheduling.

1.3 Broader Impact

Our performance modelling and scheduling framework can be extended to more data-intensive frameworks, such as Pig, Dryad and Hive. Our framework predicts the performance of data intensive applications in varying resource environments. It allows the users to achieve different SLOs such as meeting deadlines, minimization of makespan, and answering what-if questions. By allocating resources in a systematic and efficient way, our framework optimizes the total usage of cloud resources.

1.4 Assumptions and Limitations

"All models are wrong. Some models are useful" – George Box.

We make the following assumptions about the Cloud environment and MapReduce environment. First, we assume that the Cloud Environment is homogeneous, consisting of only one type of VM instance. Second, we assume that the network has infinite bandwidth. We do not model the network, and assume that it is not a bottleneck in the system. Third, we assume that the latency of new VM allocation is negligible. The process of request, allocation and registration of a VM instance has negligible latency as compared to running time of jobs. Fourth, we assume that time of completion of map and reduce tasks depends only upon amount of data, not type of data. Finally, we assume that the MapReduce cluster has no background load.

1.5 Problem Statement and Contribution

The main contributions of this research are –

1. Design of a performance model to profile MapReduce jobs in order to estimate the resource requirements to achieve performance goals.
2. Using the single performance model developed to achieve different service level objectives.

Our performance model allows estimation of job completion time given a set of resources, and estimation of number of resources required given a job deadline. First, we profile previous executions of the job, parse required information from the logs, and develop a performance model. Our performance model can be used to predict job performance parameters on different dataset sizes. This performance model is used by the scheduler to schedule (1) Deadline constraint jobs on adequate amount of resources, and (2) Minimizing makespan of a batch of jobs on a given set of resources. Our deadline aware scheduler uses malleable property of MapReduce jobs to minimize cost of operation while guaranteeing job completion within specified deadline. Our makespan aware scheduler minimizes makespan of a batch of MapReduce jobs using our proposed genetic search based BalancedPools algorithm.

1.6 Thesis Outline

The thesis is structured as follows: Chapter 2 provides background theory about Cloud Computing, Amazon EC2 service and MapReduce framework. It also discusses the state-of-art in performance modelling and scheduling of MapReduce jobs. The discussion of scheduling algorithms is divided into 3 categories according to objectives fulfilment – fairness, SLOs and cost minimization. In chapter 3 we discuss MapReduce job execution and job profile. We also propose a simple performance model for MapReduce jobs. Chapter 4 discusses the proposed malleability based deadline-aware scheduler. Chapter 5 introduces a novel BalancedPools with Genetic Search algorithm. Finally, Chapter 6 summarizes our work and concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we give a brief overview of our target environment, the Cloud, specifically Amazon Web Services, and the target application, MapReduce. We also discuss Amazon's spot market in detail. We also discuss related works in the field of performance modelling and job scheduling.

2.1 Background

2.1.1 Cloud Computing

The cloud computing environment refers "*to the hardware and systems software in the datacenters that provide computing resources as services*". [3]. According to National Institute of Science and Technology (NIST) Cloud Computing delivers software (SaaS), platform (PaaS) and infrastructure (IaaS) as services based on a pay-as-you-go model to consumers.

- **Software as a Service (SaaS)** – In SaaS, cloud provider provides application services to cloud user. Cloud user does not have control over cloud infrastructure.
- **Platform as a Service (PaaS)** – Cloud user has access to cloud provider's platform where it can run its own applications using the tools and APIs provided by the provider. Cloud user does not have control over underlying cloud infrastructure.
- **Infrastructure as a Service (IaaS)** – Cloud user has access to the underlying cloud infrastructure provided by cloud provider and it has control over network, storage, processing and other elements. User does not manage the underlying infrastructure.

Throughout this thesis, we refer to IaaS as the "cloud environment".

2.1.2 Amazon EC2

Amazon's Elastic Compute Cloud is an extremely popular IaaS provider. It offers a wide variety of virtual machine types with different capabilities. The instances have been classified into three categories based on the pricing:

- **On-Demand Instances** – On demand instances lets user pay on an hour basis, without any long-term commitments. These instances can be instantiated immediately as per requirements and removes the need for planning, purchasing and maintaining hardware. These instances are highly reliable.

- **Reserved Instances** – Reserved Instances lets user pay on a per year basis, with or without an upfront fee. They prove to be a cheaper option on a long term basis as compared to on-demand instances. These instances are highly reliable.
- **Spot Instances** – Spot instances allow users to bid on unused Amazon EC2 capacity and run those instances as long as their bid is greater than Spot Price. Spot price changes periodically according to supply, demand and current bids. These are highly unreliable. If the Spot Price exceeds the bid, amazon takes the VM instances back, without any notice.

Cost of operation can be minimized by **using spot-instances strategically**.

2.1.3 MapReduce

MapReduce is a software framework for parallel and data-intensive tasks. A MapReduce computation comprises of 2 functions – map and reduce. The map function takes input data and produces intermediate key/value pairs. The intermediated values associated with the same key are grouped into a single list. These intermediate key/value-list pairs are given to reduce function to produce final key/value pairs. Figure 2.1 shows a general MapReduce workflow.

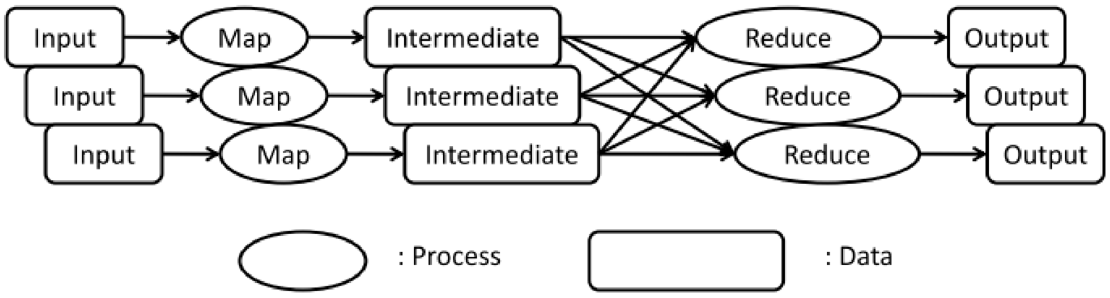

Figure 2.1 MapReduce Workflow *[4]*

MapReduce jobs are divided into 2 phases: the map phase is partitioned into a set of map tasks, and reduce phase is partitioned into a set of reduce tasks. Each map task processes a block of data. Input data is logically split into blocks of configurable sizes. The number of map tasks equals the number of blocks. Each map task takes a block of data as input, applies the user-defined map function and buffers the intermediate results. Number of reduce tasks are defined by the user. Each reduce task processes a specific set of intermediate keys, and outputs the final key/value pairs. Reduce phase consists of three sub phases – shuffle, sort and reduce. In shuffle phase, the reduce tasks fetch the intermediate data from completed map tasks, then the collected data is sorted, and finally written to disk. When intermediate data from all the map tasks is shuffled, a final pass is made to merge all the sorted files. Finally, the sorted intermediate data is given as input to the user-defined reduce function and final key/value pairs are written to the disk. Reduce phase cannot be carried out until data from all the map tasks has been received and shuffled.

There are two types of nodes in a MapReduce cluster – Master Node and Slave Nodes. Master Node performs job scheduling, and manages the slave nodes. Each slave node has a fixed number of map and reduce slots, which can run map and reduce tasks respectively. Number of slots is generally dependent upon number of cores. Conventionally, one map tasks per one core and one reduce task per two cores.

Parallel tasks can be divided into 3 categories – (1) Non-moldable, (2) Moldable, and (3) Malleable.

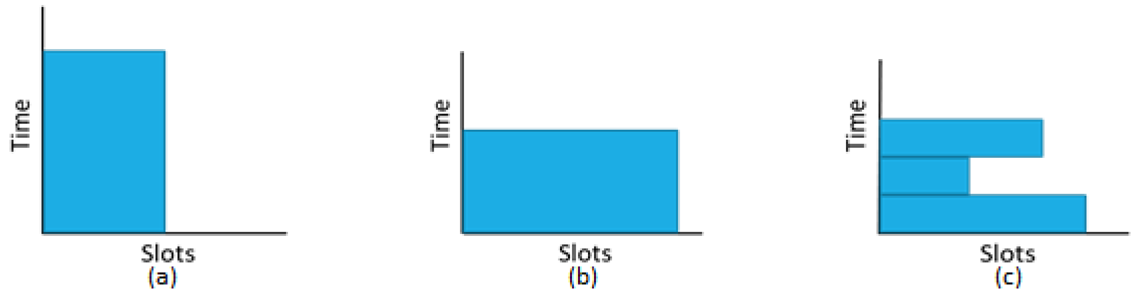

Figure 2.2 (a) MapReduce job with 5 allocated slots. (b) Moldability - MapReduce job with 10 allocated slots. (c) Malleability - MapReduce job with varying number of slots allocated with time.

A moldable job is the one which can be assigned varying number of parallel processing units, but the configuration cannot be changed over the course of its execution. The processing time depends upon the speed up function of the job. A malleable job is the one which can be assigned varying number of parallel processing units, and the configuration can be changed over the course of its execution. MapReduce jobs are malleable in nature.

2.2 Related Works

2.2.1 Performance Modelling

Performance Modelling of MapReduce jobs is required in order to estimate adequate amount of resources to complete the job with given SLOs. The correctness of job profile also determines the cost of operation. Verma et al. [2] proposed a framework for job profiling and performance modelling which can be used to estimate required resources to meet application performance goals. They also designed a model for estimating the impact of node failures on job completion time. Ganapathi et al. [5] proposed a framework for predicting the execution time of hive queries using kernel canonical correlation analysis.

2.2.2 Job Scheduling

Several different schedulers have been proposed for the MapReduce framework. The study of the state-of-art has been divided into 3 different categories depending upon the objective of the scheduler.

2.2.2.1 Fairness

Fairness measures or metrics are used in resource allocation and scheduling to determine whether users or applications are receiving a fair share of system resources. In a multiuser environment, fairness is an important measure of performance of a scheduler. By default, Hadoop (an open source implementation of MapReduce framework) comes with a FIFO scheduler which is not designed for multiuser environment. The first scheduler for Hadoop guaranteeing fairness was developed by Zaharia et al. [6] called the FAIR Scheduler. It uses a version of max-min fairness with minimum guarantees to allocate slots across pools. They also proposed an improvement on FAIR called Delay Scheduler [6], which along with fairness guarantees data locality for MapReduce jobs. Lee et al. [4] proposed a heterogeneity aware scheduler which measures fairness using the concept of "Progress Share". They also argued that using Slot Share to measure fairness, which is used in FAIR cannot be applied in heterogeneous environments. Sandholm et al. [7] proposed a Dynamic Priority parallel task scheduler which allows users to control their allocated capacity by adjusting their spending over time.

