DESIGN AND MODELING OF SCHEDULERS FOR MULTI-TASK JOBS ON COMPUTER CLUSTERS

by

Shouvik Bardhan
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

____________________	Dr. Daniel Menascé, Dissertation Director
____________________	Dr. Hassan Gomaa, Committee Member
____________________	Dr. Hakan Aydin, Committee Member
____________________	Dr. Brian L. Mark, Committee Member
____________________	Dr. Sanjeev Setia, Department Chair
____________________	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: ____________________ Spring Semester 2015
George Mason University
Fairfax, VA

Design and Modeling of Schedulers for Multi-Task Jobs on Computer Clusters

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Shouvik Bardhan
Master of Science
The Johns Hopkins University, Baltimore, USA, 1997
Bachelor of Science
Indian Institute Of Technology, Kharagpur, India, 1985

Director: Dr. Daniel Menascé, Professor
Department of Computer Science

Spring Semester 2015
George Mason University
Fairfax, VA

Copyright © 2015 by Shouvik Bardhan
All Rights Reserved

Acknowledgments

I am grateful to my thesis advisor, Dr. Daniel Menascé, for his enthusiastic guidance, encouragement, and support with my doctoral work. His technical and editorial advice was indispensable for the completion of this dissertation, and has taught me invaluable lessons on the workings of academic research. I would also like to thank my committee members Dr. Hassan Gomaa, Dr. Brian L. Mark, and Dr. Hakan Aydin for their advice and comments, which helped improve this dissertation. I would like to especially thank my wife Shoma and our son Ishan for their love, patience and understanding during the past six years. Thanks also go to my late parents, my treasured parents-in-law and other family members for their prayer and support. Last, but not least, I would like to thank my friends and colleagues, with a special shout-out for Doug Grosvenor, for their support during my years of study.

Table of Contents

List of Tables

List of Figures

Abstract

DESIGN AND MODELING OF SCHEDULERS FOR MULTI-TASK JOBS ON COMPUTER CLUSTERS

Shouvik Bardhan, PhD

George Mason University, 2015

Dissertation Director: Dr. Daniel Menascé

Over the course of the last decade, a tremendous amount of information has been generated, collected, and eventually stored on modern server scale computers. This ongoing process led to the coinage of the term Big Data, a concept that the industry uses to describe the all-encompassing activities of storing and processing this petabyte scale data. Stakeholders, academics, and technologists are trying to extract actionable intelligence from this huge volume of data, while massive clusters of computers are now being used to process the continually expanding amount of information.

This dissertation addresses two specific challenges brought about by the need to process Big Data jobs efficiently on clusters of computers. The first challenge is the design, development, and assessment of sophisticated job schedulers capable of choosing tasks from thousands of queued jobs and executing them on one of the several hundred available machines. The second challenge is to predict a job's completion time based on its various characteristics and that of other jobs executing in the same environment. Standard makespan computations that ignore the contention on compute nodes significantly underestimate a job's completion time.

This dissertation begins by proposing a mathematically sound model, based on Queuing Networks (QN), for predicting the execution time of programs running on a cluster of computers. A special emphasis is placed on Hadoop MapReduce jobs. The model captures contention at compute nodes due to the parallel execution of multiple map tasks running on a node's available slots. In multi-core computers used for most Big Data processing today, memory access times can become a significant component of an application's execution time. This work presents a multi-level, multi-core performance model that considers the contention due to this shared memory access. The model was validated through measurements on 4, 12, and 16-core machines. We also showed that there is a significant difference in predictions when memory contention is not considered. Having captured the complexity of the tasks of MapReduce jobs, including considering the effect of contention due to shared memory elements, we then used this model to augment existing Hadoop schedulers and developed a new scheduler, called the Contention Aware Scheduler. This Hadoop scheduler is based on a novel scheduling method called Least Maximum Utilization First - Threshold (LMUF-T), which we consider a step towards an autonomic scheduler. We also developed a framework called Trace Driven Analytic Model (TDAM) to assess existing and newly designed Hadoop job schedulers through the enhancement of a popular Hadoop simulator called Mumak.

Chapter 1: Introduction

1.1 Job Schedulers

On very rare occasions, a computer program is run in isolation on modern computers. More often, the program shares the machine with other programs and the operating system (OS) distributes the machine resources (e.g., processors, disks, and networks) as fairly as it can to the group of programs that need to utilize them. With organizations collecting and analyzing petabytes of data, it is virtually impossible to get the jobs finished in a timely manner on only one computer. Clusters of computers, numbering in hundreds or even thousands, are being used to run a multitude of tasks across various computing nodes. What is normally managed and controlled by the OS when only one computer is used, now has to be controlled by an overarching software program, a cluster-wide job scheduler that keeps track of the entire cluster of computers, decides which task of which job to run on which node and when and also if the tasks are being scheduled in the most efficient way possible with an eye towards the utilization and throughput of the machines. During the last decade or so, the importance of multi-node schedulers has thus come to the forefront as a result of the computing industry's focus on the phenomenon called "Big Data".

1.2 Big Data Processing and MapReduce

Big Data processing is a technique for capturing value from data, which has large volume and has tremendous variety and a high speed of generation, in a timely and cost effective manner. Big Data, which has been prevalent in the scientific community, is now making an entry into main-stream computing. Popular discussion and endless attempts to define Big Data have reached a point where scientists and technologists needed to provide something concrete in terms of architecture, design and development of Big Data processing systems. This has not proved easy. Big Data's data volume, as the name implies, is extremely large and thus the storage, processing, provenance requirements are more intensive. Data is typically generated at high speed which implies in less time available to process, analyze, and make inferences from the data. There is a wide variety of data which adds an extra level of complexity to the work of correlating data. And lastly when the data pedigree is less than well known, decision-making proves to be riskier.

Many US federal agencies including NOAA and NASA are working with ultra large datasets and their realms of work, atmosphere and space, are generating data whose magnitude is beyond dispute. The US government has hundreds of Big Data projects. In the recent past, six federal departments and agencies committed \$200 million to ”greatly improve the tools and techniques needed to access, organize, and glean discoveries from huge volumes of digital data” [94]. In the commercial landscape, financial services, retail manufacturing, social media, computational chemistry, to name a few verticals, are also addressing Big Data computing.

All that data needed to be stored and processed and new tools were needed for this purpose. Fortuitously, Google was by that time a mature company innovating new ways of computing with ultra large datasets. In a seminal paper [22], authors explained the motivation behind introducing the MapReduce programming paradigm. Most of the computations performed on a regular basis in Google fell into a pattern whereby a *map()* operation needed to be applied to each input record which generated an intermediate key/value pair and then subsequently a *reduce()* operation was needed which combined all the data with the same key in a desired fashion to form the output of the program. Over the last few years, the Big Data community has determined how to run large scale computing workloads on commodity servers in an efficient way primarily based on this groundbreaking work. The thrust of this dissertation is to analyze, understand and enhance Big Data processing software frameworks, including Hadoop [7], and enhance its vital sub-components such as the job scheduler and trace simulator with node contention awareness using mathematically sound models based on queuing network.

1.3 Multi-Node Job Scheduling

The need for job schedulers is well understood. As soon as jobs, processes or tasks needed to share a single CPU, there was a need for a supervisor and a scheduler to make sure that interleaved execution of multiple jobs was possible on a single computer (and a single CPU by extension). Since the early days of multiprogramming, much improvement has been made to scheduling disciplines that distribute and allocate computing resources among processes who are requesting them simultaneously. The main purpose of scheduling algorithms is to ensure that processes are not starved of resources and that fairness is maintained while allocating resources. Several different schedulers have been developed for Linux-like operating systems over the years. Completely Fair Scheduler (CFS) is the latest Linux scheduler [86]; its purview only extends up to the single machine on which it is running. It is not uncommon today to see a computer cluster consisting of hundreds of individual Linux machines. When a job comprising of multiple tasks joins the cluster for execution, there is a need for a job scheduler that is aware of the entire cluster. The scheduler works by keeping the state of the entire cluster, while dispensing tasks to the executing nodes based on jobs' priority and other job characteristics.

Hadoop, which is a platform for running multi-task MapReduce jobs, has such schedulers that can be used based on how the organization wants to use the cluster. The Hadoop Fair Scheduler [104] as well as the Capacity scheduler [103], which provides a way to share a large cluster with the help of multi-level queues have a root in high performance super computing (HPC). We have studied various multi-node multi-task job schedulers and have designed and developed software components that we believe give these existing schedulers awareness of the contention experienced by the executing nodes and thus schedule tasks more effectively.

1.4 Hadoop MapReduce Job Scheduling

The Hadoop Distributed File System (HDFS) is the open source implementation of the Google File System. HDFS consists of many DataNode daemons which are responsible for storing file system data in a replicated fashion and these services run as distributed components on a cluster of servers.

In a production or integration environment, hundreds or even thousands of processing nodes can help an application scale out linearly and handle a tremendous amount of data. Hadoop also does this in a way that makes it reliable, efficient, and scalable. Hadoop is reliable because it assumes that computing elements and storage may fail and, therefore, maintains several copies of working data to ensure that processing can be redistributed around failed nodes. Hadoop is also very efficient in processing certain classes of algorithms because it works on the principle of parallelization, allowing data to be processed in parallel to increase the processing speed. In addition, Hadoop relies on commodity servers, making it inexpensive and obviating the need for costly and specialized server class machines.

A popular paradigm for handling Big Data applications is the MapReduce programming model [21]. Hadoop is a popular open source implementation of MapReduce available from Apache [90]. Many commercial firms and government organizations use Hadoop and its complimentary products to process vast amounts of data on clusters of commodity hardware. Performing computations on this kind of platform is attractive because Hadoop is able to handle a vast amount of data in its native file system while supporting the MapReduce paradigm. The MapReduce model of computation is inspired by map and reduce functions commonly found in functional languages (e.g., Erlang, Lisp) but the purpose of MapReduce in the area of Hadoop and large scale data processing is quite different than how they are used in the functional language domain. At its core, a MapReduce program runs on a cluster of computers where many tasks are spawned for a particular job. These tasks are responsible for implementing the map and reduce functions which work on the dataset of a file system (unstructured data mainly) or from a database (structured data). All map and thereafter all reduce functions can be executed in parallel. Hadoop's implementation of this MapReduce framework, along with this offering of parallelism which can handle petabytes of data for a single job, also provides the potential to recover from some storage and server failures. We describe MapReduce in greater detail with examples later in this dissertation.

In summary, Hadoop is a multi-tasking software system that can process multiple data-sets for multiple jobs for many users at the same time. This means that Hadoop attempts to schedule jobs in a way that makes optimum usage of the resources available on the compute cluster. Default schedulers distributed with Hadoop did an inadequate job of scheduling tasks in an efficient fashion and even today its scheduling algorithms and schedulers are not very sophisticated.

At the inception of Hadoop, the original scheduler distributed was a first-in first-out (FIFO) scheduler woven into the Hadoop job controller known as the JobTracker (we discuss all the relevant components of Hadoop in great detail in later chapters). Even though it was simple, the implementation was inflexible and could not be tailored. After all, not all jobs have the same priority and a higher priority job is not supposed to languish behind a low priority long running batch job. Around 2008, Hadoop introduced a pluggable scheduler interface that was independent of the JobTracker. The goal was to develop new schedulers that would help optimize scheduling based on particular job characteristics. This pluggable scheduler architecture made greater experimentation possible and specialized schedulers are being designed to cater to ever increasing types of Hadoop MapReduce applications. The overarching goal of this dissertation is to develop a scheduler for Hadoop like systems that follows the tenets of autonomic computing and also is rooted in proven analytic performance modeling techniques based on Queuing Network theory.

1.5 Thesis Statement

The dissertation addresses the problem that it is not possible (with existing techniques) to perform effective scheduling without being aware of node contention when numerous multi-task jobs need compute resources on modern computer clusters consisting of hundreds of nodes.

My thesis statement is that it is possible to design and implement a dynamic cluster scheduler that

1. Follows the interface dictated by a multi-node compute framework (e.g., Hadoop MapReduce)
2. Is able to use performance modeling to predict with accuracy the makespan of jobs in the face of varying workloads
3. Is based on proven analytical performance models built upon queuing theory
4. Is robust on multi-core CPU based machines commonly used today.

1.6 Contributions of the Dissertation

The main contributions of this dissertation are as follows. Firstly, existing literature of Hadoop MapReduce performance modeling only considers the makespan of the map and reduce tasks and not the contention based on the fact that multiple processes running on computing nodes generate queuing effects on CPU and I/O devices and thus increase the makespan. Our MapReduce performance model on the other hand utilizes closed form Queuing Networks. Experimental validation shows that our model tracks the completion time of a MapReduce job more closely than the models that do not take these contentions into consideration. Secondly, most Hadoop clusters run on machines with multi-core CPUs. The performance gained by having two or more cores on a CPU is significant especially given the fact that in MapReduce jobs hundreds or more tasks are executed in parallel. With this speed up comes the issue of having to share the memory bus to read and write data from the system's main memory. Once we understand this additional queuing effect created by the shared memory bus, we are able to further enhance our model to include the memory as a separate queue (along with the well known queues at the CPU and disk). Experimental results validate our enhanced application model (which now considers the effect of memory-contention). Our model does indeed track the experimental results much better than the ones that consider only CPU and I/O and ignore memory contention

Another contribution of this dissertation is a hybrid method to assess schedulers for server clusters, such as the ones running Big Data applications. This method, called TDAM (Trace Driven Analytic Model), relies on the implementation of the scheduler under evaluation and on analytic closed queuing network (QN) models to assess resource contention at the cluster nodes. TDAM uses a novel algorithm, which we call the Epochs algorithm, that estimates the execution times of jobs in a job stream. We further implemented the TDAM method inside Hadoop's Mumak, a job-trace simulator distributed with Hadoop [97].

Our final contribution was to use the multi-core Queuing Network based models to create a Contention Aware Scheduler (CAS) for Hadoop. The main goal of the CAS is to utilize the nodes of the cluster in a way so as to be cognizant of the load exerted on the machines as a result of task scheduling. We developed CAS as a standalone scheduler adapting Hadoop's FIFO scheduler.

1.7 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 delves deeper into the issues of scheduling and Hadoop components and enumerates past and ongoing work in this field. Chapter 3 details an analytic model for a single map-only MapReduce job and provides details on the anatomy of a MapReduce job. Chapter 4 describes the technique for modeling multi-core systems and introduces performance modeling that considers memory contention. Chapter 5 describes a novel finite interval algorithm, called the Epochs algorithm, which is an essential component for modeling multiple tasks on a computer. Chapter 6 shows how to use the multi-core model and the Epochs algorithm to develop a trace driven analytic model software component and a contention aware Hadoop scheduler (CA-FIFO). Chapter 7 ends this dissertation with some concluding remarks and a list of future work possibilities.

Chapter 2: Background

2.1 Big Data and Hadoop MapReduce

According to a recent study by IBM, 90% of the total data in the world today has been created in the last few years [105]. Data are coming from everywhere - social media, online transaction records, GPS signals, drones flying over enemy territories gathering intelligence data and from sensors of various kinds. Common software tools created over the last three decades simply cannot store, analyze and manage these data in any meaningful way in an adequate amount of time. The generation of a few dozen terabytes of data at a very high rate (in a matter of days) is not all that uncommon. Big Data tools and new programming paradigms have been developed over the past several years to help make sense over this massive amount of data. Google has re-introduced MapReduce [21, 22] to the world, a programming model for processing a large amount of data on a distributed cluster in a fault tolerant manner. Apache Hadoop is an open source implementation of the MapReduce programming model.

Written in Java (though some C++ versions have become available), Hadoop allows a fundamentally new way to store and analyze vast amounts of information. It is a distributed system, capable of running on commodity server-based machines, on a cluster that can theoretically scale to thousand of nodes. A Hadoop platform consists of the MapReduce (MR) framework and the Distributed File System (DFS). Essentially, DFS allows data to be stored on Hadoop nodes and the MR framework allows parallel programs to be written to analyze the stored data. Industry and academic thrust remain strong behind the Hadoop ecosystem as is evident by the fact that Facebook, Yahoo, Netflix, Amazon and many other companies are active Hadoop users and Big Data courses in general and Hadoop in particular are taught as regular university courses..

Hadoop comes with a MapReduce engine that allows user jobs to be submitted to the cluster. Hadoop daemons (one JobTracker per cluster and one TaskTracker per node) are responsible for breaking down a job into multiple constituent tasks and running these map and reduce tasks over the available resources of the computing nodes. Typically, the compute nodes and the storage nodes are the same. This allows the MR framework to schedule tasks on nodes where the data is already present. MR jobs can be written in a variety of languages, not only Java. This dissertation devotes a large amount of effort to discussing scheduling on a Hadoop MapReduce environment.

2.2 Scheduling

Scheduling is the mechanism that determines when a process or task should be given access to a resource. This is done to load balance a system or to meet a particular Quality of Service (QoS)goal. The notion of scheduling is applicable to fields other than computing (e.g, airlines, manufacturing and broadcast programming). Scheduling is mainly concerned with improving the throughput, latency and fairness of allocation of resources to tasks. Many different kinds of scheduling schemes exist. Some of the well known scheduling strategies are first-come-first-serve, round-robin, priority based and multi-level queue. Multi-level queue is used for situations in which tasks requesting resources can be divided easily into different groups. For example, a common division, in the context of this research, is made between long running and short running Hadoop MapReduce jobs. These two types of jobs have different response-time requirements and therefore have different scheduling needs. Two of the main Hadoop MapReduce schedulers, called Fair and Capacity schedulers, have multi-level queues.

Hundreds of jobs (small/medium/large) may be present on a Hadoop cluster for processing at any given time. How the map and reduce tasks of these jobs are scheduled has an impact on their completion time and consequently on the QoS of these jobs. Hadoop uses a FIFO scheduler by default. Subsequently, two more schedulers have been developed. Firstly, Facebook developed the Fair Scheduler which is meant to give fast response time to small jobs and reasonable finish time to large production jobs. Secondly, the Capacity Scheduler was developed by Yahoo and it has named queues in which jobs are submitted. Queues are allocated a fraction of the total computing resource and jobs have priorities. We contend that no single scheduling algorithm and policy is appropriate for all kinds of job mixes. A mix or combination of scheduling algorithms may actually be better depending on the workload characteristics.

2.3 Queueing Networks

Queueing theory is the mathematical study of waiting lines or queues. Original research on queueing theory was done in reference to a telephone exchange [28] and it is generally considered that queueing theory as a discipline is part of operations research. Queueing theory has found a wide array of applicability in a variety of fields including computer networking and software performance. Modern computer systems consist of resources such as CPUs, disks, networking - all of which can be thought of as single queues. The reason is that the processes that need services from these resources will have to wait in a queue if the resource is busy. Figure 2.1 top part shows two queues. Each queue has a service center (indicated by the circles) which can be either the CPU or the disk. Each queue also has a waiting line (shown as rectangular boxes) where processes wait for their turn to use the service center.

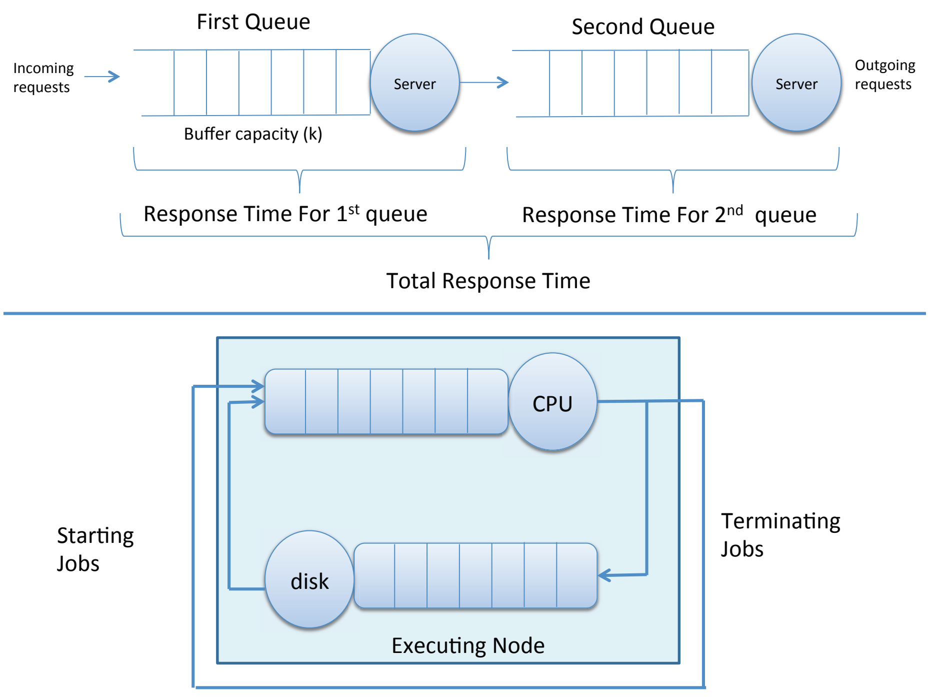


Figure 2.1: Top - Two queues with one service server each; Bottom - A closed queueing network

Modern computer systems can be thought of as a network of queues, also known as a Queueing Network (QN) as shown in the bottom part of Figure 2.1. A queueing network can be either open or closed. An open queueing network is considered when requests are made from outside the network (like a HTTP request) and after processing through the queues, the request leaves the network. Open queuing networks are used when the main input parameter is the average arrival rate of requests. An important output metric of this type of queuing networks is the average response time of requests. Closed queuing networks on the other hand are used when the input parameters consist of the concurrency level of requests, i.e., the number of requests of each type that are concurrently using or waiting to use the system resources. The main output metrics of a closed queuing network are the average response time and average throughput.