2.2.2.2 Service Level Objectives

A service level objective (SLO) is a key element of a service level agreement (SLA) between a service provider and a customer. There can be many service level objectives such as response time, availability, deadline, etc. some of which can be guaranteed by the scheduler. Wolf et al. [8] proposed a flexible scheduling scheme called FLEX, with a goal to optimize any of a variety of standard scheduling theory metrics (response time, stretch, makespan, fairness and Service Level Agreements (SLAs)). Verma et al. [1] proposed a framework for SLO-driven resource provisioning and sizing of MapReduce jobs. They proposed an automated job profiling tool, a deadline aware resource identification tool, and an SLO based scheduling algorithm. Verma et al. [9] proposed a scheduling algorithm based on a heuristic call Balanced-Pool which aims at minimizing the makespan (completion time) of a set of MapReduce jobs.

2.2.2.3 Cost Minimization

Cloud IaaS providers like Amazon Web Services (AWS) provide a variety of virtual machine instances with different performances as well as cost. Schedulers are required to make decisions in real time regarding which instance is to be instantiated and for how much time. Along with Reserved and On-Demand instances, AWS also provides cheaper, but less reliable Spot Instances which can decrease the operating cost tremendously. Early research in cost minimization revolved around minimizing the number of map and reduce slots. The more recent research advances focus on using Spot instances. Poola et al. [10] proposed a fault-tolerant job scheduling on clouds using spot instances. The proposed scheduler switched between spot and on-demand instances in order to minimize cost and ensure reliability. Navraj et al. [11] proposed a scheduler which uses spot instances in order to accelerate MapReduce jobs.

2.2.3 Research Gaps

Almost all of the MapReduce job schedulers discussed above, focus on only one of the three main objectives – fairness, SLO and cost minimization.

FAIR and Delay scheduler [6] guarantee only fairness. Dynamic Priority parallel task scheduler [7] also guarantees only fairness, however it also provides user based cost control. Heterogeneity aware scheduler [4] although provides fairness in a heterogeneous environment, it fails in the fulfilment of SLOs and cost minimization.

Both, SLO-driven scheduler [1] and Fault-tolerant job scheduler [10] satisfy deadline based SLO along with minimizing the operating cost. However, SLO-driven scheduler achieves cost minimization by minimizing number of allocated slots, while fault-tolerant scheduler uses Spot instances for the same purpose. Both can be easily extended for multiple users using FAIR scheduler at user level and combining the concept of minimum number of slots with spot instances. Spot Instance accelerated scheduler [11] uses a static set of spot instances in order to accelerate MapReduce jobs and minimize total execution time and cost of operation. A dynamically adjustable set of spot instances would be more cost efficient. Finally, FLEX [8] is the only scheduler which guarantees fairness, SLOs fulfilment and cost minimization. However, even FLEX can be improved further by using spot instances for cost minimization.

Chapter 3

Performance Modelling of MapReduce Jobs

3.1 Motivation

For many companies and organizations large scale processing of unstructured data has become a vital task. There is a steep increase in online data which has called for improved large scale data processing algorithms. Parallel and distributed computing has been widely regarded as the future of large scale data intensive computing. MapReduce and its open source implementation Hadoop is a software framework for parallel and distributed processing of large volumes of data and mining exabytes of unstructured information. Hadoop is being widely used by companies and organizations for data intensive tasks, advanced data mining and data analytics.

Originally Hadoop was developed for a single user private cluster. So, it was equipped with a simple FIFO scheduler. For a batch of jobs, for minimizing makespan FIFO scheduler is quite efficient. But, sometimes it leads to large waiting times for jobs. Once long, production jobs are scheduled in the MapReduce cluster, a short user job has to wait until the long production job finishes its execution. This makes the performance of Hadoop cluster for interactive ad-hoc queries poor. The Hadoop Fair Scheduler (HFS) [6] solves this issue by providing each user with a minimum share of MapReduce cluster, and each job a minimum share of slots. Both FIFO and HFS fail in providing any support for other Service Level Objectives and performance goals. Our goal is to develop a scheduling framework which guarantees the fulfilment of application specific goals and SLOs. In order to satisfy SLOs such as job deadline, performance modelling of MapReduce job is necessary in order to gain vital information about the job. Our aim is to develop a single, simple performance model which can be used to fulfil any performance goal and SLO.

3.2 MapReduce Job Execution and Job Profile

In this section we discuss the different executions of a MapReduce job with varying amount of resources. Our main aim is to extract a job profile which can be used to build a robust performance model of the job, encapsulating all the important performance parameters of the job during different stages of its execution.

3.2.1 MapReduce Job Execution

In this section we will discuss in detail, the entire execution process of a MapReduce job. A MapReduce job is divided into many map and reduce tasks spread across the MapReduce cluster. We are using the WordCount application as the target job. It counts the word frequencies in the given input data. First, the map splits each line into words, then reduce counts the occurrence of each word. First we consider the execution of WordCount job with 5GB of input data on 8 machines, each configured with 8 map slots and 8 reduce slots, with 64 map slots and 64 reduce slots in total.

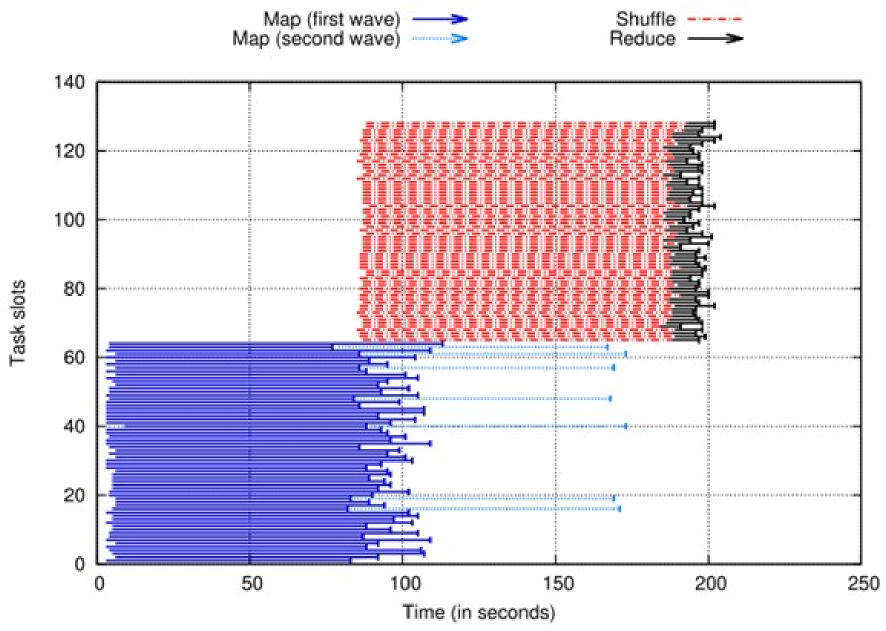

Figure 3.1 WordCount with 64 map and 64 reduce slots *[2]*

Figure 3.1 shows the progress of map tasks and reduce tasks. The configured block size is 64MB, so the number of map tasks is 5GB/64MB = 80 input splits. So for each input split there is a map task, means 80 map tasks. 80 map tasks on 64 map slots leads to 2 wave of map tasks ($ceil(80/64)$). As soon as one of the map tasks completes, reduce tasks are initiated on reduce slots. Since 64 reduce slots are running 64 reduce tasks (user configured), there is a single wave of reduce tasks. One important aspect of reduce tasks is that they do not start reduce process until the data from all the map tasks has been shuffled and sorted. Next we run the WordCount application with 5GB of data on 8 machines, each with 2 map slots and 2 reduce slots, with 16 map slots and 16 reduce slots in total.

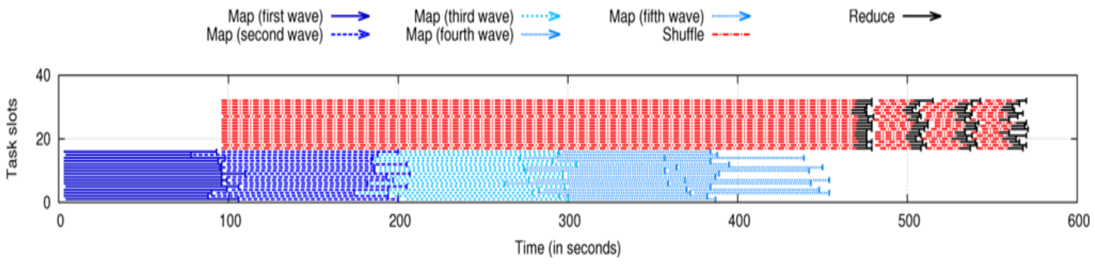

Figure 3.2 WordCount with 16 map and 16 reduce slots *[2]*

Figure 3.2 shows the progress of this job. There are 5 waves of map tasks $(ceil(80/16))$ and 4 waves of reduce tasks $(ceil\left(\frac{64}{16}\right))$.

It can be observed from Figure 3.1 and Figure 3.2 that job completion time of a MapReduce job is greatly affected by the amount of allocated resources. In next section we present a job profile which can be used for creating a performance model of the job.

3.2.2 MapReduce Job Profile

Our proposed approach is a simplified version of the one proposed in [2]. Our aim is to create a job profile which encapsulates all the important properties of a job, along with effectively summarizing the relationship between its completion time and number of allocated resources. The job profile should be performance invariant, i.e. the parameters in job profile should not change with varying resources.

A number of map tasks comprise the Map stage. Since each map tasks runs on a single slot, the completion time of single map task is independent of total number of slots assigned. The map phase job profile consists of $(Mt_{min}, Mt_{max}, Mt_{avg}, SizeMt_{input})$, where

- Mt_{min} - Minimum map task duration. It can be used to find the starting time of reduce phase, as it's the time of completion of first map task.
- Mt_{max} – Maximum map task duration. Used as an upper bound on map wave completion time.
- Mt_{avg} – Average time of map task completion.
- $SizeMt_{input}$ – Amount of input data to each map task.

Reduce phase consists of 3 steps - shuffle, sort and reduce. We are considering shuffle and sort as a single phase and call it shuffle phase. Shuffle phase starts when first few map tasks complete their executions and ends when all the map tasks finish their executions, and all the intermediate data from all the map tasks have been shuffled and sorted. Once shuffle phase is completed, reduce phase starts. The shuffle time of first reduce wave is generally much greater than the shuffle time of subsequent reduce waves (shown in Figure 3.1and Figure 3.2). So the shuffle phase job profile consists of two sets of parameters, one for first shuffle and other for remaining shuffles. For first shuffle we find the difference between end of last map and end of first shuffle wave. And for other shuffle waves, we use the average. The parameters are $(Shf_{min}, Shf_{max}, Shf_{avg}, Sho_{min}, Sho_{max}, Sho_{avg})$, where Shf_x denotes parameter for first shuffle wave and Sho_x denotes parameter for other shuffle waves.