A very well-known solution technique for closed QNs is the Mean Value Analysis (MVA) solution method [12,55]. The single job class version of MVA is given by the following three equations that can be solved recursively starting with a concurrency level equal to 0 (i.e., $n = 0$). The following standard notation is used in the equations below: $R_i^{'}(n)$: total time spent by a job at queue i, including service and waiting in the queue; $X_0(n)$: system throughput; and $\bar{n}_i(n)$: average number of jobs at queue i.

$$\begin{aligned} R_i^{'}(n) &= D_i[1 + \bar{n}_i(n-1)] \\ X_0(n) &= n / \sum_{i=1}^{K} R_i^{'}(n) \\ \bar{n}_i(n) &= X_0(n) R_i^{'}(n) \end{aligned} \tag{2.1}$$

The total job execution time for a concurrency level equal to n is $\sum_{i=1}^{K} R_i^{'}(n)$. When the processes executing in a system have different service time requirements and different service level agreements, a model that can handle and analyse multiple classes of jobs is needed. Even though it is possible to calculate aggregate values for a single class equivalent of the multi-class service demands, many typical performance questions cannot be answered this way. As a result, it becomes a necessity to use multi-class models. The exact model solution for multi-class is very expensive to calculate for moderate to large QNs. The solution method for single class queueing networks is easy to implement and its time complexity increases linearly with the concurrency level and number of queues. However, for a multi-class situation, the number of multiplications and additions increase exponentially with the number of classes. The solution to this problem is a technique which is widely known as the Bard-Schweitzer [8,64] proportional estimation algorithm or approximation. It has been shown that for up to 100 classes and 1000 queues, this method provides results with errors under 15%. The work in this dissertation borrowed heavily from this approximation technique. The multi-class version of MVA is given in [55].

2.4 Multi-core Machines

The engineering practice of scaling up the speed of servers and personal computers by increasing the clock speed of their CPUs became impractical around ten years ago. Increased heat due to current leakage and other manufacturing-related issues caused the top clock speed to flatten out at around 4 GHz. Chip designers decided instead to increase the number of processing elements on a chip. Two or more CPU cores packed in a single chip allow the peak performance of CPUs to follow Moore's Law. However, the memory subsystem now has to support the increased number of accesses generated by multiple cores. Thus, memory access becomes the bottleneck and prevents the peak speed of the CPU cores to be achieved.

A multi-core processor executing a single program will not necessarily exhibit a linear speedup unless the program takes full advantage of the parallelism inherent in multi-core machines. However, in today's server machines, hundreds of tasks are working simultaneously. In these situations, multi-core machines provide an immediate advantage because more than one task can be simultaneously executed at any given time.

The performance of a multi-core machine is highly dependent on the design of its memory subsystem because instruction and data lines have to be fetched from memory (DRAM). Figure 2.2 shows a typical architecture of a multi-core computer. The speed of memory and the bus through which the data are transferred are orders of magnitude slower than the speed of the CPU. As a result of this impedance mismatch, a hierarchy of cache memories has been introduced to alleviate the problem. Cache memory consists of high speed memory that sits between the processor and main memory. There are several levels of cache. L1 is the first level and is generally built into the chip itself with fast SRAM (static RAM), and is usually divided into Instruction and Data L1 caches. The size of an L1 cache bank varies typically between 8 KB and 64 KB. L2 and L3 caches (sometimes an L3 cache is not present in modern chips) come downstream from the L1 cache and are larger in size (e.g., 256KB for L2 and 2-4 MB for L3) [35].When a Load or a Store operation is executed, an attempt is first made to find the data in the L1 cache and then in L2, L3 and DRAM in that order. The L1 cache is private for each processor core; however, L2/L3 caches are sometimes shared by multiple cores.

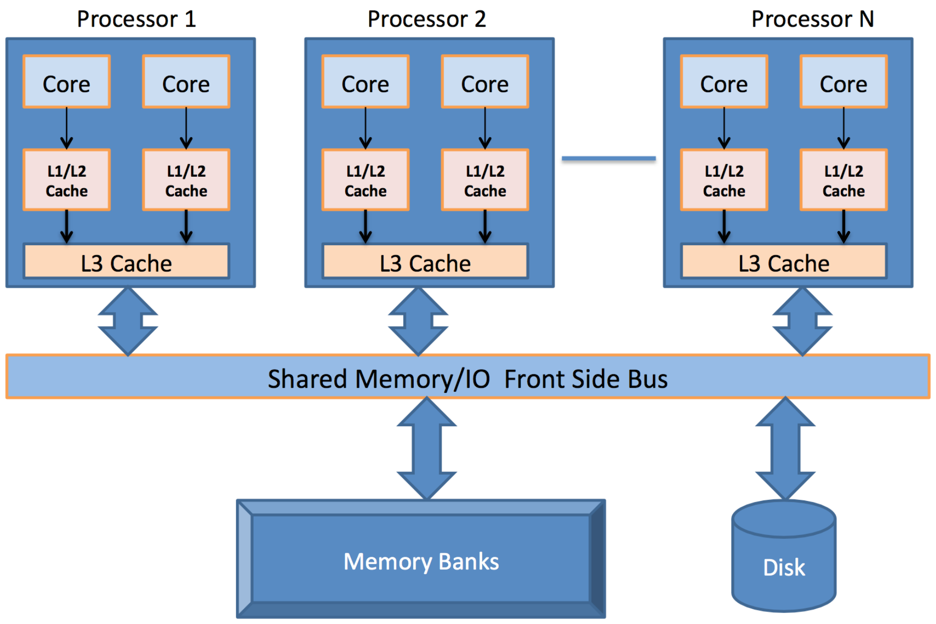


Figure 2.2: Typical architecture of a multi-core computer.

2.5 MapReduce Performance Modeling

Simple models have been created to predict the completion time of MapReduce jobs, but most have neglected resource contention in compute nodes and thereby significantly underestimate the completion time of the jobs [75–77]. Past attempts to estimate the performance of MapReduce jobs with a set of equations, comprising of the many configuration and system variables which are in play when a MapReduce job is executing [37], have provided barely acceptable results. Yahoo has been a big contributor to the Hadoop platform with researchers [42] analysing months worth of Yahoo MapReduce logs and using an instance-based learning technique to create resource utilization patterns, job patterns, and sources of failures. A performance model was discussed in [75], which estimates the amount of resources required for job completion for any given job with a known profile and its SLO (soft deadline). Most of the work is based on examining parameters like amount of map splits, number of mappers, reducers and the amount of data transfer during the shuffle phase. Our strategy is to identify key parameters in a Hadoop MapReduce environment and applying proven queuing theory techniques, which make our model superior and allowed us to create a more effective tool to predict completion times and other performance statistics.

Chapter 3: Modeling of Hadoop "map-only" MapReduce jobs

3.1 Introduction

The work described in this chapter has been published in [6, 7]. The previous chapter provided relevant background information regarding the importance of job schedulers and introductory material on multi-core machines and the Hadoop computing platform. With Big Data analytics being in the forefront of computing, we now dive deep into one of the most popular methods for developing applications dealing with Big data, called the MapReduce programming model. Even though the idea of MapReduce comes from functional languages like Lisp, it was introduced in the mainstream computing by Google [21,22], and the open source implementation of MapReduce has been available since 2008 from Apache under the name of Hadoop [90].

Hadoop, as it is offered today (all research work for this dissertation was conducted with Hadoop 0.22 version), consists of close to half a million lines of Java code and has hundreds of tunable parameters [90]. Configuring Hadoop's parameters for optimal performance for every kind of application is not a tenable goal. But, a deep understanding of the different phases of a MapReduce job allows developers and operations personnel the opportunity to better tune a Hadoop cluster and extract close-to-desired performance.

Hadoop has two main components at its core: the Distributed File System (DFS) and the MapReduce framework. MapReduce paradigm can solve a wide range of computational problems. A shared foundation for a number of systems like Pig [109], Hive [98], Accumulo [91] is provided by Hadoop so that these systems do not have to create their complex and distributed file processing storage and the accompanying methods to retrieve data. Hadoop is still very much MapReduce-based and, therefore, understanding the inner-working of the MapReduce framework is of importance. Just the sheer number of configuration parameters makes it difficult to tune a MapReduce system. Besides, the current Hadoop architecture has potential performance drawbacks for very large clusters because of its architecture of a single JobTracker for all jobs. The Hadoop community is working to resolve these issues and we discuss the main ones later in this chapter.

There have been some prior work on the performance of jobs in a MapReduce environment. Verma et al. [75] built a job profile that summarizes critical performance characteristics of the map, shuffle, and reduce phases of a Hadoop job. They then designed a MapReduce performance model for a job with the known profile and estimate the completion time. Contention based on multiple jobs or varying number of tasks on a machine is not taken into account. Ganapathi et al. [32] use statistical calculations to predict the completion time of Hive generated Hadoop jobs. In [31] they use statistical techniques to predict query performance in parallel databases.

Yang et al. [79] propose a statistical analysis approach to identify the relationships among workload characteristics. They apply cluster analysis to 45 different metrics, which derive relationships between workload characteristics and corresponding performance under different Hadoop configurations. However, they do not consider the delays due to contention caused by multiple running jobs.

Vianna et al. [78] model the performance of a different implementation of MapReduce called Hadoop Online Prototype, which allows data to be pipelined between tasks and between jobs. Their work does indeed utilize Approximate Mean Value Analysis (AMVA) methods. They use simulation rather than experimentation to validate their model.

The completion time of a job in the Hadoop environment is the time needed for all its tasks to complete. The complexity of estimating a job completion time arises because various jobs of different length can share a Hadoop cluster. In addition, jobs can be either CPU, I/O or memory bound (or indeed a mixture of those). Once the job completion times can be predicted for a certain workload and concurrency within a MapReduce cluster, that information can then be used for a myriad of activities such as efficient scheduling of the jobs to maintain a service level agreement, capacity planning, and energy preservation.

In this chapter, after discussing the Hadoop MapReduce framework in some detail, we present a methodology and a queuing network based analytic model for assessing the completion time of a MapReduce job consisting of map tasks only. This type of jobs are frequently encountered in real world scenarios. Experiments have been carried out to validate the model in both a single node as well as a 2-node Hadoop environment. The model proposed is a mathematically sound model based on closed Queuing Networks for predicting the execution time of the map phase of a MapReduce job. The model captures contention at compute nodes and parallelism gains due to increased number of slots available to map tasks. We ran experiments for different input split sizes and different map slot sizes to validate our model.

3.2 Hadoop Ecosystem

There is an extensive list of products and projects that either extend Hadoop's functionality or expose some existing capability in new ways. For example, executing SQL-like queries on top of Hadoop has spawned several products. Facebook started this whole movement when it created HiveQL, an SQL-like query language. HBase [102] and Accumulo [91] are both NoSQL databases (i.e., semi-structured table storage) built on top of Hadoop. It is very common to see users writing MapReduce jobs that fetch data from and write data into a NoSQL storage like HBASE or Accumulo. Finally an Apache project called Pig [109] provides an abstraction layer with the help of scripts in the language Pig Latin, which are translated to MapReduce jobs. Other examples of projects built on top of Hadoop include Apache Sqoop [110], Apache Oozie [107], and Apache Flume [96]. Figure 3.1 shows some of the products built upon and that complement Hadoop.

3.3 Anatomy of a MapReduce Job

The details of MapReduce internals given in this section come from many different sources such as [106,108,111,112] and from my own experience. The MapReduce programming model consists of a map <k1; v1> function and a reduce <k2; list(v2)> function. Users implement their own processing logic by specifying a custom map() and reduce() function written in a general-purpose programming language such as Java or Python. The map <k1; v1> function is invoked for every key-value pair <k1; v1> in the input data to output zero or more key-value pairs of the form <k2; v2>. The reduce <k2; list(v2)> function is invoked for every unique key k2 and corresponding

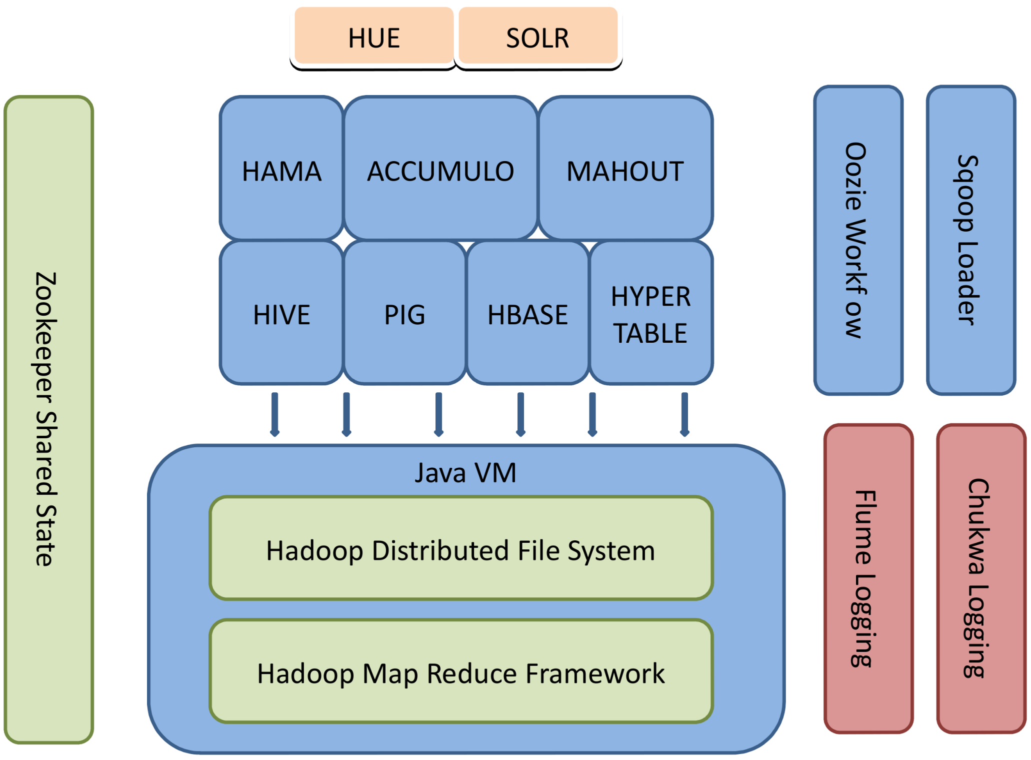


Figure 3.1: Hadoop ecosystem with multitude of products.

values list(v2) from the map output. The function reduce <k2; list(v2)> outputs zero or more key-value pairs of the form <k3; v3>. The MapReduce programming model also allows other functions such as partition(k2), for controlling how the map output key-value pairs are partitioned among the reduce tasks, and combine <k2; list(v2)>, for performing partial aggregation on the map side. The keys k1, k2, and k3 as well as the values v1, v2, and v3 can be of different and arbitrary types. Figure 3.2 shows the data flow through the map and reduce phases. We discuss the input splits and the different phases of a MapReduce job in greater detail later in this section.

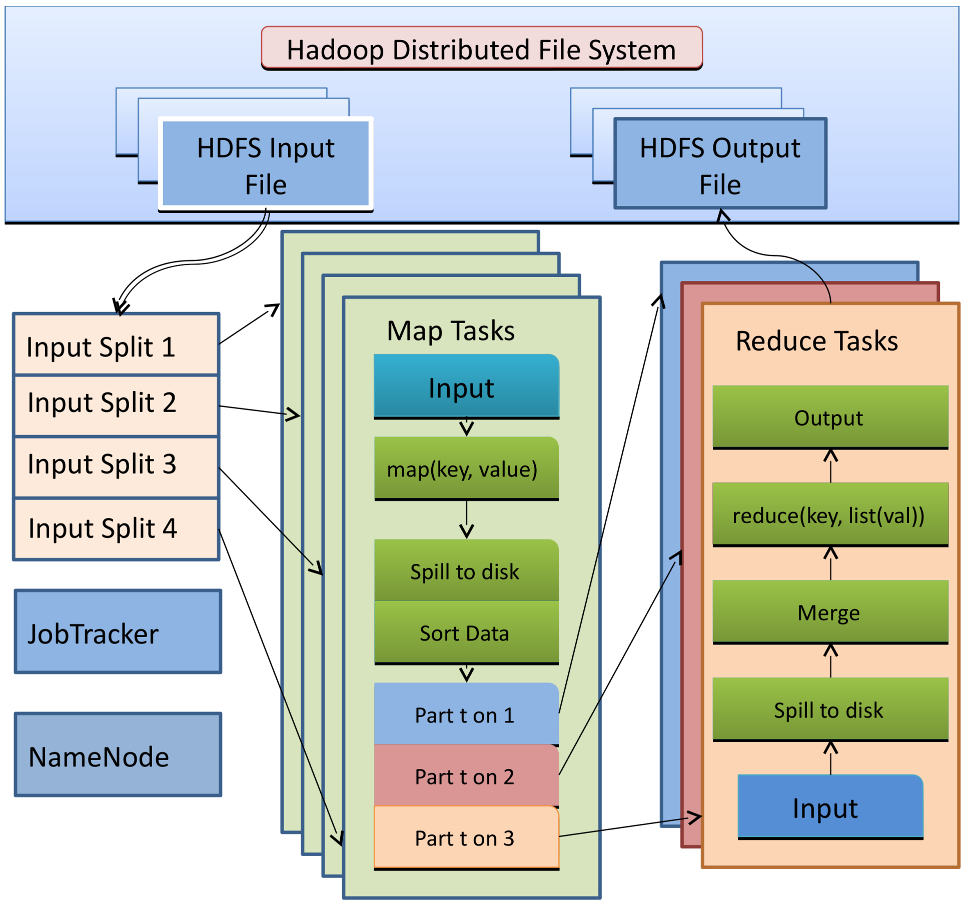


Figure 3.2: Hadoop MapReduce Data Flow

A Hadoop MapReduce cluster employs a master-slave architecture where one master node (known as JobTracker) manages a number of worker nodes (known as the TaskTrackers). Hadoop launches a MapReduce job by first splitting (logically) the input dataset into multiple data splits. Each map task is then scheduled to one TaskTracker node where the data split resides. A Task Scheduler is responsible for scheduling the execution of the tasks as far as possible in a data-local manner. A few different types of schedulers have been already developed for the MapReduce environment.

From a bird's eye view, a MapReduce job is not all that complex because Hadoop hides most of the complexity of writing parallel programs for a cluster of computers. In a Hadoop cluster, every node normally starts multiple map tasks (many times depending on the number of cores a machine has) and each task will read a portion of the input data in a sequence, process every row of the data and output a <key, value> pair. The reducer tasks in turn collect the keys and values outputted by the mapper tasks and merge the identical keys into one key and the different map values into a collection of values. An individual reducer then will work on these merged keys and the reducer task outputs data of its choosing by inspecting its input of keys and associated collection of values. The programmer needs to supply only the logic and code for the map() and the reduce() functions. This simple paradigm can solve surprisingly large types of computational problems and is a keystone of the Big Data processing revolution.

In a typical MapReduce job, input files are read from the Hadoop Distributed File System (HDFS). Data is usually compressed to reduce file sizes. After decompression, serialized bytes are transformed into Java objects before being passed to a user-defined map() function. Conversely, output records are serialized, compressed, and eventually pushed back to HDFS. However, behind this apparent simplicity, the processing is broken down into many steps and has hundreds of different tunable parameters to fine-tune the job's running characteristics. We detail these steps starting from when a job is started all the way up to when all the map and reduce tasks are complete and the JobTracker (JT) cleans up the job.

3.3.1 Startup Phase

Job submission is the starting point in the life of a MapReduce job. A Hadoop job is submitted on a single machine and the job needs to be aware of the addresses of the machines where the Hadoop NameNode (a single master daemon in the Hadoop cluster) and JobTracker daemons are running. The framework will first store any resources (e.g., java jar and configuration files) that must be distributed in HDFS. These are the resources provided via various parameters on the command-line arguments, as well as the JAR file indicated as the job JAR file. This step is executed on the local machine sequentially. The XML version of the job configuration data is also stored in HDFS. The framework will then examine the input data set, using the InputFormat class specified in the job setup to determine which input files must be passed whole to a task and which input files may be split across multiple tasks. The framework uses a variety of parameters to determine how map tasks need to be executed. InputFormat details may override a parameter like *$mapred.map.tasks* (which is a job parameter to signify the number of map tasks the user wants), for example a particular input format may force the splits to be made by line count.

In a Hadoop cluster, many times the number of slots for tasks is connected to the number of CPU cores available on the worker nodes but depending on the characteristic of a job, this may be different. The ideal split size is one file system block size or a multiple, as this allows the framework to attempt to provide data locally for the task that processes the split. The upshot of this process is a set of input splits that are each tagged with information about which machines have local copies of the split data. The splits are sorted in size order so that the largest splits are executed first. The split information and the job configuration information are passed to the JobTracker for execution via a job information file that is stored in HDFS.

A typical MapReduce job main class (WordCount as an example) snippet lines are shown below. The Job and its companions classes such as JobClient are responsible for submitting a MapReduce job. Having submitted the job, waitForCompletion() polls the job's progress once a second, and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

```
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs =
            new GenericOptionsParser(conf, args).getRemainingArgs
                ();
        if (otherArgs.length != 2) {
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]))
            ;
        FileOutputFormat.setOutputPath(job, new Path(otherArgs
            [1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

The Job and its companions classes such as JobClient are responsible for submitting a MapReduce job. Having submitted the job, waitForCompletion() polls the job's progress once a second, and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by the JobClient's submitJobInternal() method does the following in brief:

· Asks JobTracker for a new job ID (by calling getNewJobId() on JobTracker).

· Checks the output specification of the job. For example, an error is thrown and job submission is aborted if the output directory has not been specified or if it already exists.

· Computes the input splits for the job. In case of an error, the job is not submitted and an error is thrown to the MapReduce program.

· Copies the resources needed to run the job, including the job JAR file, the configuration file and the computed input splits, to the JobTracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the *$mapred.submit.replication* property, which defaults to 10) so that there are sufficient number of copies across the cluster for the TaskTrackers to access when they run tasks for the job.

· Tells the JobTracker that the job is ready for execution (by calling submitJob() on JobTracker).

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler picks it up and initializes it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the status of the job and its progress.

To create the list of tasks to run, the job tracker first retrieves the input splits computed by the JobClient from the shared file system. It then creates one map task for each split. The number of reduce tasks to create is determined by the *$ mapred.reduce.tasks* property in the JobConf object, which is set by the setNumReduceTasks() method, and the job tracker then simply creates this number of reduce tasks to be run. Tasks are given IDs at this time.

TaskTrackers run a simple loop that periodically sends heartbeat method calls to the JobTracker. There are as many TaskTrackers in a Hadoop cluster as the number of worker nodes (in the 100s or even 1000s). Heartbeats tell the JobTracker that a TaskTracker is alive, but they also double as a carrier for messages. As part of the heartbeat, a TaskTracker will indicate whether it is ready to run a new task, and if it is, the JobTracker will allocate a task, which it communicates to the TaskTracker using the heartbeat return value. The current architecture of a single JobTracker communicating with hundreds of TaskTrackers is a source of performance issues on huge Hadoop clusters today. Later in this section some alternative designs the Hadoop community is working on is described.

Now that the TaskTracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared file system to the TaskTracker's file system. It also copies any files needed from the distributed cache by the application to the local disk. Secondly, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Thirdly, it creates an instance of the TaskRunner object to run the task. The TaskTracker now

Algorithm 1 Hadoop FIFO Scheduler Pseudo Code

```
Input: taskTracker
Output: scheduledTasks
/* Fetch cluster totalMapCapacity, totalRunningMaps etc. */
taskTrackerStatus ← taskTracker.getStatus();
trackerMapCapacity ← taskTrackerStatus.getMaxMapSlots();
trackerRunningMaps ← taskTrackerStatus.getRunningMapCounts();
jobQueue ← getJobQueue();
availableMapSlots ← (trackerMapCapacity - trackerRunningMaps);
scheduledTasks ← ∅
scheduledMaps:
for $i = 0$ to availableMapSlots do
  /* Go thru each job in queue */
  for all job in jobQueue do
    if job.getStatus().getRunState() != JobStatus.RUNNING then
      continue;
    end if
  end for
  /* Try to schedule a node-local or rack-local Map task */
  Task t ← job.obtainNewLocalMapTask();
  if t != null then
    scheduledTasks.add(t);
    /* Break out after a ceiling is reached */
    if exceededMapPadding then
      break scheduledMaps;
    end if
  end if
  /* Try to schedule a non node-local or rack-local Map task */
  Task t ← job.obtainNewNonLocalMapTask();
  if t != null then
    scheduledTasks.add(t);
    /* Break out after a ceiling is reached */
    if exceededMapPadding then
      break scheduledMaps;
    end if
  end if
end for
```

launches a new Java Virtual Machine (JVM) to run each task so that any bugs in the user-defined map and reduce functions do not affect the TaskTracker (by causing it to crash or hang, for example). It is however possible to reuse the JVM between tasks. The child process communicates with its parent through the umbilical interface (previously mentioned heartbeat). This way it informs the parent of the task's progress every few seconds until the task is complete. The MapReduce framework can be configured with one or more job queues, depending on the scheduler it is running with. While some schedulers work with only one queue, others support multiple queues. Queues can be configured with various properties. Algorithm 1 depicts the main flow in pseudo code form the flow of Hadoop FIFO scheduler in deciding which map() tasks to run. Lines 4 through 7 set up the variables with the help of the TaskTracker object passed in. The main values the algorithm is interested in are the total map capacity of the TaskTracker, the total number of map tasks running, the jobs queue which will have to be inspected to find the task(s) to assign. Line 8 calculates the total number of map slots available on the TaskTracker. The scheduler then repeats the following steps for each of the available slots, only breaking out of the loop when there remains no more map tasks suitable for assigning. For each job which is in runnable state, it first attempts to find a nodel local map task in oder to assign task on the TaskTracker (if it cant find a node local tasks, it tries to find a non-local task), always making sure that all the map slots are not used up (line 23 and 33) - so as to not completely use up the available slots. The algorithm employs a similar decision making for choosing the reduce() tasks to schedule but we have omitted that in the interest of keeping the pseudo code simple.

The TaskStatus class declares the enumeration for the various statuses of a MapReduce job. The different phases are STARTING, MAP, SHUFFLE, SORT, REDUCE and CLEANUP.

Algorithm 2 below shows the main loop of the JobTracker. We have already discussed in some detail the STARTING phase of a job. We will now delve into the other phases.

Algorithm 2 JobTracker Main Flow

```
loop
  waitForMessage();
  if msg == HEARTBEAT then
    startMapTasks(); {Initialize Map Tasks}
    startReduceTasks(); {Initialize Reduce Tasks}
  else if msg == MAP_TASK_DONE then
    NotifyReduceTasks(); {Intermediate Data Fetch}
  else if msg == REDUCE_TASK_DONE then
    if allReduceTasksComplete then
      SignalJobComplete();
    end if
  else
    houseKeepingWork();
  end if
end loop
```

Figure 3.3 shows the calls between the different framework pieces when a MapReduce job is started with the client invocation up to the point a new JVM is created to run the task. The cleanup activities are not shown in this sequence diagram.

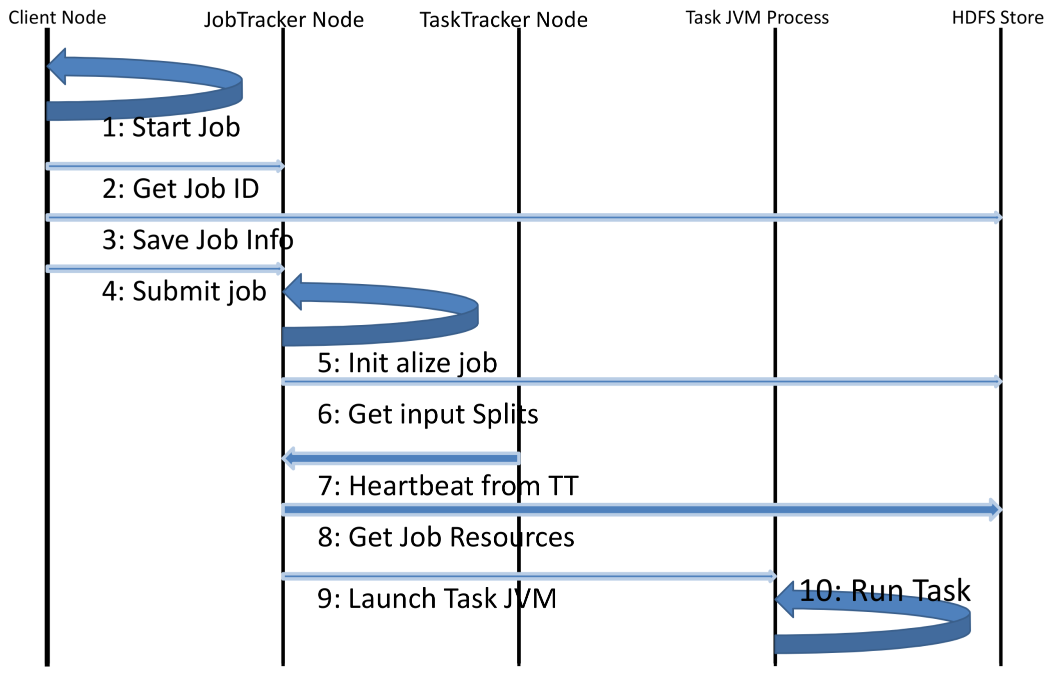


Figure 3.3: Call sequence for a Hadoop job

3.3.2 Map Phase

Let us consider the map phase in detail. A MapReduce job exists to read input data splits through a number of map tasks. The Job object created the information about all the splits and wrote a file with the split information and many other job related info so that the JobTracker has access to that file. The JobTracker is actually responsible for starting the tasks for the submitted job. Below we describe job input and job output in some detail.

InputFormat describes the input-specification for a MapReduce job. The MapReduce framework relies on the InputFormat of the job to

· Validate the input-specification of the job.

· Split-up the input file(s) into logical InputSplit instances, each of which is then assigned to an individual Mapper.

· Provide the RecordReader implementation used to create input records from the logical InputSplit for processing by the Mapper.

Most MapReduce programs implement their own InputFormat class but the default one is TextInputFormat that breaks files into lines. Keys are the position in the file, and values are the line of text (delimited by either line feed and/or carriage return). InputSplit represents the data to be processed by an individual Mapper. FileSplit is the default InputSplit. It sets *$map.input.file* to the path of the input file for the logical split. RecordReader reads <key, value> pairs from an InputSplit. Typically, the RecordReader converts the byte-oriented view of the input, provided by the InputSplit, and presents a record-oriented view to the mapper() implementations for processing. RecordReader thus assumes the responsibility of processing record boundaries and presents the tasks with keys and values appropriate for the domain.

OutputFormat describes the output-specification for a MapReduce job. The MapReduce framework relies on the OutputFormat of the job to:

· Validate the output-specification of the job; for example, check that the output directory does not already exist.

· Provide the RecordWriter implementation used to write the output files of the job. Output files are stored in the Hadoop FileSystem. TextOutputFormat is the default OutputFormat.

If the HDFS directory pointed to by the parameter *$mapreduce.job.dir* is listed while a job is in the process of executing on a Hadoop cluster, we would see the following files.

```
[mc2233]# hadoop fs -ls /tmp/hadoop/mapred/staging/
                condor/.staging/job_201303040610_0423
Found 4 items
```