The reduce phase begins after the completion of shuffle phase. The reduce phase job profile consists of $(Rt_{min}, Rt_{max}, Rt_{avg})$: minimum, maximum and average reduce task durations.

So $(Mt_{min}, Mt_{max}, Mt_{avg}, SizeMt_{input}, Rt_{min}, Rt_{max}, Rt_{avg}, Shf_{min}, Shf_{max}, Shf_{avg}, Sho_{min}, Sho_{max}, Sho_{avg})$ form the profile of a job.

3.3 Prediction of Job Performance Parameters on Different Datasets

Given a job profile, our aim is to predict $(Mt_{min}, Mt_{max}, Rt_{min}, Rt_{max}, Shf_{min}, Shf_{max}, Sho_{min}, Sho_{max})$ of same job on different dataset with different size.

Map tasks parameters depend on the complexity of map function and type of input. So, map tasks parameters are approximately constant across different datasets with different sizes as a map task always processes only a block of input data, irrespective of data size. So, with increasing data size total map completion time increases while map parameters(Mt_{min}, Mt_{max}) remain almost constant.

For reduce and shuffle phase, the story is a little different. The more the size of input data, more is the size of intermediate key/value pairs. This increases the number of key/values pairs per reduce task (for fixed number of reduce tasks), which further increases reduce and shuffle parameters $(Shf_{min}, Shf_{max}, Sho_{min}, Sho_{max}, Rt_{min}, Rt_{max})$. So with an increase in input dataset size, there is an increase in reduce phase completion time. So, with fixed number of reduce tasks, there is an increase in the average reduce and shuffle task completion times.

3.4 Estimating the MapReduce Job Completion Time

In this section we develop a simple model which can be used to predict completion time of MapReduce jobs as a function of allocated resources and input data size. We use an approach similar to the one proposed in [1].

Using the job profile described in 3.2.2, the total completion time of map phase of a job T_{map} is bounded on minimum and maximum map completion times (denoted as TL_{map} and TU_{map}):

$$TL_{map} = \left\lceil \frac{N_{maptasks}}{N_{mapslots}} \right\rceil * Mt_{min} \tag{3.1}$$

$$TU_{map} = \left\lceil \frac{N_{maptasks}}{N_{mapslots}} \right\rceil * Mt_{max} \tag{3.2}$$

$$T_{map} = (TL_{map} + TU_{map})/2 \tag{3.3}$$

Similarly, completion time of reduce phase of a job TU_{reduce} is the sum of shuffle and reduce times. Bounds on completion time of shuffle phase other than first shuffle are:

$$TL_{sho} = \left(\left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil - 1 \right) * Sho_{min} \tag{3.4}$$

$$TU_{sho} = \left(\left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil - 1 \right) * Sho_{max} \tag{3.5}$$

$$T_{sho} = (TL_{sho} + TU_{sho})/2 \tag{3.6}$$

Completion time for entire shuffle phase is:

$$T_{sh} = T_{sho} + Shf_{avg} \tag{3.7}$$

Similarly, completion time for reduce phase is given by:

$$TL_{reduce} = \left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil * Rt_{min} \tag{3.8}$$

$$TU_{reduce} = \left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil * Rt_{max} \tag{3.9}$$

$$T_{reduce} = (TL_{reduce} + TU_{reduce})/2 \tag{3.10}$$

Bounds on total job completion time is the sum of Map phase and Reduce phase, so it be given by:

$$TL_{job} = TL_{map} + Shf_{min} + TL_{sho} + TL_{reduce} \quad 3.11$$

$$TU_{job} = TU_{map} + Shf_{max} + TU_{sho} + TU_{reduce} \quad 3.12$$

$$T_{job} = T_{map} + T_{sh} + T_{reduce} \quad 3.13$$

$$T_{job} = \left\lceil \frac{N_{maptasks}}{N_{mapslots}} \right\rceil * \left(\frac{Mt_{min} + Mt_{\max}}{2}\right) + \left(\left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil - 1\right) * \left(\frac{Sho_{min} + Sho_{max}}{2}\right) + Shf_{avg} + \left\lceil \frac{N_{reducetasks}}{N_{reduceslots}} \right\rceil * \left(\frac{Rt_{min} + Rt_{\max}}{2}\right) \quad 3.14$$

So, simplified equation of T_{job} can be written as:

$$T_{job} = \frac{A}{N_{mapslots}} + \frac{B}{N_{reduceslots}} + C \quad 3.15$$

Where, $A = N_{maptasks} * \left(\frac{Mt_{min}+Mt_{\max}}{2}\right)$

$B = N_{reducetasks} * \left(\frac{Sho_{min}+Sho_{max}+Rt_{min}+Rt_{\max}}{2}\right)$

$C = Shf_{avg} - \left(\frac{Sho_{min}+Sho_{max}}{2}\right)$

3.5 Estimating Number of Slots for a Given Deadline

In this section, we propose an efficient way to predict the number of maps and reduce slots required in order to complete a job with a given deadline.

3.5.1 Fixed Number of Reduce Slots Method

Since, number of reduce tasks is configured in the job configuration file, and it is generally less than number of map tasks, we fix the number of reduce slots equal to number of reduce tasks. Now, we solve 3.15 to obtain the value of $N_{maptasks}$.

$$N_{mapslots} = \frac{A}{T_{job} - C - \frac{B}{N_{reduceslots}}} \quad 3.16$$

3.5.2 Minimization of Total Slots Method

Our aim is to solve 3.15 such that $N_{mapslots} + N_{reduceslots}$ is minimized. It is a constraint satisfaction problem, and can be solved using **Lagrange Multiplier,** as done in [2]**.**

We wish to minimize $f(N_{mapslots}, N_{reduceslots}) = N_{mapslots} + N_{reduceslots}$ over 3.15. Using Lagrange Multiplier, we obtain:

$$N_{mapslots} = \sqrt{A} * \left(\frac{\sqrt{A} + \sqrt{B}}{T_{job} - C}\right) \tag{3.17}$$

$$N_{reduceslots} = \sqrt{B} * \left(\frac{\sqrt{A} + \sqrt{B}}{T_{job} - C}\right) \tag{3.18}$$

These are optimal (minimum) values of map and reduce slots such that job completes within specified deadline.

3.6 Evaluation of Performance Model of MapReduce Jobs

In this section, we evaluate the proposed performance model based on the proposed job profile. We performed the experiments on a cluster of 6 nodes which configuration – Intel i7-3770 CPU, 8 cores at 3.40 GHz, with 8 GB RAM and 1 TB storage. We used Hadoop 1.0.3 with 1 machine as JobTracker and NameNode and all 6 as the slave nodes. Each slave is configured with 8 map slots and 4 reduce slots. So, the entire cluster has 48 map slots and 24 reduce slots. The file system block size is 64MB. The replication level is 2. We used 2 applications to validate our results – Anagrams and WordCount. We used two datasets D_1 and D_2: (1) 2GB Wikipedia articles, (2) 5GB of Wikipedia articles. Anagram finds all the anagrams from the given data corpus. WordCount counts the occurrence of each word from the given data corpus.

3.6.1 Stability Test of Job Profiles

First we validated the stability of proposed job profile across different datasets of different sizes across different configurations. The two datasets are denoted as D_1 and D_2. Job $D_{x_{m,r}}$ denotes xth dataset with m map slots and r reduce slots.

It can be seen from Table 3.1 that the job profile for same application job is very consistent. From this measurement, we can conclude that the proposed job profiles for same application job are consistent with different datasets of different sizes with different configurations.

Table 3.1 Map Profile of 4 jobs. A denotes Anagram and W denotes WordCount

Job/Application	**Mt$_{min}$**	**Mt$_{avg}$**	**Mt$_{max}$**
$\boldsymbol{D1_{24,24}}$/**A**	94	144	186
$\boldsymbol{D2_{24,24}}$/**A**	86	142	193
$\boldsymbol{D1_{48,24}}$/**A**	94	133	170
$\boldsymbol{D2_{48,24}}$/**A**	71	132	171
$\boldsymbol{D1_{24,24}}$/**W**	12	22	32
$\boldsymbol{D2_{24,24}}$/**W**	15	26	37
$\boldsymbol{D1_{48,24}}$/**W**	11	21	31
$\boldsymbol{D2_{48,24}}$/**W**	14	24.25	34.5

Table 3.2 Shuffle and Reduce Profiles of 4 jobs. All the jobs were configured to use 24 reduce tasks

Job/Application	**Sh$_{min}$**	**Sh$_{avg}$**	**Sh$_{max}$**	**Rt$_{min}$**	**Rt$_{avg}$**	**Rt$_{max}$**
$\boldsymbol{D1_{24,24}}$/**A**	121	136	152	16	24.5	33
$\boldsymbol{D2_{24,24}}$/**A**	222	267	312	34	52	70
$\boldsymbol{D1_{48,24}}$/**A**	122	137	152	15	26	37
$\boldsymbol{D2_{48,24}}$/**A**	223	265	306	35	50.5	66
$\boldsymbol{D1_{24,24}}$/**W**	65	77	89	16	26.5	37
$\boldsymbol{D2_{24,24}}$/**W**	126	140	154	45	67	89
$\boldsymbol{D1_{48,24}}$/**W**	66	79.5	93	13	26.5	40
$\boldsymbol{D2_{48,24}}$/**W**	131	141	151	41	63.5	86

3.6.2 Scaling factor

We used the same configuration as previous experiment, but this time we gradually increased the data size from 10MB to 7GB (Size of D_1 and $D_2 = 7GB$) and recorded the average map, shuffle and reduce times per task for both Anagram and WordCount applications. Number of map and reduce slots is 24 each. In Figure 3.3 and Figure 3.4, Avg. Map is average map time per task (Mt_{avg}), Avg. Shuffle is average shuffle time per task ($Shf_{avg} + Sho_{avg}$), and Avg. Reduce is average reduce time per task (Rt_{avg}).

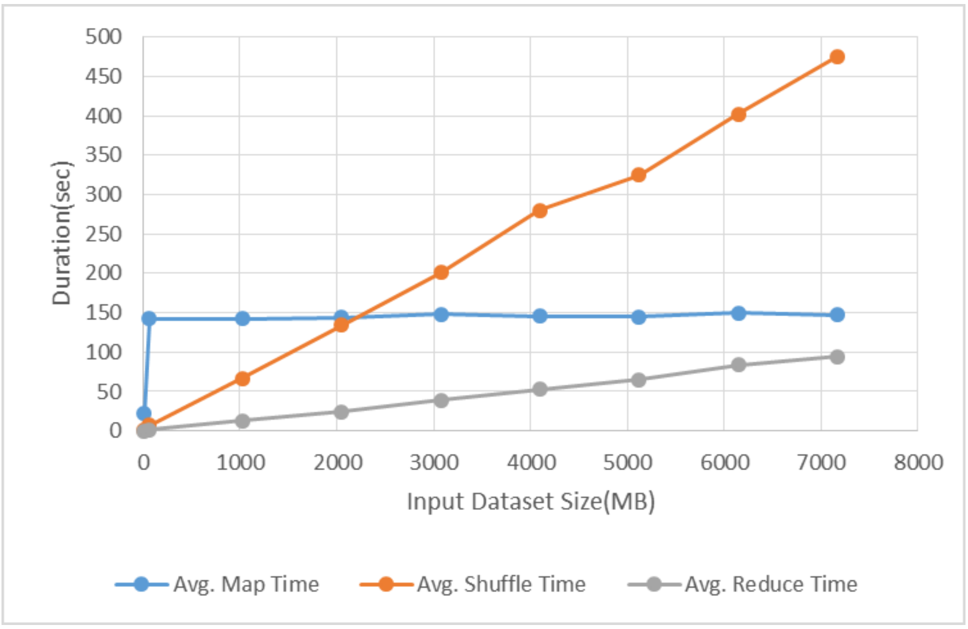

Figure 3.3 Change in map, reduce and shuffle task completion times for Anagram with different dataset sizes.