```
job_201303040610_0423/job.jar
job_201303040610_0423/job.split
job_201303040610_0423/job.splitmetainfo
job_201303040610_0423/job.xml
```

The JobSplit class groups the fundamental classes associated with reading/writing splits. The split information is divided into two parts based on the consumer of the information. These two parts are the split meta information and the raw split information. The first part is consumed by the JobTracker to create the tasks' locality data structures. The second part is used by the maps at runtime to know how to setup the InputFormat/RecordReader so that data can be read and fed into the mapper() method. These pieces of information are written to two separate files as is shown above in the directory listing.

Any split implementation extends the base abstract class - InputSplit, defining a split length and locations of the split. A split length is the size of the split data (in bytes), while locations is the list of node names where the data for the split would be local. Split locations are a way for a scheduler to decide on which particular machine to execute this split.

Executing a map task on the node where its own data exists is an important aspect since that means data does not have to be transported across the network. This type of map task is called a data-local map task. Locality can mean different things depending on storage mechanisms and the overall execution strategy. In the case of HDFS, a split typically corresponds to a physical data block size and locations on a set of machines (with the set size defined by a replication factor) where this block is physically located.

Now we pick up at the point where TaskTracker has started a child process (a class called Child) which has a main() method. This class is responsible for starting the map task. Not surprisingly, the name of the class that encapsulates the map task is called MapTask (it extends Task as does the class ReduceTask), which handles all aspects of running a map for the job under consideration. While a map is running, it is collecting output records in an in-memory buffer called MapOutputBuffer. If there are no reducers, a DirectMapOutputCollector is used, which makes writes immediately to disk. The total size of this in-memory buffer is set by the *$io.sort.mb* property and defaults to 100 MB. Out of these 100 MB, *$io.sort.record.percent* are reserved for tracking record boundaries. This property defaults to 0.05 (i.e. 5% which means 5 MB in the default case) Each record to track takes 16 bytes (4 integers of 4 bytes each) of memory which means that the buffer can track 327680 map output records with the default settings. The rest of the memory (104,857,600 bytes - (16 bytes $\times$ 327,680) = 99,614,720 bytes) is used to store the actual bytes to be collected (in the default case this will be about 95 MB). While map outputs are collected, they are stored in the remaining memory and their location in the in-memory buffer is tracked as well. Once one of these two buffers reaches a threshold specified by *$io.sort.spill.percent*, which defaults to 0.8 (i.e., 80%), the buffer is flushed to disk. For the actual data this value is 79,691,776 (0.8 $\times$ 99,614,720) and for the tracking data the threshold is 262,144 (0.8 $\times$ 327,680).

All of this spilling to disk is done in a separate thread so that the map can continue running. That is also the reason why the spill begins early (when the buffer is only 80% full) so it does not fill up before a spill is finished. If one single map output is too large to fit into the in-memory buffer, a single spill is done for this one value. A spill actually consists of one file per partition, meaning one file per reducer.

After a map task has finished, there may be multiple spills on the TaskTracker. Those files have to be merged into one single sorted file per partition which is then fetched by the reducers. The property *$io.sort.factor* indicates how many of those spill files will be merged into one file at a time. The lower the number the more passes will be required to arrive at the goal. The default is set to 100. This property can make a pretty huge difference if the mappers output a lot of data. Not much memory is needed for this property but the larger it is the more open files there will be. To determine such a value, a few MapReduce jobs expected in production should be run and one should carefully monitor the log files.

So, at this point the input file was split and read, the mapper() function was invoked for each <key, value> pair. The mapper(), after processing each pair, outputted another <key, value> pair which was collected by the framework in a memory buffer, sorted and partitioned. If memory got full, then they were spilled to the disk and merged into files. The number of files is the same as the number of different partitions (number of reducers). Now, the reducer tasks are coming into action and we will see how reducer tasks will obtain the map output data.

3.3.3 Shuffle Phase

The shuffle phase is one of the steps leading up to the reduce phase of a MapReduce job. The main actors of the shuffle phase are the reducer tasks and the TaskTrackers, which are holding on to the map output files. In other words, a map output file is stored on the local disk of the TaskTracker that ran the map task (note that although map outputs are always written to the local disk of the map TaskTracker, reduce outputs may not be), but now the address of this TaskTracker that finished the map task is needed by the TaskTracker that is about to run the reduce task for the partition. Furthermore, the reduce task needs the map output for its particular partition from several map tasks across the cluster. The map tasks may finish at different times, so the reduce tasks start copying their outputs (also called shuffling the output) as soon as each completes. This is also known as the copy phase of the reduce task. The reduce task has a small number of copier threads (5 by default) so that it can fetch map outputs in parallel. Though the default is 5 threads, this number can be changed by setting the *$mapred.reduce.parallel.copies* property.

Since the reducers have to know which TaskTrackers to fetch map output from, as map tasks complete successfully, they notify their parent TaskTracker of the status update, which in turn notifies the JobTracker. These notifications are transmitted over the heartbeat communication mechanism described earlier. Therefore, for a given job, the JobTracker knows the mapping between map outputs and TaskTrackers. A thread in the reducer periodically asks the JobTracker for map output locations until it has retrieved them all. TaskTrackers do not delete map outputs from disk as soon as the first reducer has retrieved them, as the reducer tasks may fail. Instead, they wait until they are told to delete them by the JobTracker, which is after the job has completed.

The map outputs then are copied to the reducer's memory if they are small enough (the buffer size is controlled by *$ mapred.job.shuffle.input.buffer.percent*, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size (controlled by *$mapred.job.shuffle.merge.percent*), or reaches a threshold number of map outputs (*$mapred.inmem.merge.threshold*), it is merged and spilled to disk.

3.3.4 Reduce Phase

Let us turn now to the reduce part of the process. As the copies accumulate on disk, a background thread merges them into larger sorted files. This saves some time merging later on in the process. Note that any map outputs that were compressed (by the map task) have to be decompressed in memory in order to perform a merge on them. When all the map outputs have been copied, the reduce task moves into the sort phase (which should actually be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 100 map outputs, and the merge factor was ten (the default, controlled by the *$io.sort.factor* property, just like in the map's merge), then there would be 10 rounds. Each round would merge ten files into one, so at the end there would be 10 intermediate files.

Rather than have a final round that merges these ten files into a single sorted file, the merge saves a trip to disk and additional I/O by directly feeding the reduce function in what is the last phase: the reduce phase. This final merge can come from a mixture of in-memory and on-disk segments. The number of files merged in each round is actually more subtle than the above explanation suggests. The goal is to merge the minimum number of files to get to the merge factor for the final round.

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output file system, typically HDFS. In the case of HDFS, since the TaskTracker node is also running a DataNode, the first block replica will be written to the local disk.

Among the many steps described above in starting and running of a MapReduce program, some of the responsibilities lie with the programmer of the job. Hadoop provides the remaining facilities. Most of the heavy lifting is done by the framework and Fig. 3.4 shows the different steps performed by the users and how they map to the various steps provided by the framework itself.

3.3.5 Shutdown Phase

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours or even several days to run. Because this is a significant length of time, it is important for the user to obtain feedback

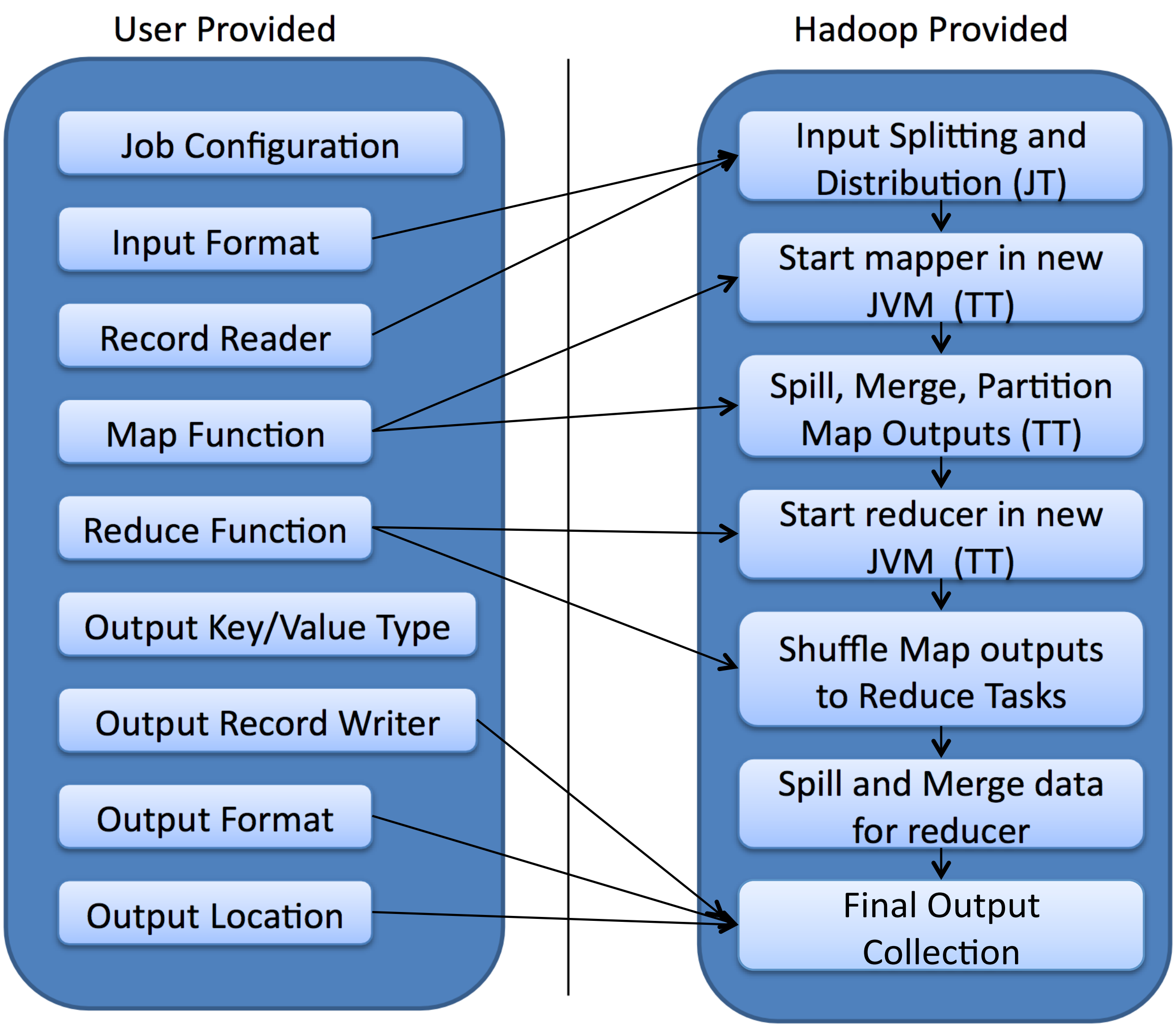

User Provided
Hadoop Provided
Job Configuration
Input Format
Record Reader
Map Function
Reduce Function
Output Key/Value Type
Output Record Writer
Output Format
Output Location
Input Splitting and Distribution (JT)
Start mapper in new JVM (TT)
Spill, Merge, Partition Map Outputs (TT)
Start reducer in new JVM (TT)
Shuffle Map outputs to Reduce Tasks
Spill and Merge data for reducer
Final Output Collection

Figure 3.4: MapReduce Steps - Who does what

on how the job is progressing. A job and each of its tasks have a status, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code). These statuses change over the course of the job and they are communicated back to the client. When a task is running, it keeps track of its progress, that is, the proportion of the task completed. For map tasks, this is the proportion of the input that has been processed. For reduce tasks the system can estimate the proportion of the reduce input processed. It does this by dividing the total progress into three parts, corresponding to the three phases of the reducer.

Measurement of progress is somewhat nebulous but it tells Hadoop that a task is doing something. For example, a task writing output records is making progress, even though it cannot be expressed as a percentage of the total number that will be written, since the latter figure may not be known, even by the task producing the output.

Progress reporting is important so that Hadoop does not fail a task that is making progress. All of the following operations constitute progress:

· Reading an input record (in a mapper or reducer)

· Writing an output record (in a mapper or reducer)

· Setting the status description on a reporter (using the Reporter's setStatus() method)

· Incrementing a counter (using the Reporter's incrCounter() method)

· Calling the Reporter's progress() method

Tasks also have a set of counters that count various events as a task runs, either those built into the framework, such as the number of map output records written, or counters defined by users. If a task reports progress, it sets a flag to indicate that the status change should be sent to the TaskTracker. The flag is checked in a separate thread every few seconds, and if set it notifies the TaskTracker of the current task status. Meanwhile, the TaskTracker is sending heartbeats to the JobTracker every five seconds (this is a minimum, as the heartbeat interval is actually dependent on the size of the cluster: for larger clusters, the interval is longer and a source of much consternation since this value can delay the time to start up a job's task(s)), and the status of all the tasks being run by the TaskTracker is sent in the call. Counters are sent less frequently than every five seconds, because they can be relatively high-bandwidth. The JobTracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks. Finally, the JobClient receives the latest status by polling the JobTracker every second. Clients can also use JobClient's getJob() method to obtain a RunningJob object instance, which contains all of the status information for the job.

When the JobTracker receives a notification that the last task of a job is complete, it changes the status of the job to "successful". Then, when the JobClient polls for status, the JobClient learns that the job has completed successfully and it prints a message to tell the user and then returns from the runJob() method. The JobTracker also sends a HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notification.url property. Lastly, the JobTracker cleans up its working state for the job, and instructs TaskTrackers to do the same (the intermediate output is deleted and other such cleanup tasks are performed).

3.4 MapReduce Scheduling Challenges

Hundreds of jobs (small/medium/large) may be present on a Hadoop cluster for processing at any given time. How the map and reduce tasks of these jobs are scheduled has an impact on the completion time and consequently on the QoS requirements of these jobs. Hadoop uses a FIFO scheduler out of the box. Subsequently, two more schedulers have been developed. Firstly, Facebook developed the Fair Scheduler which is meant to give fast response time for small jobs and reasonable finish time for production jobs. Secondly, Capacity Scheduler was developed by Yahoo and this scheduler has named queues in which jobs are submitted. Queues are allocated a fraction of the total computing resource and jobs have priorities. It is evident that no single scheduling algorithm and policy is appropriate for all kinds of job mixes. A mix or combination of scheduling algorithms may actually be better depending on the workload characteristics.

Hadoop processes multiple data-sets for in a multi-tenant environment. At the inception of Hadoop, five or so years ago, the original scheduler was a first-in first-out (FIFO) scheduler woven into Hadoop's JobTracker. Even though it was simple, the implementation was inflexible and could not be tailored. After all, not all jobs have the same priority and a higher priority job is not supposed to languish behind a low priority long running batch job. Around 2008, Hadoop introduced a pluggable scheduler interface which was independent of the JobTracker. The goal was to develop new schedulers which would help optimize scheduling based on particular job characteristics. Another advantage of this pluggable scheduler architecture is that now greater experimentation is possible and specialized schedulers are possible to cater to an ever increasing types of Hadoop MapReduce applications.

Before it can choose a task for the TaskTracker, the JobTracker must choose a job to select the task from. Having chosen a job, the JobTracker now chooses a task for the job. TaskTrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a TaskTracker may be able to run two map tasks and two reduce tasks simultaneously. The default scheduler fills empty map task slots before reduce task slots, so if the TaskTracker has at least one empty map task slot, the JobTracker will select a map task; otherwise, it will select a reduce task.

Based on published research [75], it is very evident that a single scheduler is not adequate to obtain the best possible QoS out of a Hadoop MapReduce cluster subject to a varying workload. Also, we have seen that almost none of the schedulers consider the contention placed on the nodes due to the running of multiple tasks.

3.5 MapReduce Performance Challenges

Tens of thousands of jobs of varying demands on CPU, I/O and network are executed on Hadoop clusters consisting of several hundred nodes. Tasks are scheduled on machines, in many cases with 16 or more cores each. Short jobs have a different completion time requirement than long jobs and similarly production level high priority jobs have a different quality of service requirement compared to ad-hoc query type jobs. Predicting the completion time of Hadoop MapReduce jobs is an important research topic since for large enterprises, correctly forecasting the completion time and an efficient scheduling of Hadoop jobs directly affects the bottom line. A plethora of work is going on in the field of Hadoop MapReduce performance. We briefly talk about a few prominent recent ones which are most relevant to this dissertation. In his 2011 technical report [37], Herodotou describes in detail a set of mathematical performance models for describing the execution of a MapReduce job on Hadoop. The models can be used to estimate the performance of MapReduce jobs as well as to find the optimal configuration settings to use when running the jobs. A set of 100 or so equations calculate the total cost of a MapReduce job based on different categories of parameters - Hadoop parameters, job profile statistics and a set of parameters that define the I/O, CPU, and network cost of a job execution. In a 2010 paper, Kavulya et al. [42] analysed ten months worth of Yahoo MapReduce logs and used an instance-based learning technique that exploits temporal locality to predict job completion times from historical data. Though the thrust of the paper is essentially analyzing the log traces of Yahoo, this paper extends the instance based (nearest-neighbor) learning approaches. The algorithm consists of first using a distance-based approach to find a set of similar jobs in the recent past, and then generating regression models that predict the completion time of the incoming job from the Hadoop-specific input parameters listed in the set of similar jobs. In a 2010 paper [75], Verma et al. design a performance model that for a given job (with a known profile) and its SLO (soft deadline), estimates the amount of resources required for job completion within the deadline. They also implement a Service Level Objective based scheduler that determines job ordering and the amount of resources to allocate for meeting the job deadlines.

3.6 Analytic Model

Now we describe the analytic model built to predict a MapReduce job's map phase completion time. We take into account the effect of contention at the compute nodes of a cluster and use a closed Queuing Network model [55] to estimate the completion time of a job's map phase based on the number of map tasks, the number of compute nodes, the number of slots on each node, and the total number of map slots available on the cluster. As discussed in the previous section, various different kinds of MapReduce jobs are executed in an enterprise and the execution environment varies too between clusters with only a few nodes to hundreds of nodes.

Let M be the number of map tasks of a job and let n be the total number of map slots in the worker nodes. Let c be the number of compute nodes. We assume in the following discussion that the n slots are equally distributed among the c compute nodes and that $n = k \times c$, i.e., each compute node has k slots.

The first n maps are executed in parallel using k slots in each of the c compute nodes. Then, the next n maps are executed in that fashion until the remaining n' map tasks are executed. The number of times that n tasks are executed in parallel is $\lfloor M/n \rfloor$ and $n' = \text{mod}(M, n) = M - \lfloor M/n \rfloor \times n$. For example, consider that a Hadoop job has 54 map tasks and that there are 8 slots in 4 compute nodes (each node has 2 slots). Then, there are 6 waves of 8 parallel map tasks followed by a wave of 6 parallel tasks. Thus, in this example, $M = 54, n = 8, c = 4, k = 2, \lfloor M/n \rfloor = 6$ and $n' = \text{mod}(54, 8) = 6$.

It turns out that all map tasks that execute in a compute node compete for the computational resources (e.g., CPU, disks, network bandwidth, etc.) of that node. Therefore, the execution times of map tasks is elongated by the contention (i.e., queuing for resources) generated by the concurrent execution of tasks at a compute node. We address this issue by using Mean Value Analysis (MVA) to solve the closed Queuing Network (QN) model that captures all computational resources of a node and its workload of tasks [55]. Let $T_{\text{MVA}}(s)$ be the execution time of tasks at a compute node when there are s tasks executing at the same time. This time is obtained by solving the QN that corresponds to a computational node. We are assuming that a compute node executes tasks of the same job and therefore they are similar. To extend this to a more general situation we need to use well-known multiclass closed QNs [55].

We can now compute the total execution time, $T(M, n, c)$, of the map phase of a Hadoop job as

$$T(M, n, c) = T_{\text{MVA}}(n/c) \times \lfloor M/n \rfloor + T_{\text{MVA}}(mod\ (M, n)/c) \tag{3.1}$$

The first term in equation (3.1) accounts for the first waves in which n tasks are executed in parallel. The second term accounts for the time needed to execute the balance of the tasks.

We observed in the experiments that the number of concurrent map tasks in execution is slightly lower than the number of slots in a compute node. This is due to the fact that it takes some time for the TaskTracker to detect the completion of a task, notify the JobTracker of this fact through a heartbeat message, obtain the response from the JobTracker with a request to start the following task, and lastly spawn a Java VM for a map task. We than observed that there is a linear relationship of the form

$$\text{MapTaskConcurrency} = \gamma \times \text{NumSlots} \tag{3.2}$$

as exemplified in Fig. 3.5.

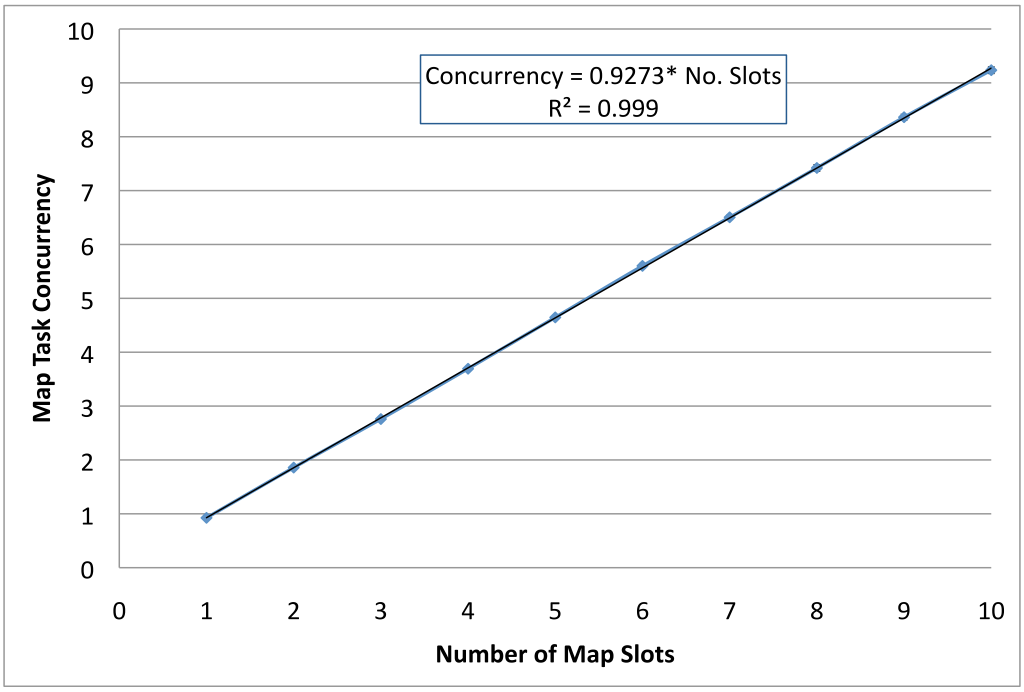


Figure 3.5: Concurrency of Map Tasks vs. No. Slots for 54 Map Tasks.

The slope γ in equation 3.2 is then used to adjust the value of the execution time $T(M, n, c)$. The adjusted execution time then becomes $T(M, n, c)/\gamma$.

If contention at compute nodes is not considered, the job execution time, T_{NoCont} would be

$$T_{\text{NoCont}}(M, n) = T_m \times \lfloor M/n \rfloor + T_m \times \epsilon(mod\ (M, n)) \tag{3.3}$$

where T_m is the execution time of a map task under no contention and $\epsilon(x)$ returns 1 if $x > 0$ and 0 if $x = 0$.

3.7 Workload and Execution Environment

3.7.1 Execution Environment

We used two different Hadoop configurations for our experiments. The first is a single-node configuration with an i5 quad-core CPU machine with 6 GB of memory. We ran all the Hadoop daemons (NameNode, SecondaryNameNode, JobTracker, DataNode, TaskTracker) and all the tasks on this machine. The daemons communicate amongst themselves to schedule jobs through the JobTracker and TaskTracker. In the 2-node configuration, we ran the single daemon components (NameNode, SecondaryNameNode and JobTracker) on a dual-core CPU with 4 GB of memory. The rest of the daemons (DataNode and TaskTracker) ran on the machine with the quad-core CPU with 6 GB of memory. MapReduce job tasks (i.e., maps) ran alongside TaskTrackers. Thus, in the first case, all the tasks ran on one machine alongside all the Hadoop daemons and in the second case the tasks ran on the second machine, which we call the Hadoop compute node. The jobs we used in our experiments do not demand significant network bandwidth. Moreover, the input file and the tasks that read the data are co-located. Therefore, we have not considered network contention and its effect in our model. For a full fledged job, running on several compute nodes where the shuffle phase (the intermediate phase between map and reduce) could potentially transfer large amounts of data, we need to pay attention to the contention on the cluster brought about by network traffic.

We briefly describe the steps for installing Hadoop on a 2-node cluster with Cloudera's [93] Hadoop distribution running Ubuntu Linux version 11.10. After receiving the tarball (tarred files) from the download website, we simply extracted it to a directory. We used the same directory on both machines to keep matters simple. We called our first machine 'Master' and the second machine 'Worker'. Hadoop has 5 important configuration files among others. They all reside in the `conf` directory of the standard distribution. The file named 'masters' defines on which machines Hadoop will start SecondaryNameNode in our multi-node cluster. In our case, this is just the Master machine. The primary NameNode and the JobTracker will also be running on this machine. The second file, 'slaves', lists the hosts, one per line, where the Hadoop slave daemons (DataNodes and TaskTrackers) will be run. We wanted only the Worker box to act as Hadoop slave because we wanted to collect performance and timing data on only one processing node. Next we consider the

3 XML configuration files - core-site.xml, hdfs-site.xml and mapred-site.xml. The first two had standard entries in our experiment, but mapred-site.xml had two entries of particular interest. The first is a property called “mapred.tasktracker.map.tasks.maximum”. This determines the number of concurrent map tasks that the JobTracker will schedule on this particular machine. We varied this value between our experimental runs to see the effect of the number of slots on the completion time of our job. The second parameter of interest is “mapred.child.java.opts”. We set this value to 400 MB to give each of our map tasks a maximum of 400 MB of Java heap space. Since our goal for the experiment was to model the CPU and I/O usage, our job did not have a high memory demand and thus we were able to run a high number of map tasks without having to use swap space on the Worker machine.

We start all the Hadoop daemons from the Master box with one command: `bin/start-all.sh`. This invokes several scripts starting NameNode, SecondaryNameNode and JobTracker on the Master machine. The scripts also SSH into the Worker machine (in clusters with hundreds of machines it does SSH to all the Slave nodes), and starts the DataNode and TaskTracker daemons. Hadoop provides a convenient way to monitor the entire cluster with web pages so that one can monitor the health of the Hadoop file system and also the progress of the MapReduce jobs that are started on the system. Once we were sure that the environment was fully up, we copied our input file with the following command.

bin/Hadoop -fs copyFromLocal /local/dir/inputFile /apps/common/in1

This has the effect of copying the POSIX file inputFile into the distributed file systems of Hadoop. In our case, with one DataNode, our 5-MB input file has a replication factor of 1 and is copied only to the Worker node. However, for a very large file, on a cluster with hundreds of DataNodes (Worker machines), the file will be split into many pieces and stored on many boxes along with their replicated copies. Once we copied the file, we were ready to run our job with the following command.

*bin/Hadoop jar Hadoop*example*.jar CpuDIskLoad*

Our goal was to measure the CPU and Disk utilization of the Worker machine for a varying number of map slots. We describe the workload and the interplay with the number of slots in more detail in the next subsection. All measurements were performed on the Worker machine with the help of Unix `iostat` command. Data was collected for both CPU and Disk at an interval of 5 sec with the command `iostat -x -t 5`. We wrote several small utility programs to calculate the average utilization from the `iostat` output file. Also, we analyzed the Hadoop MapReduce userlogs to determine the completion time for each task.

3.7.2 Workload Characteristics

Our goal for this work was to create a model for only the map tasks of a MapReduce job. A standard MapReduce job consists additionally of the shuffle and reduce phases.

We first show below the relevant parts of our Job configuration code. The MapperClass property of the Job object is set to the TokenizeMapper class and it is shown in the next listing. The number of reduce tasks is set to 0 (setNumReduceTasks(0)) so that we deal with only the map phase. The input file is a Hadoop DFS file and is stored and read from /apps/common/in1. The last line of the listing shows that we set the setMaxInputSplitSize anywhere between 100K and 600K. Since our input file size is 5MB, the varying input split size will determine the number of map tasks created to complete this job. As an example, when the split size is 200 K, we get 26 map tasks, which brings the total data handled to around 5 MB (the input file size).

```
Job Configuration Method

Job job = new Job(conf, "word count");
job.setJarByClass(LoadCreator.class);
job.setMapperClass(TokenizerMapper.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setNumReduceTasks(0);
FileInputFormat.addInputPath(job,
    new Path("/apps/common/in1"));
FileInputFormat.setMaxInputSplitSize(job,
    200000); // 100K to 600K
```

Next is the listing of the map method of our MapReduce job. With every invocation of the map method, it gets a line from the input file. We do not do any processing with that input though. Our goal is to create CPU and disk load. We do that by calling the generateSentence() method, which in this case creates and returns a Java String (Text object is just a Hadoop String wrapper for our purposes) with 3,000 words separated with spaces. Once we get the two Text objects `keyWords` and `valueWords`, we then use the context object to write this key/value pair out in a file which produces load on the disk. The CPU load is also generated as a result of work done by the code plus due to the act of TaskTracker starting up and shutting down the task JVMs and the communication between and by various Hadoop daemons.

Once we had the code below, we went and modified Hadoop's example driver to add this class as a new test class and then we were able to build Hadoop example jar and execute with the standard `bin/Hadoop jar` command.

```
public void map(Object key, Text value,
    Context context) throws ... {
  StringTokenizer itr =
   new StringTokenizer(value.toString());
  Text keyWords=generateSentence(3000);
  Text valueWords=generateSentence(3000);
  context.write(keyWords, valueWords);
}