Figure 3.3 and Figure 3.4 show that the trend for average shuffle and reduce task durations is linear and duration of a map task for a fixed block size is almost constant. The initial increase in map tasks duration is due to the size of dataset being less than the block size. When the size of dataset equals the block size, average map task completion time attains its maximum value, and then remains almost constant.

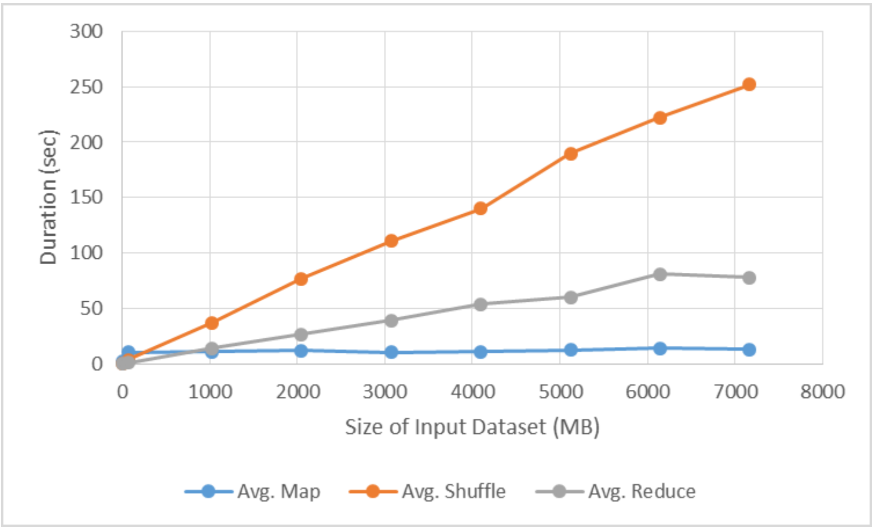

Figure 3.4 Change in map, reduce and shuffle task completion times for WordCount with different dataset sizes.

3.6.3 Prediction of Job Completion Times

We validated the process explained in 3.4 by estimating job completion time (T_{job}) for both Anagram and WordCount applications when executed on D_1 and comparing the results with practically obtained job completion times. Number of map and reduce slots is 24 each. We ran 10 instances of both applications with input dataset size randomly selected between 1GB and 2GB.

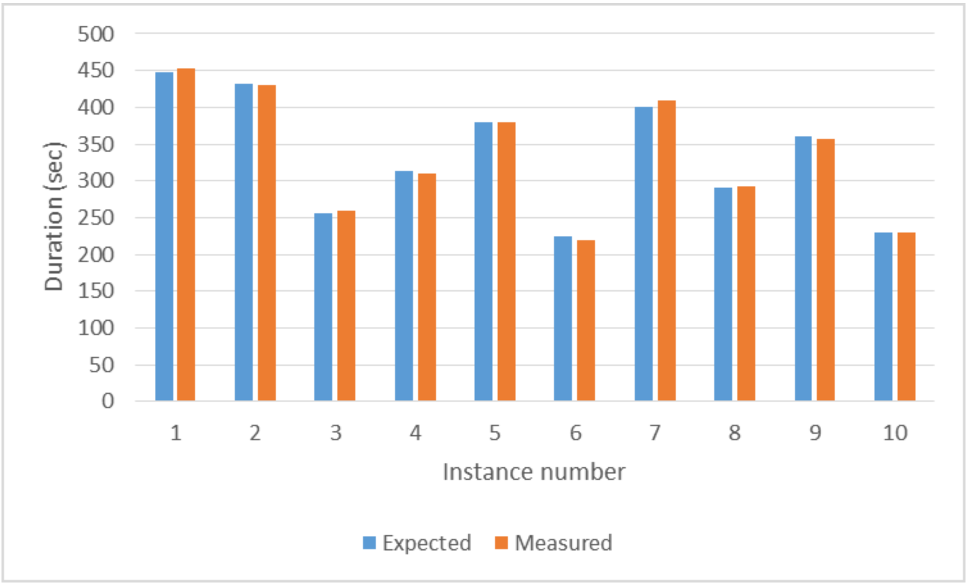

Figure 3.5 Expected and Measured job completion time for 10 Anagram jobs.

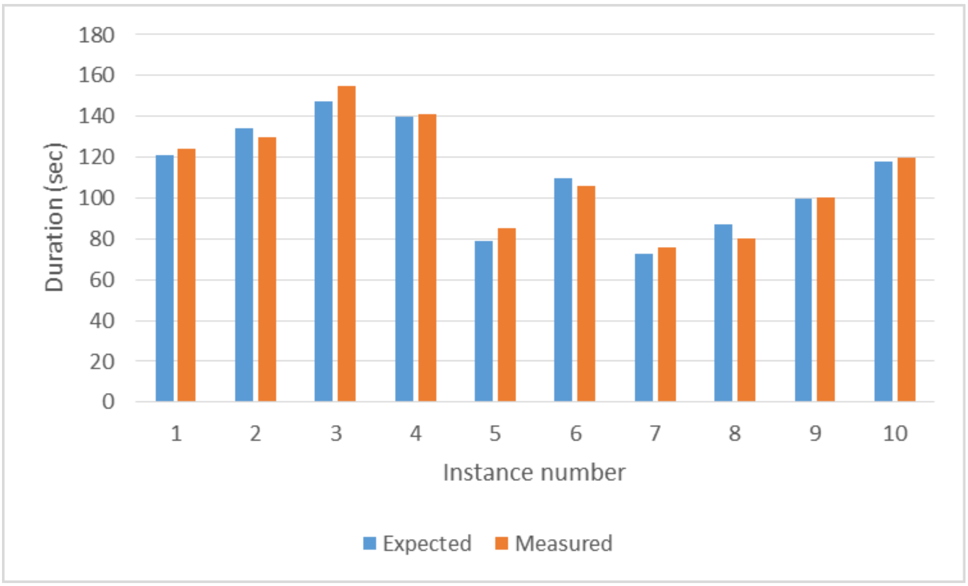

Figure 3.6 Expected and Measured job completion time for 10 WordCount jobs

It can be seen from Figure 3.5 and Figure 3.6 that the proposed job performance model correctly predicts the job completion time with error rates as high as only 0.1 per cent. The average error in prediction of job completion time for Anagram is -8.5 seconds and for WordCount is -8 seconds. This verifies that our proposed job profile can be used for job completion time prediction.

3.6.4 Prediction of Map and Reduce Slots given Job Deadline

We validated the process explained in 3.5 by estimating map and reduce slots required for Anagram application when executed on D_1 with deadline of 8 minutes. We then run the application on estimated number of slots and measure the tardiness. Maximum number of map and reduce slots is 24 each. We ran 10 instances of Anagram application with same input dataset.

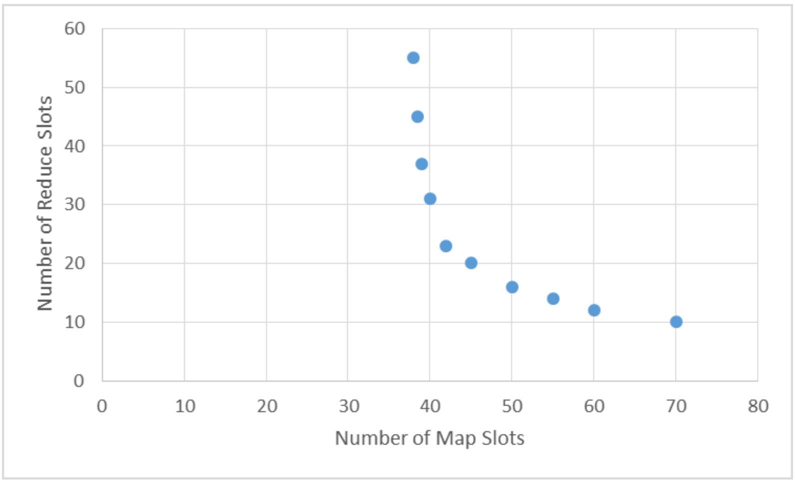

Figure 3.7 Allocation curve for Anagram with deadline 8 min and 2GB data

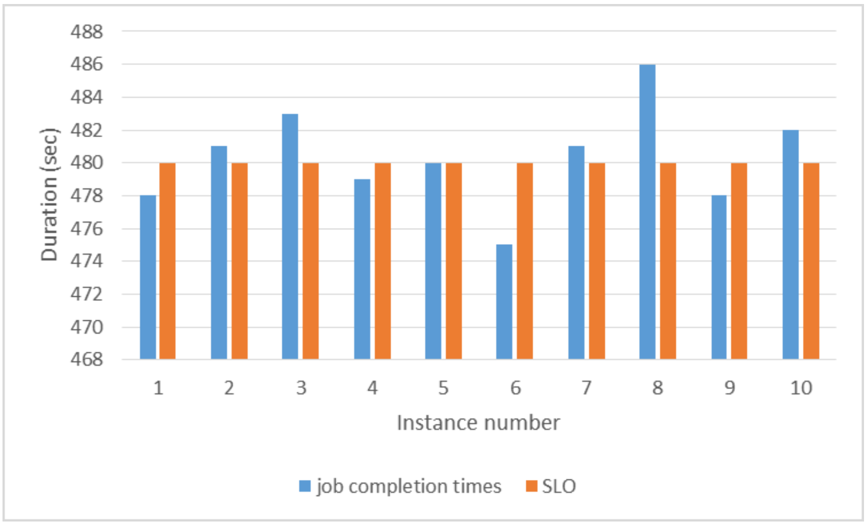

Figure 3.8 Completion times based on estimated number of slots for 10 Anagram jobs

Figure 3.7 shows the allocation curve for Anagram job with a deadline of 8 minutes. Figure 3.8 shows job completion times obtained with 42 map slots and 22 reduce slots for 10 Anagram jobs. There are 5 tardy jobs with average tardiness of 0.3 seconds. Our job profile predicts the number of map and reduce slots required to complete a job within a specified deadline with a low average error of 0.55 percent, and thus, can be used in real systems.

Chapter 4

Deadline Aware Scheduling

In this chapter we propose a novel algorithm for scheduling of MapReduce jobs with a given deadline. We use Amazon EC2 as cloud infrastructure provider. In order to minimize cost of operation we use combination of spot and on-demand instances.

4.1 Motivation

With the increase in unstructured data, enterprises are using Hadoop extensively in order to gain information from it. Companies such as Facebook, IMDb, and Google use MapReduce for data compression, i.e. converting unstructured data into structured data. These companies process petabytes of data on their MapReduce clusters for advanced data analytics, data mining, spam detection and business intelligence. Large production jobs are used for this purpose. In a multiuser environment, along with production jobs, user also submit small ad-hoc queries for getting results from processed data. These queries are generally deadline constraint. Hadoop's default FIFO scheduler and Hadoop Fair Scheduler could not guarantee the fulfilment of such SLOs and performance goals.

So, we need a scheduler which allocates resources in such way that applications performance goals and service level objectives are satisfied. Minimization of cost of operation is also vital for such a vast setup. Even a small decrease in cost of operation can prove to be extremely useful, where millions of dollar is at stake. Strategic use of spot instances can reduce as much as 10 per cent of cost of operation.

4.2 Scheduler Architecture

Figure 4.1 shows the architecture of our deadline aware scheduler. It has six interacting components:

1. **Profile Database** – Stores jobs profiles as a MySQL database. Profiles are uniquely identified by the Job name.
2. **Job Profiler** – Parses the information about new jobs and stores the required job parameters in the profile database.

3. **Slot Estimator** – Given job deadline, it predicts the number of map and reduce slots required to complete the job within the specified deadline. It can configured to use any of the 2 methods explained in 3.5 to predict the same.
4. **Instance Type Decision Maker** – Our main goal is to schedule jobs such that they complete within specified deadline at **minimum cost**. We minimize cost of operation by using 2 strategies: (1) Predicting minimum resources to achieve performance goals, (2) Using a combination of spot and on-demand instances to minimize cost. Instance type decision maker process predicts the minimum bid required for spot instances. If the bid is greater than on-demand price, on-demand instances are used, else spot instances.
5. **Slot Allocator** – This process allocates idle slots to the tasks. Using malleability of MapReduce jobs, slots running other tasks are also pre assigned. If new resources are required, this process requests Amazon EC2 service to allocate new Virtual Machines.

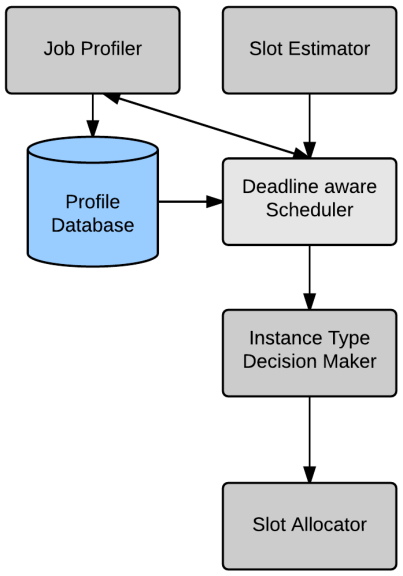

Figure 4.1 Deadline aware scheduler architecture

The Deadline-aware scheduler is –

1. **Event based**– Instead of making scheduling decisions at regular epochs, it makes decisions with the occurrence of events.

2. **Non-queue based** – A job is scheduled as soon as it arrives. It exploits the notion of infinite resources in cloud by allocating as many resources as required to a job to complete it within specified deadline.

4.3 Scheduling Algorithms

We propose 2 scheduling algorithms for the deadline-aware scheduler – (1) Naïve Deadline-Aware Scheduling Algorithm and (2) Malleability Based Deadline-Aware Scheduling Algorithm.

4.3.1 Naïve Deadline-Aware scheduling Algorithm (NDASA)

NDASA allocates slots to a job as if the job is "only moldable". So after determining the number of map and reduce slots, NDASA first, checks list of idle slots. If more slots are required it sends the request to Instance type decision maker process which determines the type of Instance to be requested (On-Demand or Spot). Then, Slot allocator process requests Amazon EC2 the desired instances. Once new instances are received, NDASA allocates new idle slots to the job. So, all the slots are allocated at the beginning of the job execution and there is no change in slot allocation of the job during the course of its execution, ensuring only Moldability.

Algorithm 1 Naïve Deadline-Aware Scheduling Algorithm (NDASA)

```
1:  When JobTracker adds j to ready queue:
2:  Job profile is fetched from profile database
3:  Slot Estimator process determines the minimum number of map and reduce slots
    ($m_j, r_j$) required using one of the 2 methods.
4:  for each slot $s$ in idle map slot list do
5:      if job $j$ has unlaunched map task t then
6:          Launch $t$ on $s$
7:      end if
8:  end for
9:  if job $j$ has unlaunched map tasks then
10:     $N_{VM} \leftarrow \frac{Number_{maptasks}}{Number_{MapslotsperVM}}$
11:     Instance Type Decision Maker identifies the type of VM to be requested as
        type $T$
```

```
12:     $N_{VM}$ VMs of type $T$ are requested from Amazon EC2 and slots are added to
        idle list.
13:     **for each** slot $s$ in idle map slot list **do**
14:         **if** job $j$ has unlaunched map task t **then**
15:             Launch $t$ on $s$
16:         **end if**
17:     **end for**
18:   **end if**
19:   wait for one of the map tasks to finish
20:   repeat steps 4 to 17 for reduce tasks
```

4.3.2 Malleability Based Deadline-Aware Scheduling Algorithm (MDASA)

MDASA allocates slots to a job by treating it as malleable. So after determining the number of map and reduce slots, MDASA first, checks list of idle slots. If more slots are required, it allocates currently running slots to the job. If the job still requires slots it sends the request to Instance type decision maker process which determines the type of Instance to be requested (On-Demand or Spot). Then, Slot allocator process requests Amazon EC2 the desired instances. Once new instances are received, MDASA allocates new idle slots to the job. So, in this case, some of the slots are allocated at the beginning of the job execution, while remaining slots are allocated during the course of its execution, ensuring malleability. Algorithm 2 gives the detailed schema of MDASA.

Algorithm 2 Malleability based Deadline-Aware Scheduling Algorithm (MDASA)

```
1:  **When JobTracker adds** j **to ready queue:**
2:  Job profile is fetched from profile database
3:  Slot Estimator process determines the minimum number of map and reduce slots
    $(m_j, r_j)$ required using one of the 2 methods.
4:  **for each** slot $s$ in idle map slot list **do**
5:      **if** job $j$ has unlaunched map task t **then**
6:          Launch $t$ on $s$
7:          set $occupancy(s) \leftarrow currenttime + Mt_{avg}$
8:      **end if**
9:  **end for**
10: **if** job $j$ has unlaunched map tasks **then**
11:     **for each** slot $s$ running tasks **do**
```

```
12:              if $endtime(j) > occupancy(s)$ then
13:                 $N_{tasks} = \left\lfloor \frac{endtime(j)-occupancy(s)}{Mt_{avg}} \right\rfloor$
14:                 schedule $N_{tasks}$ number of tasks on $s$
15:                 set $occupancy(s) \leftarrow occupancy(s) + Mt_{avg} * N_{tasks}$
16:              end if
17:          end for
18:  end if
19:  if job $j$ has unlaunched map tasks then
20:      $N_{VM} \leftarrow \frac{Number_{maptasks}}{Number_{MapslotsperVM}}$
21:  Instance Type Decision Maker identifies the type of VM to be requested as type
     $T$
22:  $N_{VM}$VMs of type $T$ are requested from Amazon EC2 and slots are added to idle
     list.
23:      for each slot $s$ in idle map slot list do
24:          if job $j$ has unlaunched map task t then
25:              Launch $t$ on $s$
26:          end if
27:      end for
28:  end if
29:  wait for one of the map tasks to finish
30:  repeat steps 4 to 28 for reduce tasks
```

Similar to NDASA, MDASA first tries to allocate idle slots to the job, shown in Lines 4-9. If the job requires more slots, running slots are checked. $\boldsymbol{occupancy(s)}$ denotes the time until slot s is occupied. $\boldsymbol{occupancy(s)}$ less than the job deadline means slot s will become idle before the expected completion time of job j. So slot s can be used to run the tasks of the job. Minimum number of tasks which can be scheduled on s is given by the expression in Line 13. In Lines 10-18, tasks are assigned on running slots. If more slots are required then new VMs are requested from Amazon EC2 which can be of type on-demand or spot as identified by the Instance Type Decision Maker process. In Lines 23-27 the new slots are assigned to the remaining map tasks. Same process is carried out for reduce tasks.

4.4 Experimental Setup and Results

The proposed scheduler architecture has been implemented in CloudSim simulator version 3.0.3. We performed simulations with two workloads denoted by W1 and W2 used in [12].

W1 is a 4 hour long Facebook Hadoop job trace consisting of 1000 jobs. W2 is a 24 hour long Facebook Hadoop job trace consisting of 5893 jobs. The datasets provide job profile parameters which are stored in job profile database.

The job deadline is set using the following process. Let T_{min} be the job completion time with single map and single reduce wave. The job deadline is uniformly distributed in the interval $[T_{min}, 3 * T_{min}]$.

For spot market price prediction, we use the model described in [13]. The price of OnDemand and Spot instance is taken from the amazon's EC2 website.

4.4.1 Based on VM allocation policy

In order to emphasize on the risk of using spot instances, we first compare the percentage of tardy jobs in W1 and W2 by using 3 VM allocation policies – (1) Only OnDemand instances, (2) Only Spot instances, and (3) Hybrid. MDASA is used as the scheduling algorithm. The Slot allocator process requests new VMs from Amazon EC2 based on the configured VM allocation policy.