private Text generateSentence(int
    noWords) {
  StringBuffer sentence =
    new StringBuffer();
  String space = " ";
  for (int i = 0; i < noWords; ++i) {
    sentence.append("arbitraryword");
    sentence.append(space);
  }
  return new Text(sentence.toString());
}
```

3.8 Experimentation Results and Model Validation

We first provide some results obtained from the experimental setup described in previous sections. Figure 3.6 shows the variation of the execution of our Hadoop job with 54 map tasks as a function of the number of slots. The figure also shows the 95% confidence intervals for the experimental measurements. As it can be seen, the execution time initially decreases as more slots are available, which increases the parallelism. However, after a certain number of slots (4 slots in the figure), the execution time starts to increase because the effect of the increase in contention at the compute node exceeds the benefits of additional parallelism.

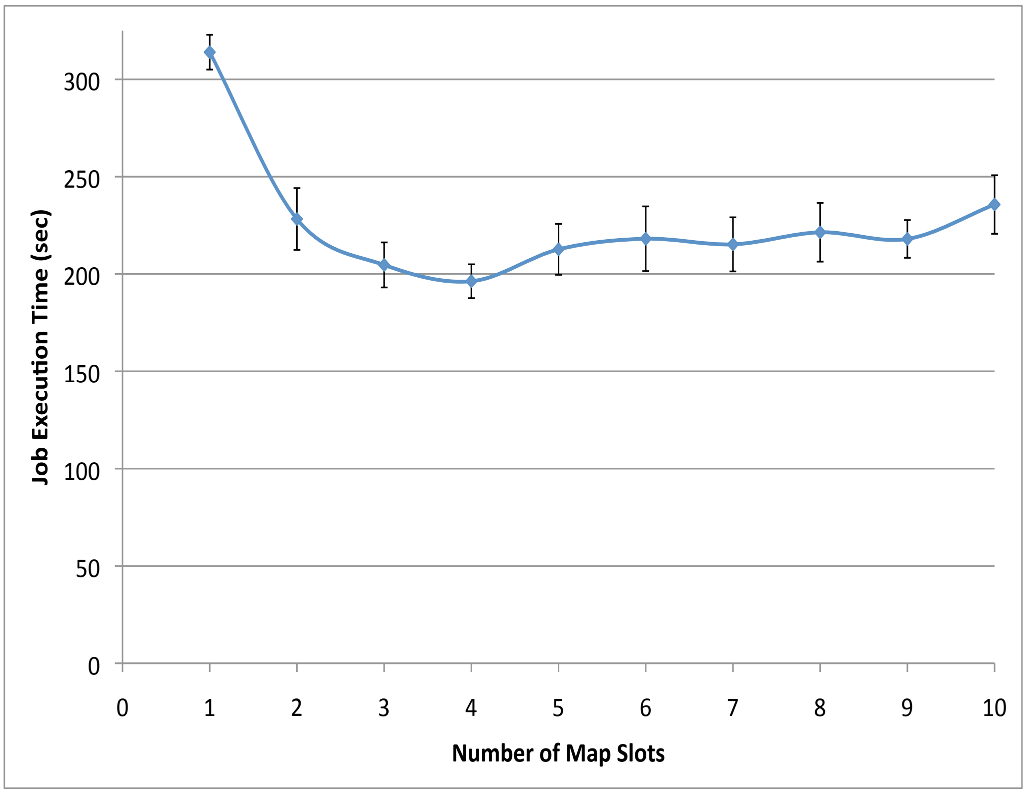


Figure 3.6: Job Execution Time (in sec) vs. No. Slots for 54 Map Tasks.

Figure 3.7 shows the variation of the job execution time as a function of the number of map slots for three different values of the number of map tasks: 11, 28, and 54. For each value of the number of map tasks we observe a similar behavior as in Fig. 3.6; the effect is more pronounced for 54 map tasks though. We also observe that there is not much difference between 11 and 28 map tasks but the execution time for 54 map tasks is significantly higher than for 11 and 28 map tasks. Note that in the 54-map case, the TaskTracker has to create and stop a much larger number of JVMs, which increases the completion time.

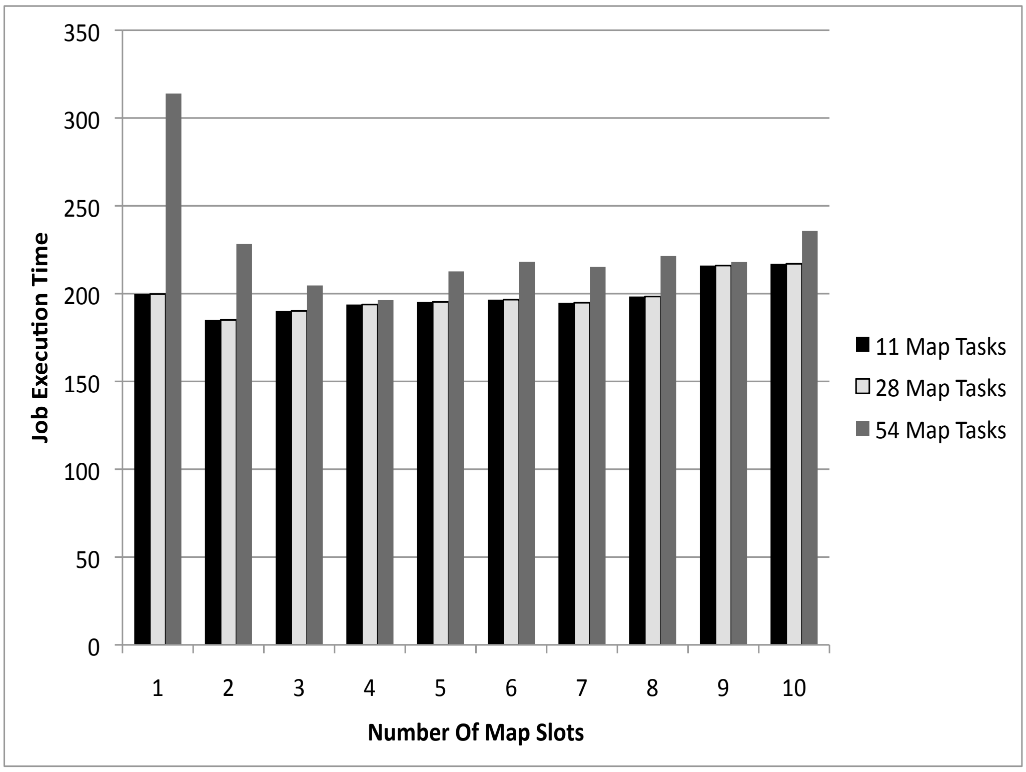


Figure 3.7: Job Execution Time (in sec) vs. No. Slots for 54, 28, and 11 Map Tasks.

In order to validate the model, we compared the values obtained from experimentation with the analytic model values. First, we had to obtain the service demands to be used as parameters for the closed QN model. As indicated below, the compute node uses a quad-core CPU. The machine also had a single disk. Thus, to obtain the service demand values, we ran experiments with a single slot configuration (to eliminate contention) and computed the service demands for the map tasks using the Service Demand Law ($D_i = U_i/X_0$; i.e., the service demand at resource i is equal to its utilization divided by the system throughput) [55].

Since our experiments were all conducted on multi-core machines, we use an approximation proposed by Seidmann et al. [65] to model multiprocessors. The basic idea is that a single queue with m servers and a service demand value of D at each server can be replaced with two queues in series. The first is a single-server load independent queue with service demand equal to D/m, i.e., with a server that works m times faster than any of the original servers. The second queue in the series is a delay server, where no queuing takes place. The service demand for the delay server is equal to $D(m-1)/m$ [57]. Experimentation has shown that the error due to this approximation is between 5% and 11% depending on whether the load is light or heavy.

Figure 3.8 shows a comparison between the job execution time obtained from experiments, from our predictive model, and from a model where no contention at the compute nodes is considered. As it can be seen, the predictive model tracks reasonably well the measurements while a basic model that does not consider contention (i.e., T_{NoCont}) and only considers gains due to increased parallelism and does not capture the negative effect of contention.

3.9 MapReduce 2.0

The discussion in the preceding sections was all based on current Hadoop (also known as Hadoop V1.0). In clusters consisting of thousands of nodes, having a single JobTracker manage all the TaskTrackers and the entire set of jobs has proved to be a big bottleneck. The architecture of one JobTracker introduces inherent delays in scheduling jobs which proved to be unacceptable for many large Hadoop users.

3.9.1 Facebook's Corona

The bulk of the workload on the Hadoop clusters in Facebook are Hive [98] queries. These queries translate into small Hadoop jobs and it is important that the latency experienced by these jobs be small. Therefore, small job latency is crucial for Facebook's environment. Facebook noticed that the single JobTracker is the bottleneck and many times the startup time for a job is several tens

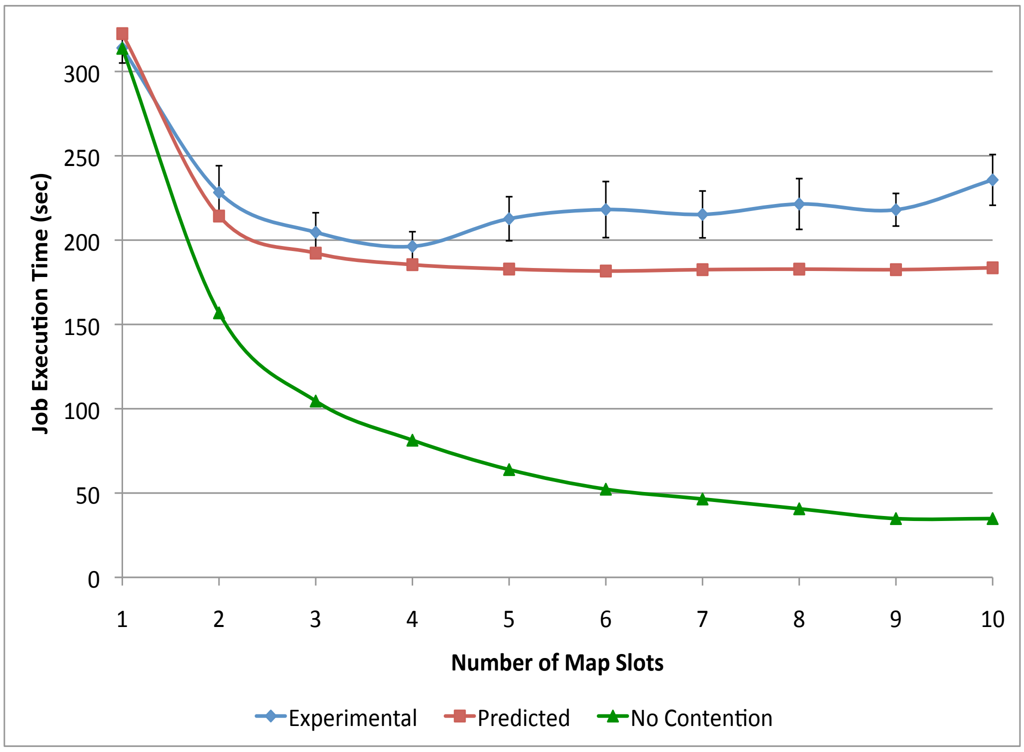


Figure 3.8: Job Execution Time (in sec) vs. No. Slots for 54 Tasks: Experimental, Predicted using MVA , and No Contention.

of seconds [95]. The JobTracker also could not handle its dual responsibilities of managing the cluster resources and the scheduling of all the user jobs adequately. At peak load at Facebook, cluster utilization dropped precipitously due to scheduling overhead using the standard Hadoop framework [95]. Along with the problems described above, Facebook also found that the polling from TaskTrackers to the JobTracker during a periodic heartbeat is a big part of why Hadoop's scheduler is slow and has scalability issues [95]. And lastly, the slot based model is a problem when the slot configuration does not match the job mix. As a result, during upgrades to change the number of slots on node(s), all jobs were killed, which was unacceptable.

Facebook decided that they needed a better scheduling framework that would improve this situation by bringing better scalability and cluster utilization, by lowering latency for small jobs and by scheduling based on actual task resource requirements rather than a count of map and reduce tasks [114].

To address these issues, Facebook developed Corona [95], a new scheduling framework that separates cluster resource management from job coordination. Corona introduces a cluster manager whose only purpose is to track the nodes in the cluster and the amount of free resources. A dedicated job tracker is created for each job, and can run either in the same process as the client (for small jobs) or as a separate process in the cluster (for large jobs). One major difference from the previous Hadoop MapReduce implementation is that Corona uses push-based, rather than pull-based, scheduling. After the cluster manager receives resource requests from the job tracker, it pushes the resource grants back to the job tracker. Also, once the job tracker gets resource grants, it creates tasks and then pushes these tasks to the task trackers for running. There is no periodic heartbeat involved in this scheduling, so the scheduling latency is minimized.

The cluster manager does not perform any tracking or monitoring of the job's progress, as that responsibility is left to the individual job trackers. This separation of duties allows the cluster manager to focus on making fast scheduling decisions. Job trackers track only one job each and have less code complexity. This allows Corona to manage a lot more jobs and achieve better cluster utilization. Figure 3.9 depicts a notional diagram of Corona's main components.

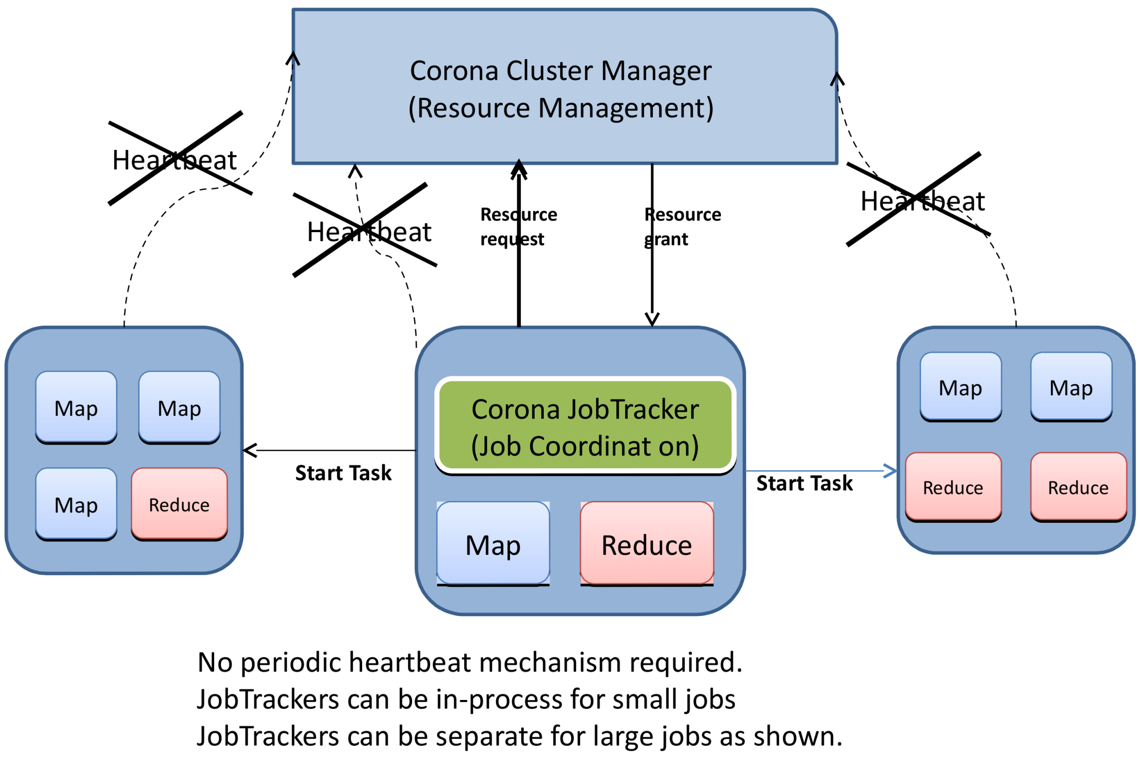


Figure 3.9: Corona - A new Hadoop

Once they developed Corona, Facebook measured some of these improvements with several key metrics:

Average time to refill slot - This metric gauges the amount of time a map or reduce slot remains idle on a TaskTracker. Given that the scalability of the scheduler is a bottleneck, improvement was observed with Corona when compared with a similar period for MapReduce. Corona showed an improvement of approximately 17% in a benchmark run in the simulation cluster.

Cluster utilization - Cluster utilization also improved along with the refill-slot metric. In heavy workloads during simulation cluster testing, the utilization in the Hadoop MapReduce system topped out at 70%. Corona was able to reach more than 95%.

Scheduling fairness - Resource scheduling fairness is important since it ensures that each of the pools actually gets the resources it expects. The old Hadoop MapReduce system is found to be unfair in its allocation, and a dramatic improvement was seen with the new system.

Job latency - Average job latencies were expected to go down just by virtue of the new design in Corona. The best metric for measuring this was found to be a latency job that was run every four minutes. It was seen that the latency of the test job was cut in half (25 seconds, down from 50 seconds).

3.9.2 Yahoo's YARN

Yahoo has been a major player in the Hadoop ecosystem already. MapReduce version-2 was conceived and designed by Yahoo when it was found that JobTracker is a bottleneck in their huge Hadoop clusters. Current Hadoop has indeed reached a scalability limit of around 4,000 nodes. Scalability requirement of 20,000 nodes and 200,000 cores is being asked for and was not possible with the current JobTracker/TaskTracker architecture. YARN [113] stands for Yet Another Resource Negotiator and the next generation of Hadoop (0.23 and beyond, most places are running 0.20 or 0.21 today) and is also called MapReduce2.0 or MRv2. YARN breaks the function of the JobTracker into 2 parts (see Fig. 3.10 for a block level diagram of YARN's main components). The job life cycle management portion of JobTracker will be handled by Resource Manager (RM) in YARN clusters which will be responsible for assignment of compute resources to applications and the job life-cycle management portion of the responsibility will be handled per application by the Application Master (AM). The AM is created and lives for the duration of the application (for MapReduce and other kinds of applications) and then exits. There is also a per machine Node Manager (NM) daemon that manages the user processes on the machine. Resource Manager therefore has 2 parts - Applications Manager (RM-ASM) and Scheduler (RM-S). Scheduler is a plugin. Resource Manager talks to Node Manager and Applications Master. It looks like being a MapReduce platform is not the only goal of YARN. Running applications of different kind other than current MapReduce seems to be one of the motivations for the YARN effort. TaskTracker seems to be not present in the YARN framework at all [113].

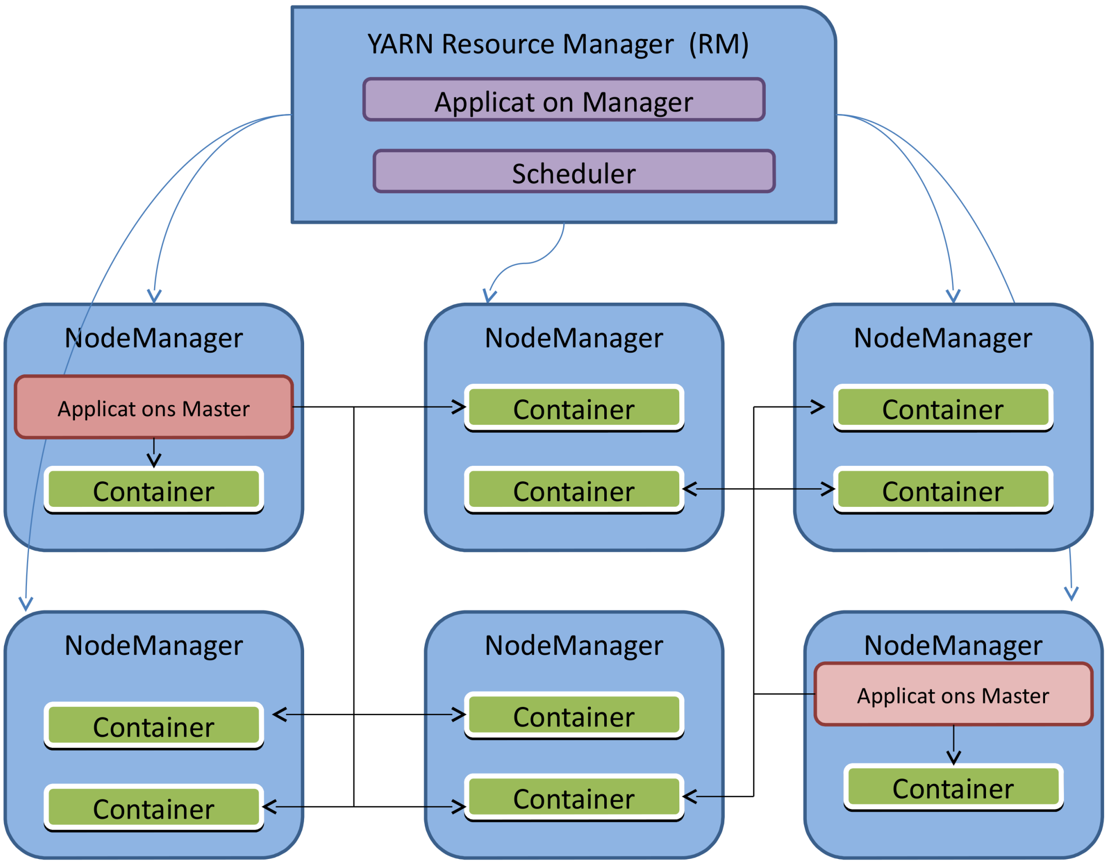


Figure 3.10: YARN - Yet another new Hadoop

The Scheduler models only memory in YARN v1.0. Every node in the system is considered to be composed of multiple containers of minimum size of memory (say 512 MB or 1 GB). The ApplicationMaster can request any container as a multiple of the minimum memory size. Eventually, YARN wants to move to a more generic resource model, however, for YARN v1 the proposal is for a rather straightforward model.

The resource model is completely based on memory (RAM) and every node is made up of discreet chunks of memory. Unlike Hadoop, the cluster is not segregated into map and reduce slots. Every chunk of memory is interchangeable and this has huge benefits for cluster utilization - currently a well known problem with Hadoop MapReduce is that jobs are bottlenecked on reduce slots and the lack of fungible resources is a severe limiting factor.

The lifecycle of a Map-Reduce job running in YARN is as follows:

1. Hadoop MR JobClient submits the job to the YARN ResourceManager (ApplicationsManager) rather than to the Hadoop MapReduce JobTracker.
2. The YARN RM-ASM negotiates the container for the MR Application Master with the Scheduler (RM-S) and then launches the MR AM for the job.
3. The MR AM starts up and registers with the RM-ASM.
4. The Hadoop MapReduce JobClient polls the ASM to obtain information about the MR AM and then directly talks to the AM for status, counters, etc.
5. The MR AM computes input-splits and constructs resource requests for all maps and reducers to the YARN Scheduler.
6. The MR AM runs the requisite job setup APIs of the Hadoop MR OutputCommitter for the job.
7. The MR AM submits the resource requests for the map/reduce tasks to the YARN Scheduler (RM-S), gets containers from the RM and schedules appropriate tasks on the obtained containers by working with the Node Manager for each container.
8. The MR AM monitors the individual tasks to completion, requests alternate resource if any of the tasks fail or stop responding.

9. The MR AM also runs appropriate task cleanup code of completed tasks of the Hadoop MR OutputCommitter.

10. Once the entire map and reduce tasks are complete, the MR AM runs the requisite job commit APIs of the Hadoop MR OutputCommitter for the job.

11. The MR AM then exits since the job is complete.

3.10 Concluding Remarks

In this chapter we have mainly concentrated on the Map phase of a MapReduce job and provided a model for predicting the completion time of the map tasks based on their service demands on CPU and disk using Mean Value Analysis (MVA). We also discussed in some detail the way current Hadoop's MapReduce handles the completion of a job through different phases. Hadoop looks quite different though with its 2.0 version released during summer of 2013. Some of the main reasons as to why Facebook implemented Corona instead of getting into YARN (which is being adopted as the de-facto Hadoop distribution) and adapt that to their need are detailed below (taken from a Facebook engineer's blog posting).

Firstly, YARN is still not push based scheduling and thus would not have solved one of the main probelems of lowering job latency. Secondly, in Corona JobTracker and the Job Client can run in the same JVM. In YARN, the JT equivalent has to be schedued as a separate process and thus extra steps are required to start the job. YARN is more geared towards Capacity scheduler vs. Fair Scheduler for Corona. Next, Corona used Thrift based RPC vs. YARN's protocol buffers. Facebook was not all that interested in working with protocol buffers. Lastly and perhaps most importantly, Facebook already had huge investment in Hadoop and basically had a lots of changes on the 0.20 Hadoop distribution. Adopting YARN, which is 0.23 and consequently essentially a new forked code of Hadoop, would have been very time consuming for Facebook. And as such Corona is a small codebase, not as diverse from current Hadoop as YARN is. This dissertation work was done using Hadoop 1.0 and is fully applicable to Hadoop 2.0. Even though the scheduler interfaces have changed some, all the models and software tools discussed and developed as part of this dissertation can be easily adapted for Hadoop 2.0.

Chapter 4: Analytic Performance Models of Applications in Multi-core Computers

4.1 Introduction

The work described in this chapter has been published in [5]. The top clock speed for CPUs peaked out at 4 GHz or so around a decade ago. Chip designers added multiple CPUs per socket and multiple sockets per machine to increase the processing power. However, the pressure on the memory subsystem increased due to multiple CPU cores accesses. The two main sources of performance inefficiency in a multi-core architecture are: (1) The overhead of cache-coherency management. (2) Memory bandwidth, memory latency and contention for memory access among cores [43].

The main contributions of this chapter are: (1) a method to quantify the slowdown due to access to shared memory bus and (2) simple and efficient predictive approximate single and multi-class analytic queuing models that consider contention due to shared memory elements (e.g., shared caches, bus, and memory banks) on SMP and NUMA multi-core machines.

Several papers [19,25,73] studied the effects of memory contention on multi-processor systems. This work differs from previous work in that it provides a simpler model whose parameters are easy to obtain in practice and allows for performance prediction of multiple applications of different kind (i.e., the multiple class case). Our model was validated with experimentation on several multi-core machines. Other approaches to assessing the performance characteristics of multi-core systems include simulation [59] and benchmarking [60]. More related work is covered in Section 6.8.

The rest of this chapter is organized as follows. Section 4.2 shows experimental results that demonstrate the performance impact of contention for shared memory bus in multi-core computers. The next section presents a single-class analytic model used to predict the performance of applications running on SMP and NUMA multi-core machines. Section 4.4 describes how memory access time can be measured. The following section presents results that validate the single-class model with experimental data. Section 4.6 extends the model to multiple classes of jobs and Section 4.7 discusses the experimental validation of the multi-class model. The next section discusses related work and Section 5.8 presents concluding remarks and future work possibilities.

4.2 Effects of Memory Contention on Multi-core Computers

Modern CPU cores have complex memory hierarchies [35]. Level 1 (L1) cache (sometimes separately for data and instructions) is the closest to the CPU and has the fastest access, L2 comes after L1 away from the CPU and there may even be an L3 cache. Programs that execute on multi-core machines compete for access to shared memory components (e.g., data banks, front side bus, and shared caches). This contention generates a queuing effect that cannot be ignored when predicting the performance of concurrent applications running on such machines. The effect of contention for access to memory can be illustrated in Fig. 4.1, which depicts the average execution time versus the concurrency level (i.e., number of concurrent programs in execution) for two versions of a C program running on a 12-core Intel Westmere-EP machine. Similar results were observed for the same programs on a 4-core Intel Sandy Bridge machine.

The characteristics of the two versions of the C program are summarized in rows 1, 2, 4, and 5 of Table 4.1. For the Westmere-EP machine case (depicted in Fig. 4.1), the program performs 700 allocations and deallocations of 50-MB chunks of main memory. In one case (bottom curve of the graph), the allocated memory is never accessed by the program. In the other (top curve), each allocated memory address is accessed once by the program. In all cases, the programs do not execute any disk I/O and perform sequential memory access. We noted an even more dramatic effect of memory contention on the execution times of concurrently running programs when memory access is random.

The graphs in Fig. 4.1 show that contention for memory plays a significant role on a program's execution time in multi-core machines. See also [49]. For the 12-core machine, when there is only one copy of the program in execution (i.e., concurrency level equal to one), the difference in execution time is 14.1 sec, which can be attributed to accessing 7 GB of main memory. As the concurrency level increases from 1 to 12, the difference in execution time increases to 87.8 sec.

Table 4.1: Specification of the experiments ran using the micro benchmark (MBench)

	Architecture	**Code Parameters**	**Amount of Memory**	**Disk I/O**	**Concurrency**	**PMU Collected**
1	Sandy Bridge	Memory Allocation	700 alloc of 10 MB	None	1 to 8	None
2	Sandy Bridge	Memory Allocation, Access	700 alloc of 10 MB	None	1 to 8	Backend stall cycles for Concurrency
3	Sandy Bridge	Memory Allocation, Access and I/O	700 alloc of 10 MB	Several hundred MB file r/w with cold disk cache	1 to 8	Backend stall cycles for Concurrency
4	Westmere EP	Memory Allocation	700 alloc of 50 MB	None	1 to 18	None
5	Westmere EP	Memory Allocation. Access	700 alloc of 50 MB	None	1 to 18	Backend stall cycles for Concurrency
6	Westmere EP	Memory Allocation, Access and I/O	700 alloc of 50 MB	Several hundred MB file r/w with cold disk cache	1 to 18	Backend stall cycles for Concurrency

That means that an additional 73.7 sec (i.e., 87.8 - 14.1) are incurred due to mainly memory (and possibly other) contention.

After the concurrency level exceeds the number of cores, the difference in execution time continues to increase due to contention for cores. For example, for a concurrency level of 18, the difference in average execution time is 126 sec, of which 14.1 sec are due to memory access time, 73.7 sec due to contention for memory access, and 80.1 sec due to contention for cores. Thus, over 40% of the execution time can be attributed to memory contention. For a 4-core Sandy Bridge machine, more than 16% of the execution time is due to memory contention.

Therefore, models that predict the execution time of applications in multi-core machines must take memory contention into account. These models are presented and validated in the rest of this chapter.

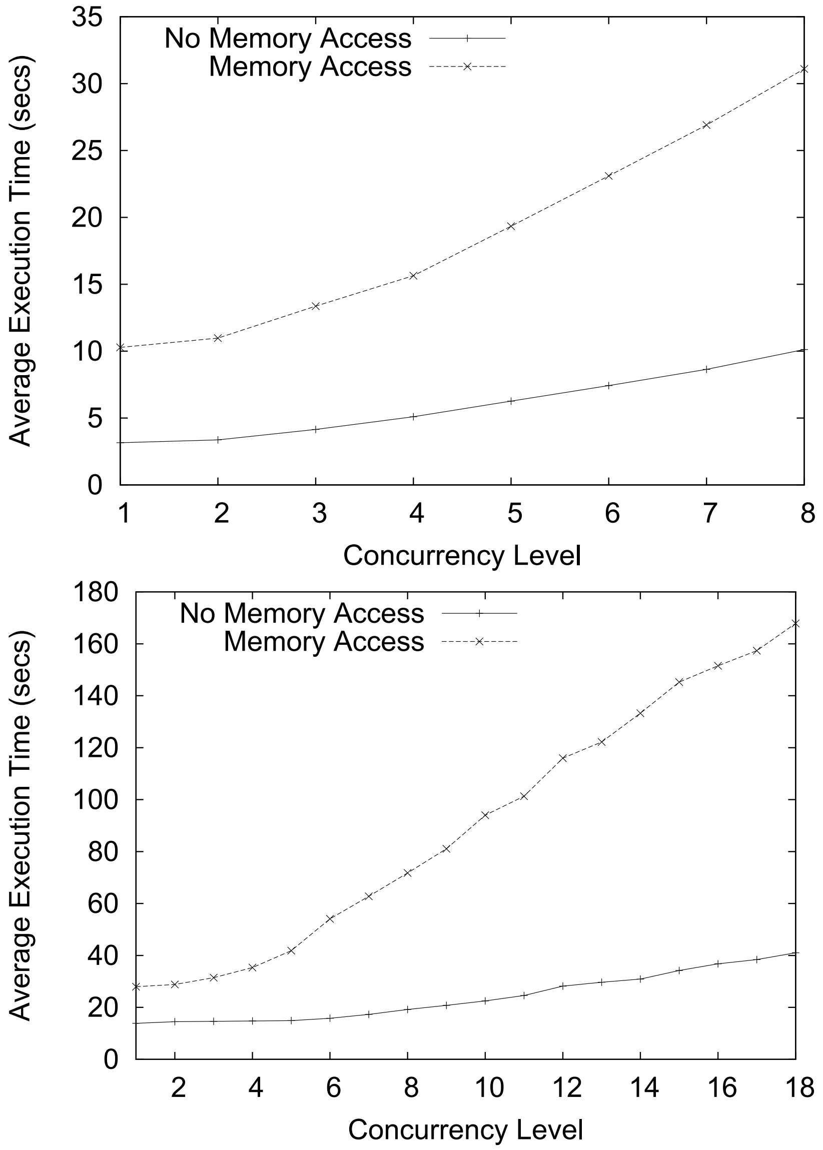


Figure 4.1: Top: Effect of memory contention on a 4-core Sandy Bridge machine. Bottom: Effect of memory contention on a 12-core Westmere EP machine.

4.3 A 2-layer Analytic Model For Multi-Core Computers

This section presents a simple single class (i.e., all jobs are equivalent in terms of their resource consumption) queuing network based model that captures memory contention. Section 4.6 extends the model to multiple classes.

4.3.1 Basic Model

We consider that there are m cores and n instances of a job concurrently being executed (i.e., the single class case). We extend the model to the case in which there are concurrent instances of different job types (i.e., the multiclass case) later in the chapter. We consider that jobs use the CPU and perform I/O and are not bound to any of the m cores; they are assigned to any available core when dispatched by the operating system. It is assumed that all the cores of the machines are identical as is the case with most modern server class machines. Relaxing that assumption would need adjusting for machines with heterogeneous cores with varying capacity.

Figure 4.2 shows a pictorial description of our analytic queuing model of multi-core systems. The model has two layers. The bottom layer is the memory contention model and consists of three devices:

- A delay device called *Core-I* that represents the total time, $D_{\text{Core}-\text{I}}$, spent by a core executing instructions while not waiting for memory access (for data or instructions). We call $D_{\text{Core}-\text{I}}$ the service demand at the *Core-I* device.
- A load independent queuing device called *Memory* that represents the system memory elements shared by all cores (e.g., L3 cache, memory bus, and memory banks). For now let us consider only one combined memory queue. The service demand at this device is denoted by D_{Mem}. The contention for memory (i.e., queuing for this device) is caused by the fact that several cores are concurrently executing instructions that require access to memory. For the basic model, only the local memory of Fig. 4.2 should be considered (see Section 4.3.2 for the NUMA extension).
- A load independent queuing device that represents I/O on a disk. Let D_d be the service demand on the disk.

The memory contention model is a closed queuing network model with a population equal to k, where $k = 1, \cdots, m$. If there are $n > m$ jobs in execution, there cannot be more than m active cores (i.e., executing instructions or waiting for memory). This model can be solved using Mean Value Analysis (MVA) [63] and yields as a result the value $T(k) = D_{\text{Core}-\text{I}} + R'_{\text{Mem}}(k)$ for $k = 1, \cdots, m$,

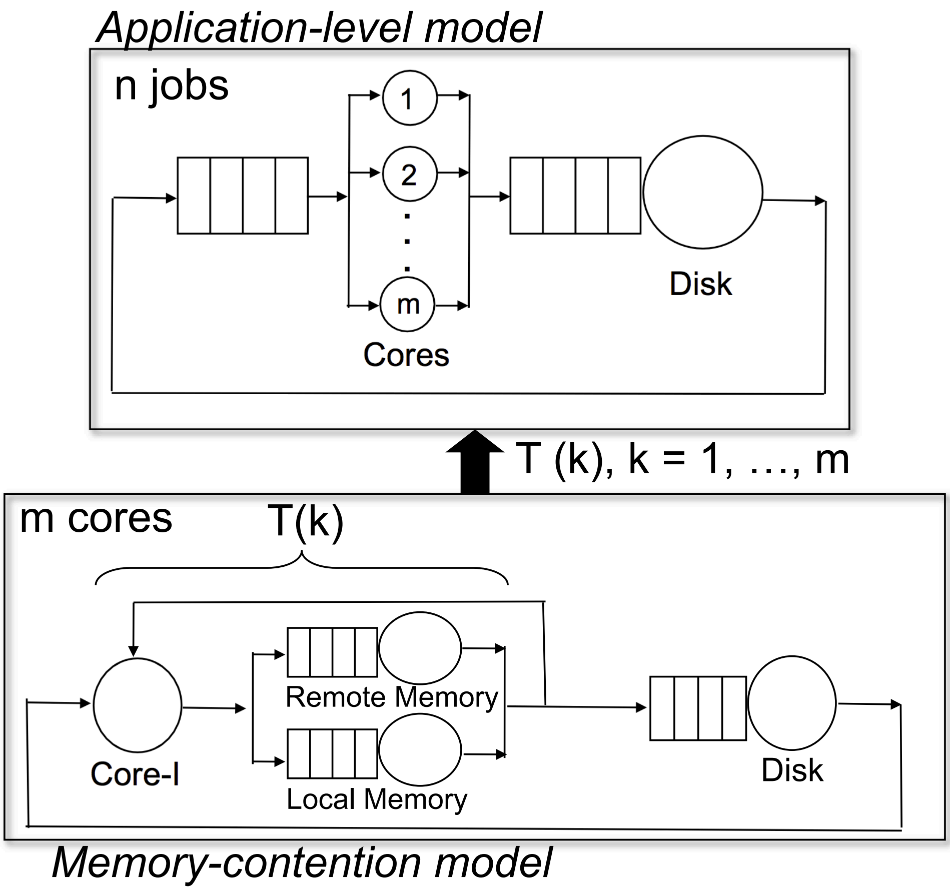


Figure 4.2: Two-layer analytic model of a multi-core computer.

where $R'_{\text{Mem}}(k)$ is the memory residence time (i.e., total time waiting to access memory + total time accessing memory). $T(k)$ is then the average time spent by a job either using a core, waiting to access memory, or accessing memory, when there are k jobs in execution concurrently.

The service demands for ***Core-I***, ***Memory***, and ***Disk*** are computed by running a single instance of the application multiple times to obtain average values. The disk service demand is computed as follows using the Service Demand and Utilization Laws [55]:

$$D_d = \frac{X_d \times S_d}{X_0(1)} \tag{4.1}$$

where X_d, S_d, and $X_0(1)$ are the disk throughput (in I/Os per second), average disk service time (in seconds), and average system throughput for a concurrency level of one, respectively. The values of X_d and S_d can be easily obtained by running `iostat` or any equivalent utility on a Unix/Linux system. $X_0(1)$ is just the ratio between the number of instances of the program executed by the duration of the experiment.

The values of $D_{\text{Core-I}}$ and D_{Mem} are computed as follows. First, as we run the program with a concurrency level of one, we must collect the average job elapsed time, $E(1)$, and the percent of "stalled cycles backend," **PercStalledBackendCycles**, which can be obtained by running `perf` as described in Section 4.4. PercStalledBackendCycles is the percentage of clock cycles that cannot be used by the processor because it is waiting for instructions or data to come from the memory subsystem using a shared bus.

Since with a concurrency level of one there is no I/O contention, we can write that

$$T(1) = D_{\text{Core-I}} + D_{\text{Mem}} = E(1) - D_d. \tag{4.2}$$

We can also write that

$$D_{\text{Mem}} = T(1) \times \text{PercStalledBackendCycles}. \tag{4.3}$$

Combining Eqs. (4.2) and (4.3) we get

$$D_{\text{Core-I}} = (E(1) - D_d) \times (1 - \text{PercStalledBackendCycles}). \tag{4.4}$$

The values of $T(k)(k = 1, \cdots, m)$, are then used in the application-level model. We now provide a detailed explanation of the algorithm used to solve the combined memory-contention and application level models (see pseudo code in Algorithm 3). Consider the following notation:

- $R_i^{'}(k)$: residence time, i.e., total time spent by a job at device i (waiting to use or using the device) when the concurrency level is equal to k.
- $\bar{n}_i(k)$: average number of jobs at device i when the concurrency level is equal to k. This includes jobs waiting to use the device and using the device.
- $R_0(k)$: application response time, in seconds, when the concurrency level is equal to k.
- $X_0(k)$: throughput, in jobs/sec, when the concurrency level is equal to k.

The goal of the algorithm is to compute the execution time and the throughput of a program when the concurrency level is equal to n. Lines 1 to 12 in Algorithm 3 use regular single-class MVA equations [63] to solve the memory-contention level model. As a result, one obtains $T(m)$, $R_d^{'}(m)$, $R_0(m)$, and $X_0(m)$. The solution of the application level model starts at line 15. The concurrency level varies from $m + 1$ to n. The solution of this model follows the single class MVA equations with a few modifications described in what follows. Line 18 is the regular MVA equation for the residence time at the disk. Note that when $j = m + 1$, this equation needs $\bar{n}_d(m)$ obtained in the memory-contention model when $k = m$. Line 19 uses the Bard-Schweitzer (B-S) approximation [64] to compute the average queue length, $\bar{n}_d(j)$, at the disk. If the disk demand is greater than zero (line 20), the job throughput $X_0(j)$ is computed as $\bar{n}_d(j)/R_d^{'}(j)$ by combining Little's Law [51] with the Forced Flow Law [55]. Otherwise, $X_0(j)$ is computed in line 22 as follows.

According to Little's Law [51], the average number, $\bar{n}_{\text{Core}}(j)$, of jobs using one of the m cores, is equal to $X_0(j) \times T(m)$ for $j = m + 1, \cdots, n$. But, when $j > m$, all m cores are busy all the time and $\bar{n}_{\text{Core}}(j) = m$. Thus,

$$\bar{n}_{\text{Core}}(j) = m = X_0(j) \times T(m) \tag{4.5}$$

Algorithm 3 Multi-core analytic model - single class

```
    Input: $m, n, D_d, D_{\text{Core-I}}, D_{\text{Mem}}$
    /* Compute $T(k), k = 1, \cdots, m$ */
    Initialize: $\bar{n}_d(0) \leftarrow 0; \bar{n}_{\text{Mem}}(0) \leftarrow 0$
    for $k = 1 \rightarrow m$ do
 5:     $R_d^{'}(k) \leftarrow D_d[1 + \bar{n}_d(k-1)]$
        $R_{\text{Mem}}^{'}(k) \leftarrow D_{\text{Mem}}[1 + \bar{n}_{\text{Mem}}(k-1)]$
        $T(k) \leftarrow D_{\text{Core-I}} + R_{\text{Mem}}^{'}(k)$
        $X_0(k) \leftarrow k/[T(k) + R_d^{'}(k)]$
        $\bar{n}_d(k) \leftarrow X_0(k) R_d^{'}(k)$
10:     $\bar{n}_{\text{Mem}}(k) \leftarrow X_0(k) R_{\text{Mem}}^{'}(k)$
    end for
    $R_0(m) \leftarrow T(m) + R_d^{'}(m)$
    /* Set R and X in case $n = m$ */
    $R \leftarrow R_0(m); X \leftarrow X_0(m)$
15: /* Compute the execution time $R$ and throughput $X$
      for the application */
    for $j = m + 1 \rightarrow n$ do
        $R_d^{'}(j) \leftarrow D_d[1 + \bar{n}_d(j-1)]$
        $\bar{n}_d(j) \leftarrow \frac{j}{j-1} \times \bar{n}_d(j-1)$ /* B-S approximation */
        if $D_d > 0$ then
20:         $X_0(j) \leftarrow \bar{n}_d(j)/R_d^{'}(j)$
        else
            $X_0(j) \leftarrow m/T(m)$ /* See Eq. (4.6) */
        end if
        $\bar{n}_{\text{CPU}}(j) \leftarrow j - \bar{n}_d(j)$
25:     $R_{\text{CPU}}^{'}(j) \leftarrow \bar{n}_{\text{CPU}}(j)/X_0(j)$
        $R_0(j) \leftarrow R_{\text{CPU}}^{'}(j) + R_d^{'}(j)$
    end for
    $R \leftarrow R_0(n); X \leftarrow X_0(n)$
```

which implies that

$$X_0(j) = \frac{m}{T(m)} \tag{4.6}$$

for the case in which there is no (or negligible) I/O.

In line 24, the average queue length at the CPU can be obtained by subtracting the average queue length at the disk from the concurrency level. Then, in line 25, Little's Law and the Forced Flow Law are used to compute the residence time at the CPU. The response time $R_0(j)$ is just the sum of the residence times at the CPU and disk (line 26).

It is instructive to note that if we were to model situations in which processes are bound to cores, we would only need to modify the application-level model as follows. Instead of having a single queue for all the cores, we would need to have a dedicated queue in front of each core. The resulting application-level queuing network is a standard queuing network model that can be solved using well known methods and does not require the Bard-Schweitzer approximation explained in the preceding paragraphs.

This section presented a two-level single-class algorithm that considers the time spent at the cores, waiting to use the cores (contention for cores), accessing memory, waiting to access memory (memory contention), performing I/O, and waiting to access I/O devices (I/O contention). While we only showed one disk in our model, it would be trivial to consider any number of disks by adding the corresponding MVA residence time and queue length equations, as would be recognized by those who are familiar with queuing network models.

While we have not considered burstiness in memory access, Eager et al. developed and validated a model that uses Approximate MVA to deal with this situation [26].

4.3.2 Non Uniform Memory Access (NUMA) Extension

The NUMA architecture was designed to mitigate the scalability limits of the Symmetric Multi Processing (SMP) architecture [35]. With SMP, all memory accesses are performed on the same shared memory bus. This performs well for a small number of CPU cores, but congestion on the shared bus limits scalability on machines with dozens of CPU cores, common on modern machines. NUMA alleviates these bottlenecks by limiting the number of CPU cores on any one memory bus, and connecting the various nodes by means of a high speed interconnect. In a NUMA architecture, memory access time depends on the location of the memory relative to the processor core. As an example, the NUMA machine used for conducting the experiments for this work has 12 CPU cores and 8 GB of memory equally divided between two nodes of 4-GB of memory each. For six out of the 12 cores, 4 GB of memory are considered local and the other 4 GB remote.

To extend our basic model to NUMA architectures, we need to consider both memory queues shown in Fig. 4.2: local and remote. These two load independent devices have service demands that we denote by $D_{\text{Mem}-\text{Local}}$ and $D_{\text{Mem}-\text{Remote}}$ and can be written as $D_{\text{Mem}-\text{Local}} = N_{\text{L}} \times T_{\text{L}}$ and $D_{\text{Mem}-\text{Remote}} = N_{\text{R}} \times T_{\text{R}}$, where N_{L}, T_{L}, N_{R}, and T_{R} denote the average number of accesses to local memory, access time to local memory, number of accesses to remote memory, and access time to remote memory, respectively. Let us define the following ratios: $K_N = N_{\rm L}/N_{\rm R}$ and $K_T = T_{\rm R}/T_{\rm L}$. Thus, $D_{\rm Mem-Local}/D_{\rm Mem-Remote} = K_N/K_T$.

A process can potentially have its data segment either on a local or a remote memory node. Despite the long history of Linux, NUMA awareness is a relatively new addition to Linux kernels. Starting from 2.6.X, the Linux kernel has adapted to larger memory and number of cores. The latest Linux kernels make every attempt to keep a process on the node where its memory is initially allocated. Memory intensive programs, when migrated between cores can have its data on both local and remote memory segments. Linux uses the "local node first" scheduling policy; our experiments have shown that on a fully utilized machine, a process is scheduled to a non-local node between 15% to 25% of the time. Remote memory latency is substantially higher than local memory latency (i.e., $K_T \gg 1$).

Our measurements of the NUMA effect using a test program and also the `numactl` utility showed a slowdown of approximately four times when the accessed memory was remote to the executing core (i.e., $K_T = 4$). Based on the observations above regarding Linux scheduling, we considered that a program executes 20% of the time on a remote node (i.e., the number of accesses to local memory is four times the number of accesses to a remote memory, or equivalently $K_N = 4$). Then $K_N/K_T = 4/4 = 1$ and $D_{\rm Mem-Local} = D_{\rm Mem-Remote}$.

Table 4.5 shows the parameters obtained by running several programs on a NUMA-based Intel XEON machine. The columns represent the parameters for two different sets of experiments consisting of three classes of jobs. The NUMA access characteristics discussed above can be ignored if tasks are considered to have core affinities.

4.4 Measuring Memory Access Time

We conducted experiments on Intel machines with two different architectures: an Intel Sandy Bridge Micro-Architecture based system (4 cores) and an Intel Westmere-EP Micro-Architecture based system (12 and 16 cores). See Table 4.2 for their characteristics.

We developed a micro-benchmark (MBench) in C and used it to gather experimental data (see pseudo code in Algorithm 8). Others have developed micro-benchmarks for similar reasons [40].

Algorithm 4 Micro-benchmark (MBench) Pseudo Code