Table 4.1 Percentage of tardy jobs using MDASA with different VM allocation policies

Allocation Policy/Dataset	Percentage of Tardy jobs in W1	Percentage of Tardy jobs in W2
OnDemand	4.4	4.5
Spot	49.0	48.7
Hybrid	17.7	17.3

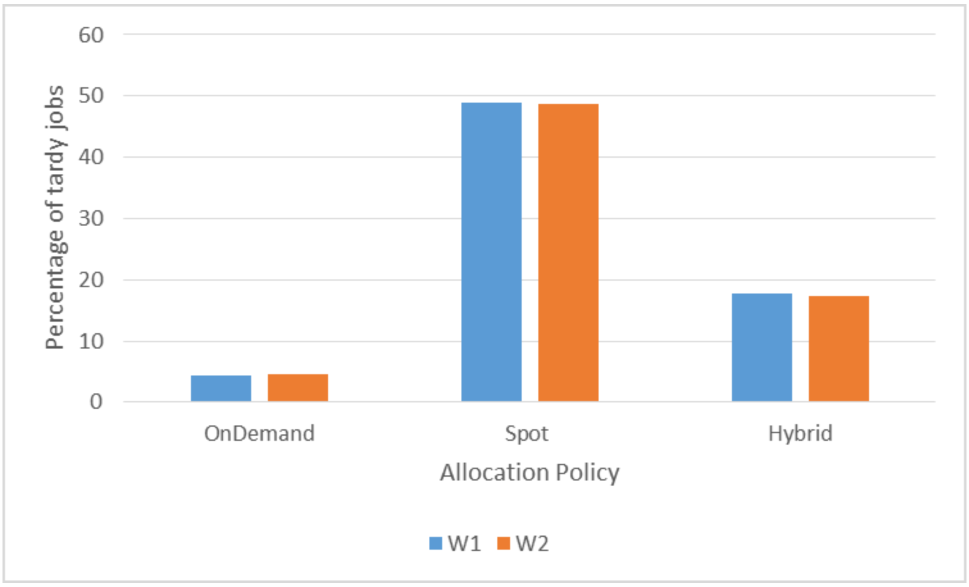

Figure 4.2 Percentage of tardy jobs in W1 and W2 with different VM allocation policies with MDASA

Figure 4.2 shows that percentage of tardy jobs for OnDemand allocation policy is minimum (4.4 for W1 and 4.5 for W2). OnDemand Instances are highly reliable and the number of tardy jobs is a result of error in number of slots prediction. While for Spot allocation policy, percentage of tardy jobs is extremely high (about 50 percent for both W1 and W2). Spot instances are highly unreliable, and as a result yield maximum tardy jobs.

However it can be seen that strategic use of Spot instances along with OnDemand instances decreases percentage of tardy jobs to 17 percent for both W1 and W2. Still, this is much more than that obtained in only OnDemand allocation policy. One important observation is that percentage of tardy jobs is independent of size of workload for all the three scheduling policies. W1 is a 4 hour long workload with 1000 jobs, and W2 is a 24 hour long workload with approximately 6000 jobs. So W2 can be considered as 6 folds of W1 workload, where in each fold the ratio of tardy jobs to total number of jobs in a fold is approximately equal to W1's, which is consistent with the observation of constant hourly spot instance failure probability as shown in [13]. So, if considered as a whole, W2 has percentage of tardy jobs equal to that of W1, and it should not vary if we further increase the dataset size.

Now, we will compare the average cost of VMs incurred in running W1 and W2 using the three VM allocation policies. We are considering a homogeneous cloud environment, with only one type of VM instance. We used amazon EC2 m3.medium instance in our calculations. Its OnDemand price is 0.07$ per hour and Spot price lies between 0.008$ to 1$ per hour. We used the spot instance bid data from [13].

Table 4.2 Average Cost of VMs ($) incurred in W1 and W2 with different allocation policies with MDASA

Allocation Policy/Dataset	Cost of VMs for W1 ($)	Cost of VMs for W2 ($)
OnDemand	307	2486
Spot	272	2300
Hybrid	170	1279

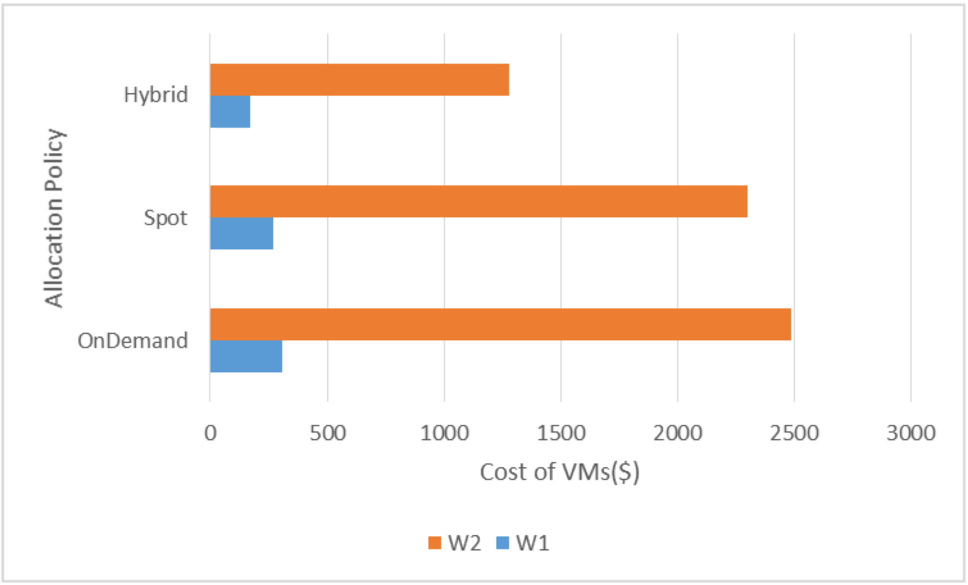

Figure 4.3 Cost of VMs incurred in W1 and W2 with different allocation policies with MDASA.

OnDemand allocation policy incurred maximum average cost of VMs due to high prices of OnDemand instances. Costs of VMs incurred by Spot allocation policy are comparable to those obtained in OnDemand policy. This is because spot instance market price of m3.medium instance fluctuates between 0.01$ and 1.0$, where the upper limit is extremely high as compared to its OnDemand instance price of 0.07$. We obtained highly fluctuating results for cost of VMs for Spot allocation policy due to the highly variable cost of spot instances. In some cases, the cost incurred was as low as 160$ and in others, as high as 330$ for W1. On the other hand, Hybrid allocation policy incurs minimum average cost of VMs for both W1 and W2. So, we can see that, with strategic use of Spot instances along with OnDemand instances, we can decrease cost of operation by around 50 percent.

So, we see that Hybrid allocation policy provides the best cost/performance trade-off of all the three allocation policies. So, for high priority deadline constraint tasks, OnDemand allocation policy should be used due to its high reliability. For all other tasks, Hybrid allocation policy should be used due to its best cost/performance trade off.

4.4.2 Comparison with other scheduling algorithms

In this section we'll compare our proposed algorithm MDASA with NDASA, epoch based Earliest Deadline First (epEDF) [1] and event based Earliest Deadline First (evEDF). All the schedulers have exactly same architectures, only difference is in the scheduling algorithm module. All the schedulers use Hybrid VM allocation policy.

Table 4.3 Percentage of tardy jobs for W1 and W2 using different scheduling algorithms

Scheduling Algorithms/Datasets	Percentage of tardy jobs in W1	Percentage of tardy jobs in W2
MDASA	18.7	17.4
NDASA	18.9	17.45
epEDF	21	22.26
evEDF	24.6	26.5

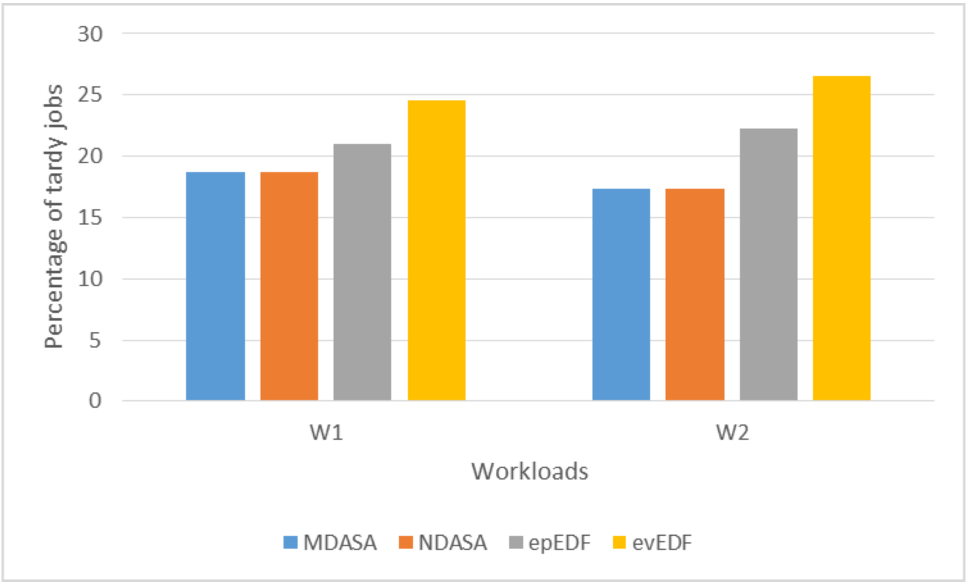

Figure 4.4 Percentage of tardy jobs in W1 and W2 for different scheduling algorithms.

We can see from Figure 4.4 that MDASA outperforms all other competitors regarding the number of tardy jobs. MDASA and NDASA perform similarly because both use same concept of slot allocation. The difference will be seen in cost of operation as NDASA allocates more new slots. evEDF and epEDF have more tardy jobs because the workloads also have small jobs. If the epoch time in epEDF is greater than deadline of a newly arriving job, the deadline will be definitely missed. Similarly, in evEDF which waits for events such as low utilization, and low load, waiting process can lead to expiration of deadline of many jobs.

In Figure 4.5, we compare the cost of VMs incurred when MDASA, NDASA, epEDF, evEDF are used to schedule workloads W1 and W2. Table 4.4 summarizes the results. MDASA outperforms other algorithms in case of cost of operation also, as shown in Figure 4.5.

Table 4.4 Average Cost of VMs ($) incurred on W1 and W2 with different scheduling algorithms

Scheduling Algorithm/Dataset	Cost of VMs incurred for W1 ($)	Cost of VMs incurred for W2 ($)
MDASA	170	1279
NDASA	210	1586
epEDF	185	1427
evEDF	180	1348

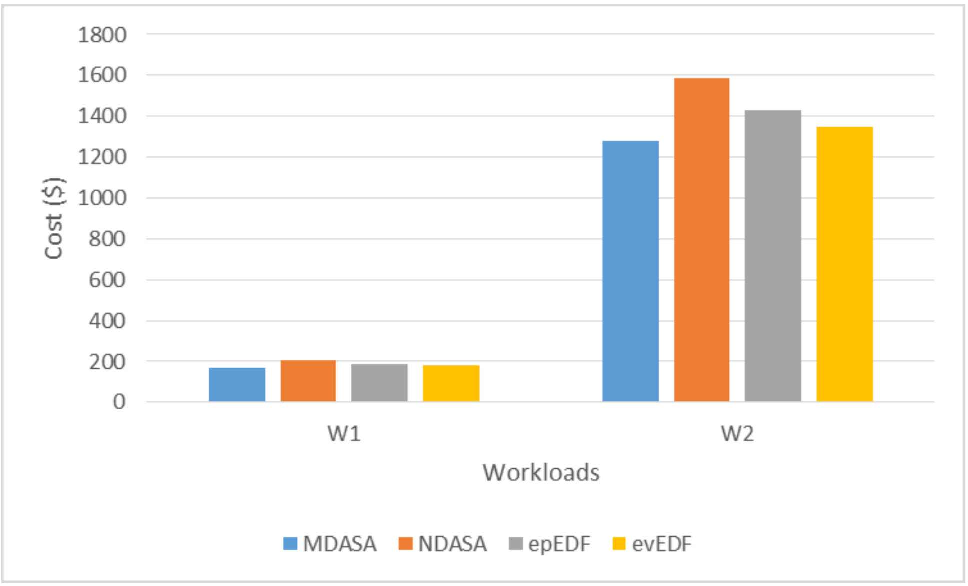

Figure 4.5 Cost of VMs incurred on W1 and W2 using different scheduling algorithms

NDASA incurred much larger cost than MDASA as it doesn't take job malleability into account and allocates more "new" slots. In case of epEDF, epoch can actually work in favor by releasing resources within epochs. As a result the scheduler need not allocate new slots, thus reducing cost of VMs. However, increased waiting time of jobs leads to increased demand for slots, as a result balancing out the previous gain, and increasing the overall cost. evEDF incurs cost of VMs comparable to that incurred by epEDF due to reasons stated above.