```
Input: $memAccessRequired, diskWriteRequired$
Input: $MEM\_SIZE, REPEAT$
for $i = 0 \to REPEAT$ do
   if $diskWriteRequired$ then
      performDiskIO;
   end if
   $byte * ptr\_mem \leftarrow$ malloc($MEM\_SIZE$)
   $hold\_ptr \leftarrow ptr\_mem$;
   for $i = 0 \to MEM\_SIZE$ do
      Access the byte pointed to by $ptr\_mem$
      if $memAccessRequired$ then
         $ptr\_mem + +$
      end if
   end for
   free $hold\_ptr$
end for
printTimingInfo;
```

The program parameters determine if the code actually accesses the allocated memory and whether it performs disk I/O. This allowed us to control and quantify the behavior of the code and measure the performance effect of memory access and I/O. See Table 4.1 for the various combinations of experiments with the micro-benchmark.

Table 4.2: Specifications of the Intel Sandy Bridge and Westmere-EP machines used in the experiments

System	Intel Sandy Bridge	Intel Westmere-EP
Processor	I5-2520M	XEON X5660
Family/Model	6/42	6/44
Speed	2.5 GHZ	2.8 GHz
Memory	6 GB	48 GB
Cores	4	12 and 16
NUMA	No	Yes
Operating System	Ubuntu 12.04	RHEL 2.6.32
On-chip cache	L2-2 x 256 KB, L3-3MB	L2-6 x 256 KB, L3 - 12 MB

Our goal was to estimate the memory service demand, D_{Mem}, i.e., the amount of time spent by a job accessing memory when not competing with other jobs for shared memory elements. To that end, we chose to measure the time taken by a program to access memory with the help of the hardware based Performance Monitoring Unit (PMU) built into modern Intel processors (AMD and Alpha processors have similar functionality; however, we have not used these in the experiments discussed in this work). The hardware counters available inside the CPU provide a very precise picture of CPU resource utilization and there are literally hundreds of events to choose from [33,50].

Eranian [27] gives a very detailed description of Intel processor performance counters and indicates that these counters are the key to understanding issues with the memory subsystem. He shows how to collect memory-related metrics from today's hardware using the performance counters with the help of the Linux tool `perfmon2` that his team developed. His paper also discusses the nuances of Non Uniform Memory Access (NUMA) based machines.

Our experiments are based on a Linux utility called `perf` [85], based on `perfmon2`, that allows us to read the PMU counters. Figure 4.3 shows a representative output for a `perf stat` command. We concentrate on the line containing the counter "stalled-cycles-backend" and note that a total of 27.33% of the total cycles were due to stalls for this particular execution. Back end stalls have two main sources: memory sub-system stalls and execution stalls. These resource stalls happen at the back-end of the pipeline when a resource cannot be allocated and the cost to access memory is a big part of the cost incurred. Similarly, there are quite a few events such as L1 instruction cache miss and instruction buffer full, that contribute to front end stall cycles. Instructions that involve Load and Store from the L1, L2, L3 caches or from memory result in wasted cycles. The CPU of modern processors contain specific hardware counters to measure the number of wasted cycles. We have considered the total stalls value as an indicator of the time spent performing memory access and this value is reflected as part of the "stalled-cycles-backend" metric. It is to be noted that the amount of prefetching over the entire execution of the program is captured by the value of the metric 'PercStalledBackendCycles'. Even though a program may exhibit different memory access patterns during the lifetime of its execution, our model considers the average value during the execution of the program. Our experimental results confirm that such an approach yields accurate results that can be obtained by following a simple and feasible measurement procedure.

The most important registers and counters on Intel processor microarchitectures are kept consistent across all architectures (we used Sandy Bridge and Westemere-EP). That makes it easy to use a simple C-language structure to setup the event select register used by the Linux tool `perf`. For the backend stalled cycles, the value of interest is an event value of 0xb1 and umask value of

```
Performance counter stats for './MBM YES NO 7':

      36483.335874 task-clock                #    0.998 CPUs utilized
                43 context-switches          #    0.000 M/sec
                 0 CPU-migrations            #    0.000 M/sec
           476,999 page-faults               #    0.013 M/sec
   112,341,248,111 cycles                    #    3.079 GHz                     [83.33%]
    72,222,333,153 stalled-cycles-frontend   #   64.29% frontend cycles idle    [83.33%]
    30,700,100,241 stalled-cycles-backend    #   27.33% backend  cycles idle    [66.67%]
   114,989,289,020 instructions              #    1.02  insns per cycle
                                             #    0.63  stalled cycles per insn [83.34%]
    14,576,388,567 branches                  #  399.536 M/sec                   [83.33%]
           952,160 branch-misses             #    0.01% of all branches         [83.33%]