Chapter 5

Makespan Aware Scheduling

In this chapter we propose a novel algorithm for scheduling of a batch of MapReduce jobs with the objective of makespan minimization.

5.1 Motivation

Hadoop has a default FIFO scheduler in order to schedule jobs. The primary objective of a batch of production jobs is to minimize the makespan. Such production jobs are analyzed off-line for optimizing their execution. Production jobs are submitted periodically, each set has a similar structure, and set of applications, but processing different data sets. Minimization of makespan is necessary as cluster can be used for other processing tasks and ad-hoc queries after the execution of production jobs.

In this chapter we are going to consider a batch of unrelated production jobs, i.e. map and reduce phases of all the jobs are independent and jobs can be executed in any sequence. For data unrelated jobs, once a job completes its map phase, its reduce phase can be started, while next job's map phase can initiated on the freed up map resources. So, there is an overlap of job execution.

We use the abstraction of MapReduce jobs used in [9]. A MapReduce job is represented as a two stage process – map stage and reduce stage, each executed on two machines map machine and reduce machine. Set of map slots available for execution form map machine and set of reduce slots available for execution form reduce machine. We store job profiles discussed in Chapter 3 and use that to predict job phase completion times on new data sets. Using these job phase completion times use predict the order of jobs with minimum makespan.

Classical Johnson algorithm can be used to schedule 2 stage jobs onto 2 machines to minimize makespan. But sometimes Johnson algorithm can give a suboptimal solution in our case because there can be jobs which partially utilize a machine, as a result leading to underutilization of cluster. We compared the Johnson algorithm with genetic algorithm for the problem of finding a makespan optimal schedule. We then propose a modified version of BalancedPools heuristic proposed in [9] based on genetic algorithm and compare their results.

5.2 Scheduler Architecture

Figure 5.1 shows the architecture of our makespan aware scheduler. Job profiler and Job profile database play the same role as in Deadline-aware scheduler discussed in 4.2. The other three components are:

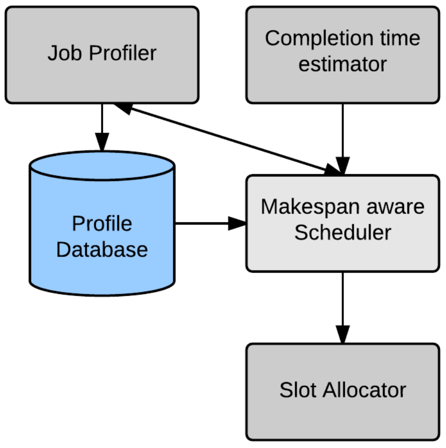

Figure 5.1 Makespan aware scheduler architecture

1. **Completion Time estimator** – Given a job profile and number of map and reduce slots, it predicts map and reduce phase completion times. The estimated completion times are then used by the scheduler module to find out the makespan optimal schedule.
2. **Makespan aware Scheduler** – Given a set of production MapReduce jobs and phase completion time estimates from Completion time estimator, makespan-aware scheduler gives a scheduler with minimum makespan.
3. **Slot Allocator** – Allocates the “map machine” and “reduce machine” to map and reduce tasks respectively.

5.3 MapReduce Job Abstraction

Each MapReduce job consists of a number of map tasks and reduce tasks. Number of map tasks depends upon input data size, number of reduce tasks is configured according to number of slots available and input data size. The abstraction used in our approach as same as used in [9]. Each MapReduce job is represented as a two stage process: (1) map stage and (2) reduce stage. Let for job J, M_j and R_j be the map and reduce phase completion times. J is represented as(M_j, R_j). Using this approach and considering jobs as unrelated, we can pipeline the execution of these jobs, as shown in Figure 5.2.

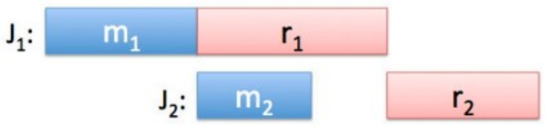

Figure 5.2 Pipelined execution of two MapReduce jobs J_1 and J_2 *[9]*

Order of execution of jobs affects the total batch completion time. Consider execution of 2 MapReduce jobs: $J_1 = (20s, 2s)$ and $J_2 = (2s, 20s)$, which arrived in order J_1 and J_2. FIFO scheduler's scheduler is shown in Figure 5.3. The makespan of this schedule is 40s.

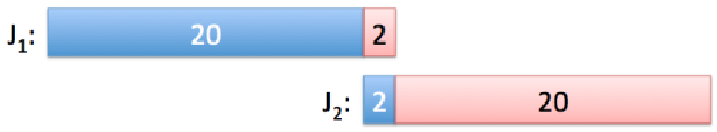

Figure 5.3 FIFO un-optimal schedule *[9]*

While the optimal schedule would be the one shown in Figure 5.4. Makespan of this schedule is 24s.

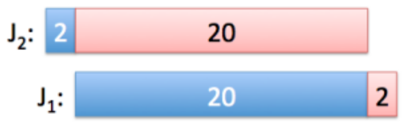

Figure 5.4 Optimal Schedule *[9]*

So, there can be significant improvement in the total batch completion time depending on the order of execution.

5.4 Problem Formulation

We are given a set of n jobs $J = \{J_1, J_2, \dots, J_n\}$, which are unrelated and have no data dependencies. Our aim is to determine the order of execution if we have m map and r reduce slots, such that the order is makespan optimized.

5.5 Existing Algorithms

In this section we discuss the already existing algorithms proposed for this purpose.

5.5.1 Johnson's Algorithm [14]

In 1953, Johnson gave a classic rule for 2 stage job scheduling. We can use this rule to minimize makespan of a set of MapReduce jobs on two machines – map and reduce. Let D be the list of tuple (D_i, T_i) where, D_i is $\min(M_i, R_i)$ and T_i is either M or R, for job J_i. Algorithm 3 shows Johnson's algorithm in detail:

Algorithm 3 Johnson's rule [14]

```
1:  Sort the set $J$ of jobs according to the list $D$ according to $D_i$
2:  $i \leftarrow 1, j \leftarrow n, S = \{\}$
3:  for each job $J_i$ in $J$ do
4:      if $T_i = M$ then
5:          $S_i = J_i$
6:          $i \leftarrow i + 1$
7:      else
8:          $S_j = J_i$
9:          $j \leftarrow j - 1$
10:     end if
11: end for
```

5.5.2 Genetic Algorithm

Metaheuristics are being widely used in scheduling problems. Scheduling a set of jobs to minimize makespan is an NP hard problem. Metaheuristics are an efficient method for such problems. We consider genetic algorithm as our target metaheuristic.

A random schedule of the form $S_i = \{J_x : x \in (1, n)\}$ forms an individual of the population. Fitness of an individual is given by $makespan(S_i)$. We use roulette wheel selection and single point crossover.

Algorithm 4 Genetic Algorithm

```
1:  Choose initial population of size $f(n)$
2:  for each individual $S_i$ do
3:          $fitness_{Si} = makespan(S_i)$
4:  end for
5:  determine average fitness of population
6:  repeat
7:          select best ranking individuals to reproduce
8:          mate pairs at random
9:          apply crossover operator
10:         evaluate each individual's fitness
11: until enough generations have been produced
```

5.5.3 BalancedPools Heuristics [9]

Johnson and Genetic algorithm sometimes give sub optimal solution. The problem lies with the abstraction method we have used. We consider only one job on one machine at a time. Suppose a job requires only half of the total slots available, then we are underutilizing the cluster.

Consider an example of five jobs shown in Table 5.1. Number of map and reduce slots is 30 each. J_1, J_2, J_5 each have 30 map and 30 reduce tasks. J_4 and J_3 have 20 map and 20 reduce tasks.

Table 5.1 Example of five MapReduce Jobs. m_i denotes total map phase time of ith job on 30 map slots. r_i denotes total reduce phase time of ith job on 30 reduce slots

J_i	m_i	r_i
J_1	4	5
J_2	1	4
J_3	30	4
J_4	6	30
J_5	2	3

Both Johnson's algorithm and genetic algorithm give the schedule shown in Figure 5.5 with makespan = 47s.

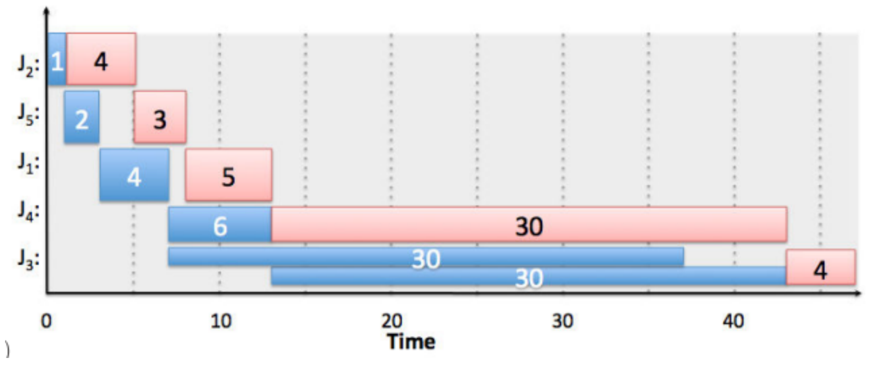

Figure 5.5 Job schedule using Johnson's and genetic algorithm *[9]*

But this schedule is not optimal. Let us split the available resource pool into two parts – Pool1 with 10 map and 10 reduce slots, Pool2 with 20 map and 20 reduce slots. Now, schedule J_1, J_2, J_5 on Pool1 and J_4 and J_3 on Pool2 using Johnson's algorithm. The resultant schedule is shown in Figure 5.6 with makespan = 40.

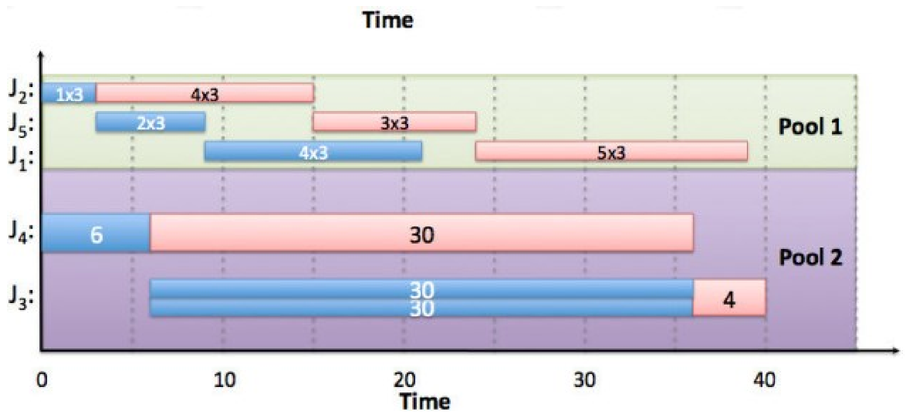

Figure 5.6 Optimal solution with resource pool splitting *[9]*

BalancedPools algorithm proposed in [9] finds a split point in resource pool and a split point in jobs list such that a combination of these two split points produces a schedule whose makespan is less than Johnson's algorithm makespan.

5.6 Proposed Algorithm – BalancedPools with Genetic Search

We observed that abstraction problem affects both Johnson algorithm and Genetic algorithm for finding schedule with minimum makespan. However, Genetic algorithm performs better than Johnson's algorithm when the search space is extremely large, i.e. number of jobs in the batch is very high. So, we have proposed a modified version of BalancedPools algorithm [9], which uses Genetic algorithm as the search algorithm within a pool.

Algorithm 6 BalancedPools with Genetic Search Algorithm

```
1:  Sort $J$ based on increasing number of map tasks
2:  BestMakespan ← $Simulate(J, GeneticOrder(J), M)$
3:  for $s \leftarrow 1$ $to$ $n - 1$ do
4:      $Jobs_a \leftarrow (J_1, \dots, J_s)$
5:      $Jobs_b \leftarrow (J_{s+1}, \dots, J_n)$
6:      $Size_{begin} \leftarrow 1, Size_{end} \leftarrow M$
7:      repeat
8:          $Size_{mid} \leftarrow (Size_{begin} + Size_{end})/2$
9:          $Makespan_a \leftarrow Simulate(Jobs_a, GeneticOrder(Jobs_a), Size_{mid})$
10:         $Makespan_b \leftarrow Simulate(Jobs_b, GeneticOrder(Jobs_b), M - Size_{mid})$
11:         if $Makespan_a < Makespan_b$ then
12:             $Size_{end} \leftarrow Size_{mid}$
```

```
13:             else
14:                 $Size_{begin} \leftarrow Size_{mid}$
15:             end if
16:         until $Size_{begin} \neq Size_{end}$
17:         $Makespan \leftarrow \max(Makespan_a, Makespan_b)$
18:         if $Makespan < BestMakespan$ then
19:             $BestMakespan \leftarrow Makespan$
20:         end if
21:     end for
```

5.7 Experimental Setup and Results

The proposed scheduler architecture has been implemented in CloudSim simulator version 3.0.3. We use Yahoo! M45 cluster workload. As used in [9], we used this workload to build a production workload consisting of 6 jobs every one hour in our simulation. Each job consists of number of map and reduce tasks drawn from the distribution $\mathcal{N}(150, 550)$ and $\mathcal{N}(19, 145)$ respectively, where $\mathcal{N}(a, b)$ is the normal distribution with parameters a and b. Duration of these tasks is defined by $\mathcal{N}(50, 200)$ and$\mathcal{N}(100, 300)$, respectively. Each VM is configured for 2 map and 2 reduce slots. So for 100 map/reduce slots, we require 50 VMs. We compare our proposed approach with Johnson's rule, Genetic algorithm and BalancedPools.

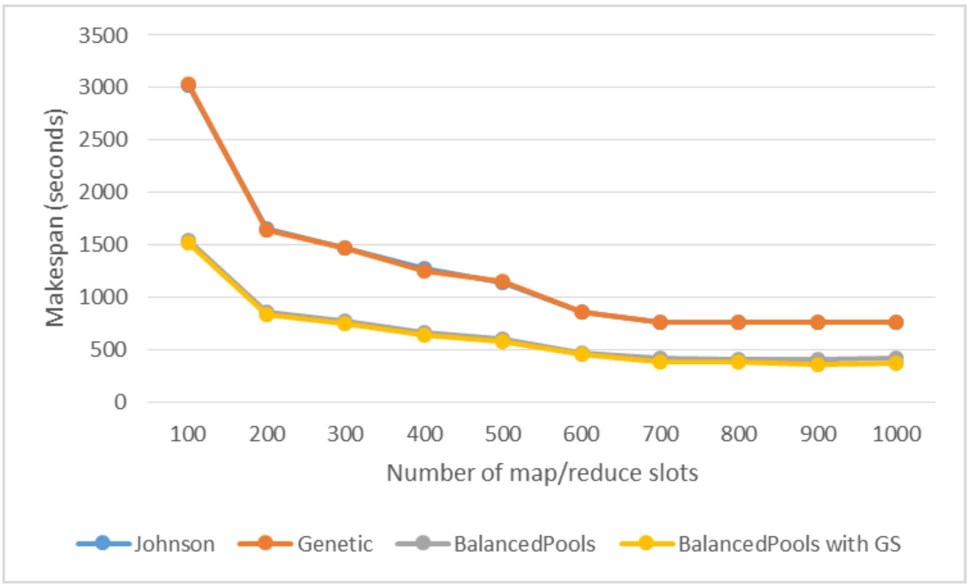

Figure 5.7 Average makespan of one day's production workload for different cluster sizes

Figure 5.7 shows average makespan of 1 day's production workload across different cluster configurations. Johnson's and Genetic algorithm perform similarly with suboptimal solutions. However, genetic produces better results in certain configurations. Once we consider our abstraction at tasks/slot level, Johnson and Genetic give suboptimal results. BalancedPools algorithm gives much better results with a decrease of around 31 percent in average makespan of the proposed schedules. The performance of the proposed BalancedPools with Genetic Search algorithm is mixed. In some cases, its proposed schedule is better than BalancedPools, in other cases it is not. However, it performed a bit better than BalancedPools by giving a decrease of around 34 percent in average makespan as compared to Johnson and Genetic Algorithm.

Chapter 6
Conclusion and Future Research Directions

We have detailed work to support our hypothesis, that only a single performance model of MapReduce environment running on Cloud is sufficient for enabling different service level objectives. We build a performance modelling framework which is simple, yet encapsulates all the important and critical characteristics of MapReduce jobs. This model was then used in two scheduling frameworks – (1) Deadline-aware scheduler and (2) Makespan-aware scheduler. For deadline constraint scheduling we proposed a novel Malleability based Deadline-aware Scheduling Algorithm. We compared it with existing algorithms and verified that it had minimum tardy jobs and minimum cost of operation of all the competing algorithms. For large production jobs we proposed a modified version of BalancedPools heuristic algorithm based on Genetic Search. Our experimental results showed that our proposed algorithm outperformed the BalancedPools algorithm, Johnson's rule and Genetic algorithm. Taken as a whole, our research demonstrates practical performance models and tools that enable different service level objectives in MapReduce environments.

Below we would like to state some future research directions for our work:

- Extension of the proposed performance model and scheduling algorithms to heterogeneous cloud environment.
- Addition of support for multi-user cluster.
- The proposed modification of BalancedPools heuristic splits resource pool into two pools, like the original BalancedPools heuristic. A multi-split BalancedPools algorithm can be developed to split the resource pool more efficiently into more than two pools.
- Finally, incorporation of all the entities i.e., performance model, deadline-aware job scheduler and makespan-aware job scheduler into a single resource inference and allocation system.

References

[1] A. Verma, L. Cherkasova, and R. Campbell, “SLO-Driven Right-Sizing and Resource Provisioning of MapReduce Jobs,” *LADIS 2011 5th Work. Large Scale Distrib. Syst. Middlew.*, no. 126, 2011.

[2] A. Verma, L. Cherkasova, and R. Campbell, “Resource provisioning framework for mapreduce jobs with performance goals,” *Middlew. 2011*, 2011.

[3] A. Fox, R. Griffith, and A. Joseph, “Above the clouds: A Berkeley view of cloud computing,” *... , Berkeley, Rep. ...*, 2009.

[4] G. Lee, B. Chun, and R. Katz, “Heterogeneity-aware resource allocation and scheduling in the cloud,” *Proc. HotCloud*, 2011.

[5] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, “Statistics-driven workload modeling for the Cloud,” *2010 IEEE 26th Int. Conf. Data Eng. Work. (ICDEW 2010)*, pp. 87–92, 2010.

[6] I. S. Matei Zaharia, D Borthakur, J S Sarma, K Elmeleegy, S Shenker, “Job scheduling for multi-user mapreduce clusters,” *EECS Dep. Univ. Calif. Berkeley Tech Rep UCBEECS200955 Apr*, 2009.

[7] T. Sandholm and K. Lai, “Dynamic proportional share scheduling in hadoop,” *Job Sched. Strateg. parallel Process.*, pp. 110–131, 2010.

[8] J. Wolf, D. Rajan, K. Hildrum, and R. Khandekar, “Flex: A slot allocation scheduling optimizer for mapreduce workloads,” *Middlew. ’10 Proc. ACM/IFIP/USENIX 11th Int. Conf. Middlew.*, 2010.

[9] R. H. C. Abhishek Verma, Ludmila Cherkasova, “Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance,” *Proc. 2012 IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS 2012*, 2012.

[10] D. Poola, K. Ramamohanarao, and R. Buyya, “Fault-tolerant Workflow Scheduling using Spot Instances on Clouds,” *ICCS 2014. 14th Int. Conf. Comput. Sci. Fault-Tolerant*, vol. 29, pp. 523–533, 2014.

[11] N. Chohan, C. Castillo, and M. Spreitzer, “See spot run: using spot instances for mapreduce workflows,” *HotCloud’10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput.*, 2010.

[12] Y. Chen and A. Ganapathi, “The case for evaluating MapReduce performance using workload suites,” *Model. Anal. ...*, 2011.

[13] B. Javadi, R. K. Thulasiramy, and R. Buyya, “Statistical Modeling of Spot Instance Prices in Public Cloud Environments,” *2011 Fourth IEEE Int. Conf. Util. Cloud Comput.*, pp. 219–228, Dec. 2011.

[14] S.-C. N. Peng-Sheng Ku, “On Johnson’s Two-Machine Flow Shop with Random Processing Times.pdf,” *Oper. Res.*, vol. 34, no. 1, pp. 130–136, 1986.