      36.554921783 seconds time elapsed
```

Figure 4.3: Output of measurements with `perf`.

0x3f in that structure [33].

We execute our micro-benchmark program `mbm` with parameters specified in `[params]` using the command `perf stat ./mbm [params]` which gives us the stall values and various other predetermined events for this program for a job concurrency level of 1. The value of backend stall cycles provided in the output is used in determining the time spent by the program to access memory (see Eq. (4.3)). The next section shows the results of using the model presented in the previous section to predict the application performance for higher concurrency levels.

4.5 Single Class Results

This section reports the results obtained by running MBench with various parameters (memory allocation with memory access and no disk access, memory allocation and access and disk access), on the Sandy Bridge and Westmere-EP machines as described in Table 4.1 (rows 2, 3, 5, and 6). We also present the results of running programs from well-known Linux benchmarks (Hbench [87] and UnixBench [89]) as well as from SPECCPU2006. The results show that considering memory contention in the model significantly improves its accuracy.

MBench always allocates a big chunk of memory. However, an input parameter to the program

Table 4.3: Input parameters for the MBench models obtained by running the programs on the Intel Sandy Bridge and Westmere-EP machines.

Parameter	Disk Access		No Disk Access	
	Intel Sandy Bridge	Intel Westmere-EP	Intel Sandy Bridge	Intel Westmere-EP
$D_{\text{Core-I}}$	10.94 sec	20.16 sec	26.26 sec	19.70 sec
D_{Mem}	8.67 sec	8.88 sec	11.8 sec	8.31 sec
PercStalled BackendCycles	34%	30%	31%	30%
D_d	8.93 sec	0.8 sec	0	0

determines if it accesses the memory or simply allocates and frees it. This way, we are able to see the effect of cache misses and DRAM access. It is very evident from the results that the portion of the time performing memory access (and the eventual contention experienced by the program due to shared memory elements) is appreciably higher when the program actively accesses the allocated memory.

The input parameters obtained by running the programs are provided in Table 4.3. The first two columns represent the case of memory and disk access and the last two columns contain input parameters for the case of no disk access.

All results depicted in the figures that follow in this chapter show three graphs: (1) Experimental results obtained by running real applications on multi-core machines. These curves represent average values for several runs and are accompanied by 95% confidence interval bars. (2) Results obtained through the model that considers memory contention, as presented in this work. (3) Results obtained by using a queuing network (QN) model such as the application-level model at the top of Fig. 4.2. Such a model only considers contention for cores but does not capture contention for memory. The solution to the model without memory contention is obtained by using MVA (for the single-class case) and AMVA (for the multi-class case) [55] in addition with Seidmann's approximation [65] for multi-server queues (representing the multiple cores in this case). Moreover, at the top of each graph we indicate what contention elements are present, which program was used in the experiment, and the number of cores of the machine used. For example, the top of Fig. 4.4 displays "CPU/Memory contention - MicroBench (4 cores)" to indicate that there is only contention for cores and memory but no disk activity and that our microbenchmark was ran on a 4-core machine. The top of Fig. 4.6 displays "CPU/Memory/Disk contention - MicroBench (4cores)" to indicate that the experiments also include disk activity.

Figure 4.4 shows three curves for the Sandy Bridge experiments with no disk access and memory access. These curves show the average execution time of the program vs. the concurrency level. The lower curve represents the results obtained with an analytic model that does not consider memory contention. Because this model does not consider memory contention and there is no disk contention, the average execution time does not vary until the concurrency level exceeds the number of cores (4 in this case). After that point, the execution time increases due to contention for cores. The curve above that shows the results obtained with our proposed model that considers memory contention. As it can be seen, the execution time increases from concurrency 1 to 4 due to memory contention and continues to increase after that due to core contention. The curve above that shows the results obtained from running experiments. As it can be seen, our model tracks very closely the experimental results. For example, for a concurrency level equal to 4, the relative percent error (i.e., 100 $\times$ (experiment - model) / experiment) between the experimental results and the model with memory contention is 11.2%, while the percent relative error between the experimental results

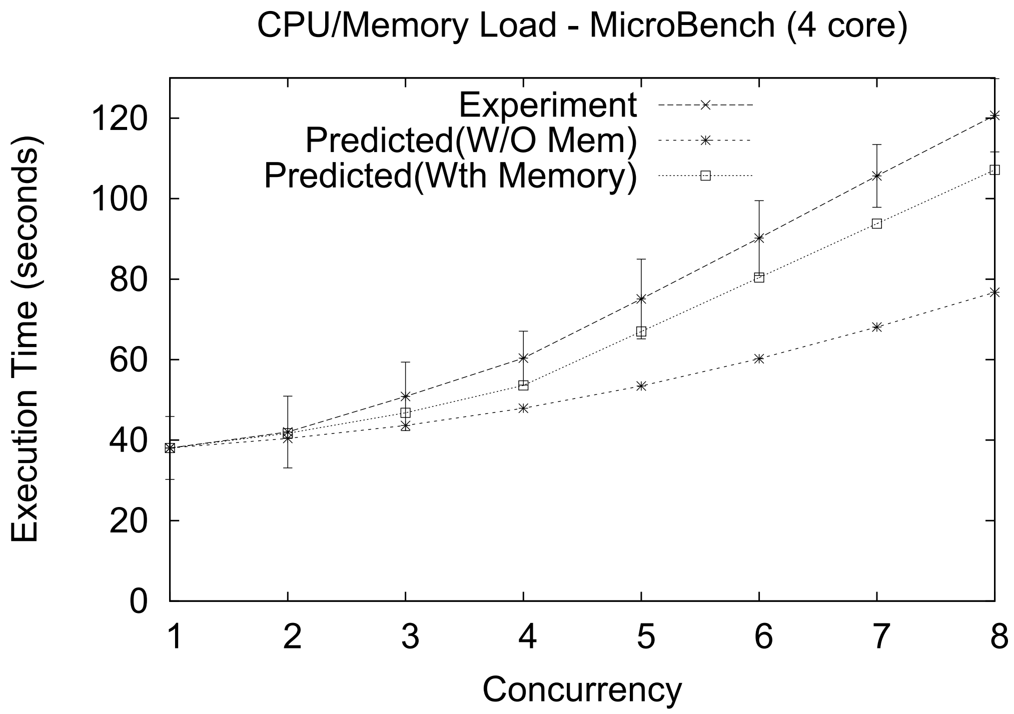


Figure 4.4: Average execution time vs. concurrency level for MBench without disk I/O on Sandy Bridge (4 cores).

and the model that does not consider memory contention is 37% for that concurrency level. For a concurrency level of 8, the model with memory contention predicts an average execution time that is 89% of the experimental value while the model without memory contention predicts a value that is 63% of the experimental value.

Similar results can be observed for the experiments carried out with the Westmere-EP machine as illustrated in Fig. 4.5. As it can be seen, the model with memory contention tracks very closely the experimental results and exhibits a maximum absolute percent relative error of 14.1% for all concurrency levels. The model that does not consider memory contention exhibits very large errors. For example, for a concurrency level of 18, the model without memory contention predicts an average execution time that is only 25% of the experimental value, while the model with memory contention predicts a value that is 90% of the experimental value. The errors in the case of the Westmere-EP machine (as compared to Sandy Bridge) are more pronounced because in the Westmere-EP case we have 12 cores competing for memory and the program in this case accesses 35 GB of memory as opposed to 7 GB. On the Westmere-EP machine, we experimented with and without hyper-threading enabled and the results are statistically equivalent.

The next two figures illustrate a situation where the program also performs I/O. These situations correspond to rows 3 and 6 in Table 4.1 for the Sandy Bridge and Westmere-EP machines, respectively. Figure 4.6 shows three curves for the Sandy Bridge machine: experiment (top), model with memory contention (middle) and model without memory contention (bottom). Again, the model with memory contention tracks the experimental results much better. In this case, since there is disk contention, the model without memory contention shows an increase in the average response time even in the range from 1 to 4. However, the model with memory contention also predicts memory contention besides disk contention. The percent relative error for a concurrency level of 4 for the memory contention model is 17.6% while that of the model without memory contention is 30.6%.

The situation for the Westmere-EP case depicted in Fig. 4.7 shows a very good correlation for the model with memory contention: a 12.5% percent relative error at concurrency level of 12 and 14% for concurrency level equal to 18. The model without memory contention exhibits a percent relative error of 71% percent at a concurrency level of 12 and an error of 76% at a concurrency level of 18. This means that the model without memory contention shows a value more than four times smaller than the experimental value for a concurrency level of 18 while the model with memory contention computes a value that is 86% of the experimental value.

We now describe validation results obtained by running programs of the UnixBench [89] and HBench [87] benchmarks. HBench has its roots in lmbench [88], which is another popular memory bandwidth and latency benchmark useful in comparing Unix/Linux system performance. We focus first on HBench, a suite of portable benchmark programs, which along with memory read, write and copy also performs numerous other tests such as network speed. We focused on the memory read, write, and copy functions. For each of these functions, HBench performs tests on many different sizes of memory starting from a few kilobytes all the way up to several megabytes. While running the tests and observing the stall values through `perf stat`, it was obvious that some of the programs were experiencing low stall values and some of them high stall values. This is not surprising since the runs with a high memory exposure have to spend more time doing memory access and, as a result, experience more contention due to memory access when multiple copies of

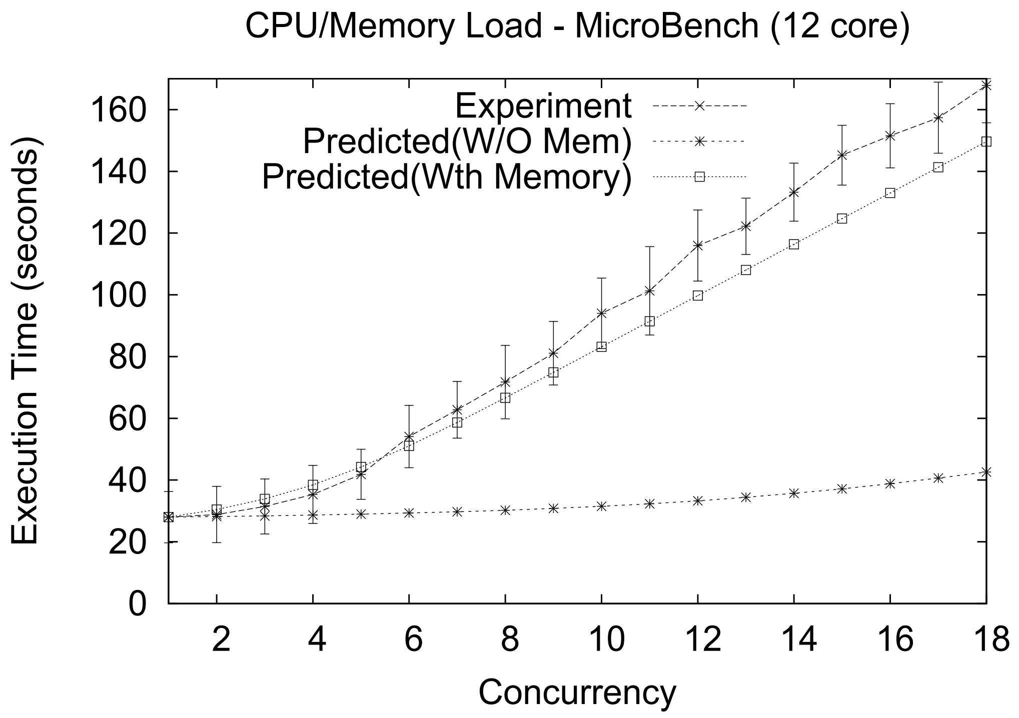


Figure 4.5: Average execution time vs. concurrency level for MBench without disk I/O on Westmere (12 cores) .

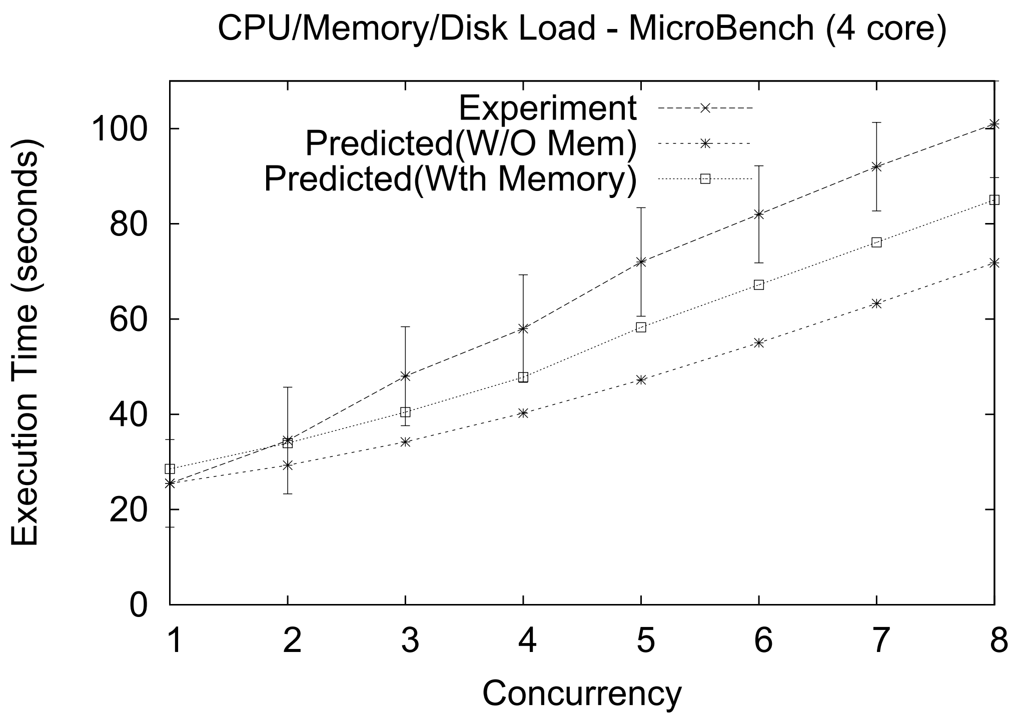


Figure 4.6: Average execution time vs. concurrency level for MBench with disk I/O on Sandy Bridge (4 cores).

the same program execute concurrently. We collected execution time information along with the stall values for each of these runs. The top graph of Fig. 4.8 shows curves of average execution time vs. concurrency level on a 4-core Sandy Bridge machine running HBench programs. The observed backend stall percentage is 51%. The bottom graph shows curves for HBench programs running on a 12-core Westmere machine; the backend stall percentage is 25% in this case. As it can be seen, the model with memory contention tracks the experimental results very closely while the model without memory contention deviates significantly from the experimental results.

We now discuss results obtained by running programs from UnixBench, which includes programs that perform multiple types of tests of Unix/Linux systems: DhryStone, Towers of Hanoi (a purely recursive algorithm), Arithmetic, Grep of a large file, graphics, system calls, and numerous others. We discuss in what follows results obtained by running Towers of Hanoi and Grep. These programs were chosen (classical benchmark, Unix utilities and a well known recursive algorithm) for representing a good mix of programs with a varied amount of CPU, memory, and I/O service demands.

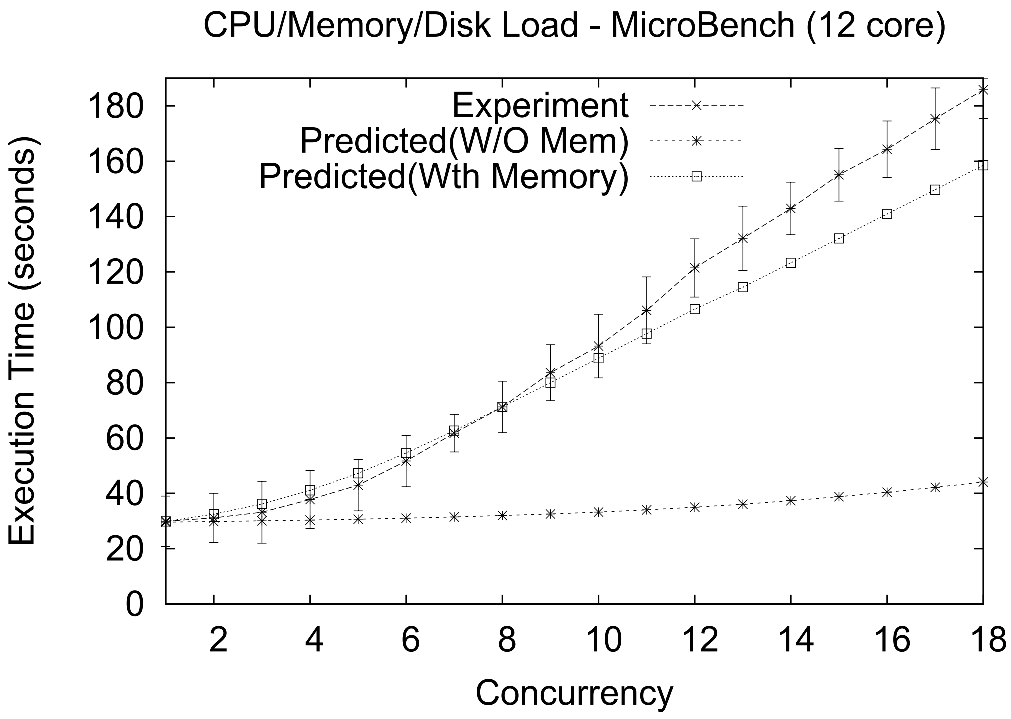
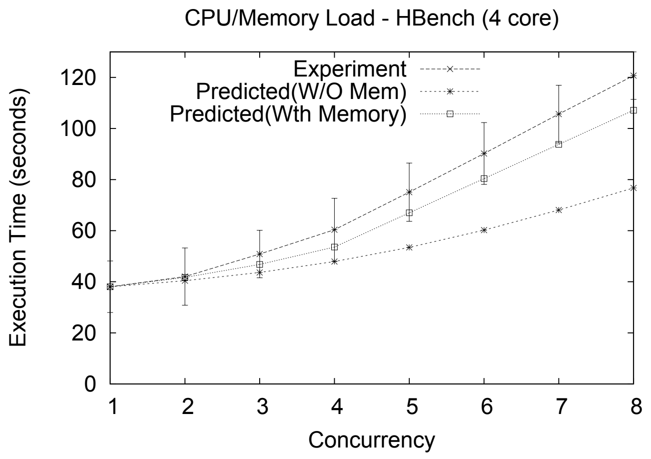


Figure 4.7: Average execution time vs. concurrency level for MBench with disk I/O on Westmere (12 cores).

The graph in Fig. 4.9 shows multiple instances of the Towers of Hanoi program running on a 16-core Westmere machine. The percentage of backend stalls is large enough (10%) to make the model without memory contention deviate significantly from the experimental results, while the model with memory contention remains much closer to the experimental results. For example, the percent relative error of the model with memory contention is 19.7% for the 16-core case and a concurrency level equal to 24 while the same error is 50% for the same concurrency level for the model without memory contention.

The top graph of Fig. 4.10 shows the results for `grep` on a 4-core Sandy Bridge machine; the percent stall is 35%. One can see that the model with memory contention tracks much better the experimental results than the model without memory contention. The same behavior can be seen in the bottom graph of the figure, which shows the results of running many instances of `grep` on a 16-core Westmere with a 10% percent backend stall.

We also ran experiments with the SPECCPU2006 benchmark suite on 4-core and 12-core Linux CentOS servers. We chose two benchmarks from the suite: 429.mcf from the Integer benchmarks



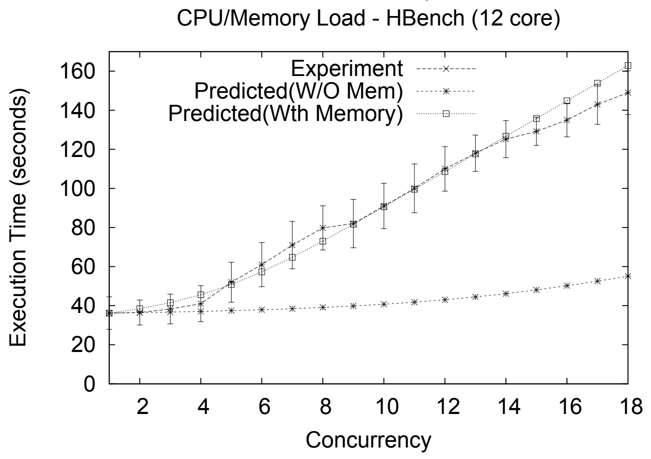


Figure 4.8: Average execution time vs. concurrency level for the HBench benchmark. Top: Sandy Bridge (4 cores) and 51% backend stall. Bottom: Westmere (12 cores) and 25% backend stall.

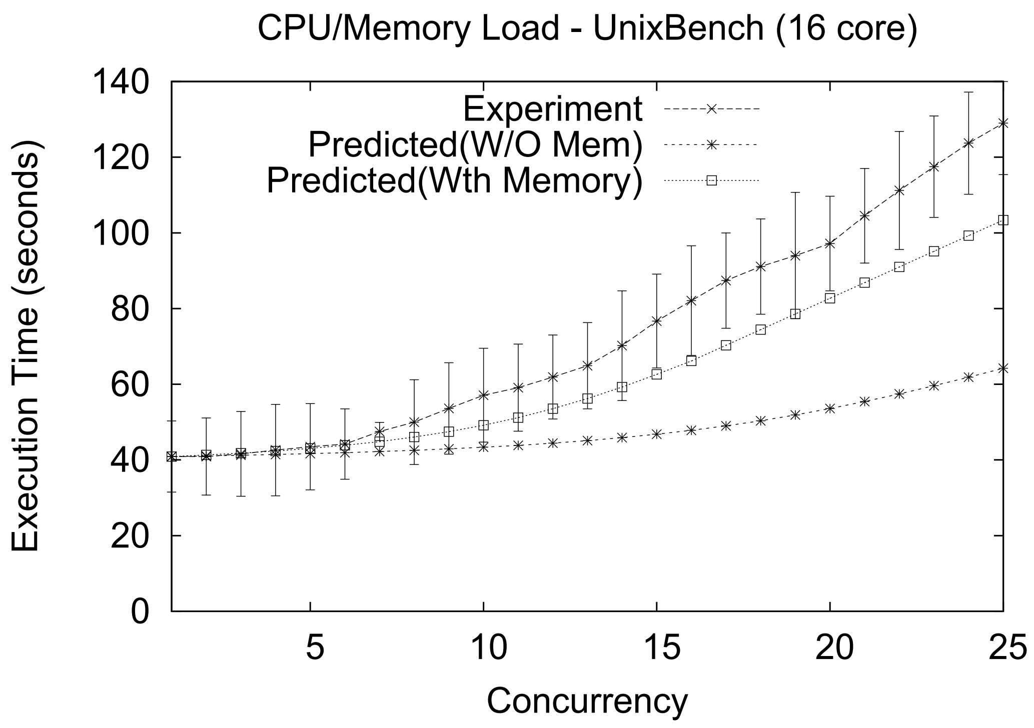


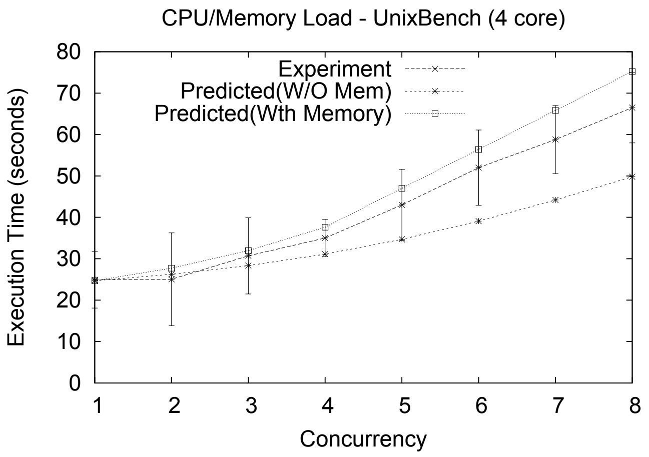
Figure 4.9: Average execution time vs. concurrency level for the Unixbench benchmark Towers of Hanoi program. Westmere (16 cores) with 10% backend stall

and 470.lbm from the Floating point benchmarks. Both benchmarks have high memory demand. The experimental results mirrored very closely (error ranges from -5% to 22%) the predicted values of our memory contention model. Figure 4.11 shows the result of running benchmark 429.mcf on a 4-core Sandy Bridge machine.

4.6 Extending the Analytic Model to Multiple Job Classes

The previous sections discussed our two-layer multi-core performance model from a single class workload perspective. We have also discussed the effects of NUMA-based machines common today. We now extend the model to multiple classes of jobs (i.e., multiple types of jobs in terms of resource demands) and concentrate on the more general NUMA-based architecture introduced in Section 4.3.2. Note that the SMP case can be treated by removing the remote memory queue from the memory contention model.

The notation used in the multi-class case model is given in Table 4.4. The multi-class model



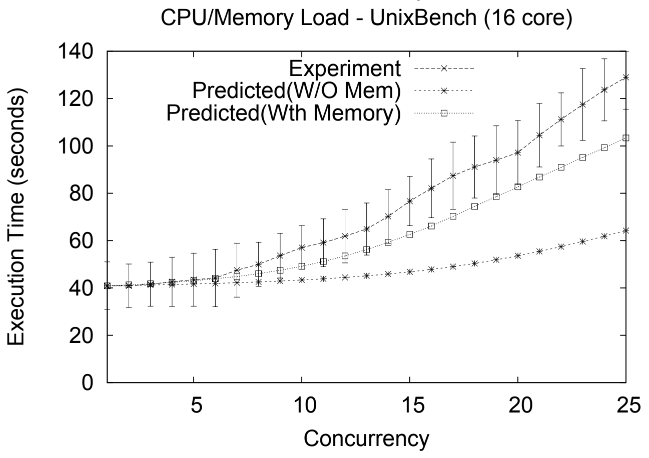


Figure 4.10: Average execution time vs. concurrency level for the Unixbench benchmark Grep program. Top: Sandy Bridge (4 cores) with 35% backend stall. Bottom: Westmere (16 cores) with 10% backend stall.

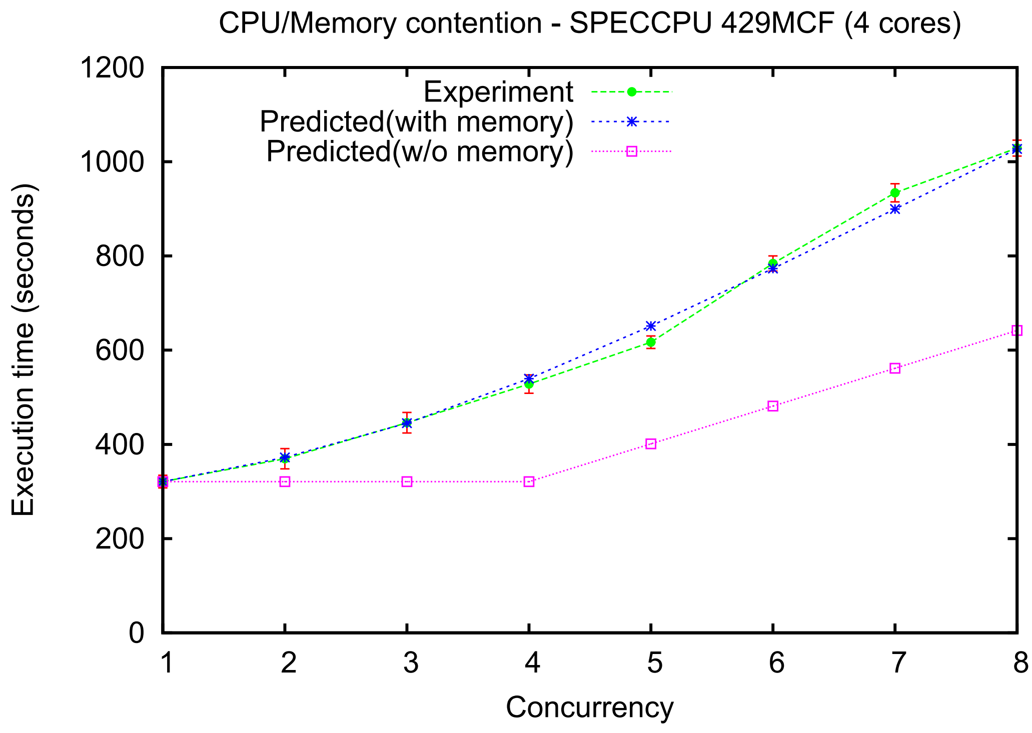


Figure 4.11: Average execution time vs. concurrency level for the SPECCPU 429.MCF benchmark; Sandy Bridge (4 cores) with 40% stall.

extension requires the use of Approximate MVA (AMVA) [64] due to the high and known computational complexity of solving exact MVA for multiple classes.

Table 4.4: Notation for Multi-class Model.

R	number of job classes
$\vec{N} = (N_1, \cdots, N_r, \cdots, N_R)$	population vector indicating the number of jobs in each class
$\vec{D}_{\mathrm{CORE-I}}$	vector of service demands for each class for device CORE-I
$\vec{D}_{\mathrm{Mem-Local}}$	vector of service demands for each class for device Mem-Local
$\vec{D}_{\mathrm{Mem-Remote}}$	vector of service demands for each class for device Mem-Remote
$\vec{D}_d$	vector of service demands for each class at the disk
$\vec{N}^{\mathrm{mem}} = (N_1^{\mathrm{mem}}, \cdots, N_r^{\mathrm{mem}}, \cdots, N_R^{\mathrm{mem}})$	vector of number of jobs of each class in the memory contention model
$\vec{T} = (T_1, \cdots, T_r, \cdots, T_R)$	vector of execution times per class for the memory contention model
$X_{0,r}(\vec{N})$	throughput of class r jobs for a job population $\vec{N}$
$\bar{n}_{\mathrm{CPU},r}(\vec{N})$	average number of class r jobs using or waiting for cores for a job population $\vec{N}$
$R'_{\mathrm{CPU},r}(\vec{N})$	average class r CPU residence time (i.e., time spent waiting or using a resource) for a job population $\vec{N}$

Algorithm 5 shows the multi-class algorithm for the memory contention model. The loop that starts at line 4 estimates the average number, N_r^{mem}, of class r jobs in the memory contention model. The total number of jobs in the memory contention model cannot exceed m, the number of cores. The question is how this number should be apportioned to each job class. We use an approximation that apportions the total number of class r jobs in the memory contention model based on the proportion of class r jobs in the population $\vec{N}$. For example, suppose that there are three job classes, 12 cores, and that the population vector $\vec{N} = (8, 12, 4)$. Then, $\vec{N}^{\mathrm{mem}} = (8/24 \times \min(12, 24), 12/24 \times \min(12, 24), 4/24 \times \min(12, 24)) = (4, 6, 2)$.

We then use $\vec{N}^{\mathrm{mem}}$ as the population vector to solve a closed QN model using AMVA. The algorithm for solving AMVA is well-known and can be found for example in [55]. Line 10 of Algorithm 5 indicates that such a model is solved using as parameters $\vec{N}^{\mathrm{mem}}$ and the service demands for all classes at the cores, local, remote memory, and I/O devices. The solution to this QN network provides the execution times, T_r, of jobs of class r in the memory model. The values in $\vec{T}$ are used in the application level model. It is also worth noting that some of the elements of the vector $\vec{N}^{\mathrm{mem}}$ may not be integers as a result of the computation in line 5 of Algorithm 5. This is not a problem because AMVA allows for non-integer population values.

The multi-class application-level model is shown in Algorithm 6. This algorithm uses the same approximation for apportioning classes to cores as Algorithm 5 and does not require the use of the B-S approximation when there is no or little I/O activity. Then, the average number, $\bar{n}_{\mathrm{CPU,r}}(\vec{N})$, of class r jobs using or waiting for cores is simply the class r population N_r (first statement in the loop). Each class throughput, $X_{0,r}(\vec{N})$, is estimated by applying Little's Law on a per class basis to the set of cores (not including the queue for cores). The average time spent by a class r job using a core (including executing instructions, waiting for, and accessing memory) is given by T_r, obtained by the memory contention model. To apply Little's Law on a per class basis to the set of cores, we estimate the average number of class r jobs using the cores as in Algorithm 5. Finally, we apply Little's Law on a per class basis to the entire system consisting of the queue for cores and the cores themselves to obtain the average class r CPU residence time, $R^{'}_{\mathrm{CPU},r}(\vec{N})$, which represents the execution time of class r jobs for a job population of $\vec{N}$.

If the I/O activity is not negligible, then the application level model can be solved using regular AVMA enhanced by the B-S approximation to capture contention for cores. The service demands

Algorithm 5 Memory contention model - multiple classes

```
    Input: $m, r, \vec{N}, \vec{D}_{\mathrm{Core-I}}, \vec{D}_{\mathrm{Mem-Local}}, \vec{D}_{\mathrm{Mem-Remote}}, \vec{D}_d$
    /* Calculate the elements of a concurrency vector whose values are
    in proportion to the population vector $\vec{N}$ */
    for $r = 1 \rightarrow R$ do
5:      $N_r^{\mathrm{mem}} \leftarrow N_r / \sum_{s=1}^{R} N_s \times \min\ (\sum_{s=1}^{R} N_s, m)$
    end for
    /* The vector $\vec{T}$, obtained by solving a closed QN model,
       provides the total time spent by jobs of each class processing
       instructions, waiting for, and accessing memory */
10: $\vec{T} \leftarrow$ SolveClosedQNAMVA ($\vec{N}^{\mathrm{mem}}, \vec{D}_{\mathrm{Core-I}},$
                $\vec{D}_{\mathrm{Mem-Local}}, \vec{D}_{\mathrm{Mem-Remote}}, \vec{D}_d$);
```

Algorithm 6 Application model - multiple classes

```
    for $r = 1 \to R$ do
        $\bar{n}_{\text{CPU,r}}(\vec{N}) \leftarrow N_r$
        /* Apply Little's Law per class to the set of cores */
        $X_{0,r}(\vec{N}) \leftarrow [(N_r \times m)/(\sum_{s=1}^{R} N_s]/T_r$
5:      /* Apply Little's Law per class to the entire system */
        $R^{'}_{\text{CPU,r}}(\vec{N}) \leftarrow \bar{n}_{\text{CPU,r}}(\vec{N})/X_{0,r}(\vec{N})$
    end for
```

at the cores is obtained from Algorithm 5. The next section discusses the experimental setup and the results of the multi-class experiments.

Table 4.5: Input parameters obtained by running the programs on NUMA based Intel XEON machines.

Parameter	Workload 1			Workload 2		
	UBench	MBench	HBench	UBench	MBench	HBench
$D_{\text{Core-I}}$	7.08 sec	14.7 sec	26.55 sec	27.94 sec	25.2 sec	8.65 sec
$D_{\text{Mem-Local}}$	0.1 sec	3.2 sec	13.0 sec	0.43 sec	5.4 sec	2.3 sec
$D_{\text{Mem-Remote}}$	0.1 sec	3.2 sec	13.0 sec	0.43 sec	5.4 sec	2.3 sec
PercStalledBackendCycles	3%	30%	50%	3%	30%	35%

4.7 Multi-class Model Results

Multi-class experiments were conducted on a 12-core, 8-GB XEON-E5-2620 NUMA server running Linux 2.6.32. We generated two different workloads, each with three job classes derived from UBench, HBench, and our own MBench by modifying the number of times the main loop is executed. This allowed us to generate different service demands for these three programs and obtain two very distinct workloads (see Table 4.5).

As in Section 4.4, memory service demands were collected using Linux's `perf`. For each workload, we ran experiments that varied the concurrency level of one of the classes while keeping constant the concurrency level of the other classes.

All figures presented in this section display experimental average execution times (with 95% confidence intervals) as well as the predicted execution times from the models with and without memory contention. Figures 4.12 and 4.13 show similar results. The top (HBench) and middle

(MBench) graphs show that the analytic model with memory contention tracks very well the experimental results while the model that does not consider memory contention does a very poor job at estimating the average execution time. For example, for HBench and a concurrency level of 12, the model with memory contention predicts an average execution time of 245 sec and the model without memory contention predicts 93 sec. The average measured execution time is 210 sec $\pm$ 10 sec. So, the model with memory contention predicts an execution time 11% above the upper bound of the 95% confidence interval for the measured values. The model without memory contention predicts an average execution time that is 46.6% of the lower bound of the 95% confidence interval for the measured values.

There is no significant difference between the models with and without memory contention for UBench (bottom curves in the graphs of Figs. 4.12 and 4.13) because UBench has a very small (i.e., 3%) value for `PercStalledBackendCycles` (see Table 4.5). It is to be noted that when the total number of jobs is less than the number of cores, there is no contention in the model which does not consider memory contention. However, when the total number of jobs exceeds the number of cores, we use two approximations for the model without memory contention: AMVA as well as BS. This explains the sharp jump in execution time from concurrency 4 to concurrency 5 in the multi-class graphs when there is no disk demand for the "w/o memory" lines in the graphs. At that concurrency level, all three classes will have a concurrency level of four for a total number of jobs equal to 12, the number of cores.

We also ran multi-class experiments with disk activity on a 4-core machine with the following job classes: MBench and HBench modified with disk load and an unmodified UBench (no disk activity). See service demands for these experiments in Table 4.6. Figure 4.14 shows the measured average execution times for MBench, the predicted execution times for the models with and without memory contention. As the graph shows, the model with memory contention predicts the average execution time with much better accuracy (the results are within the confidence intervals for the measured values in most cases) than the model that does not consider memory contention (the results are always outside of the confidence intervals).

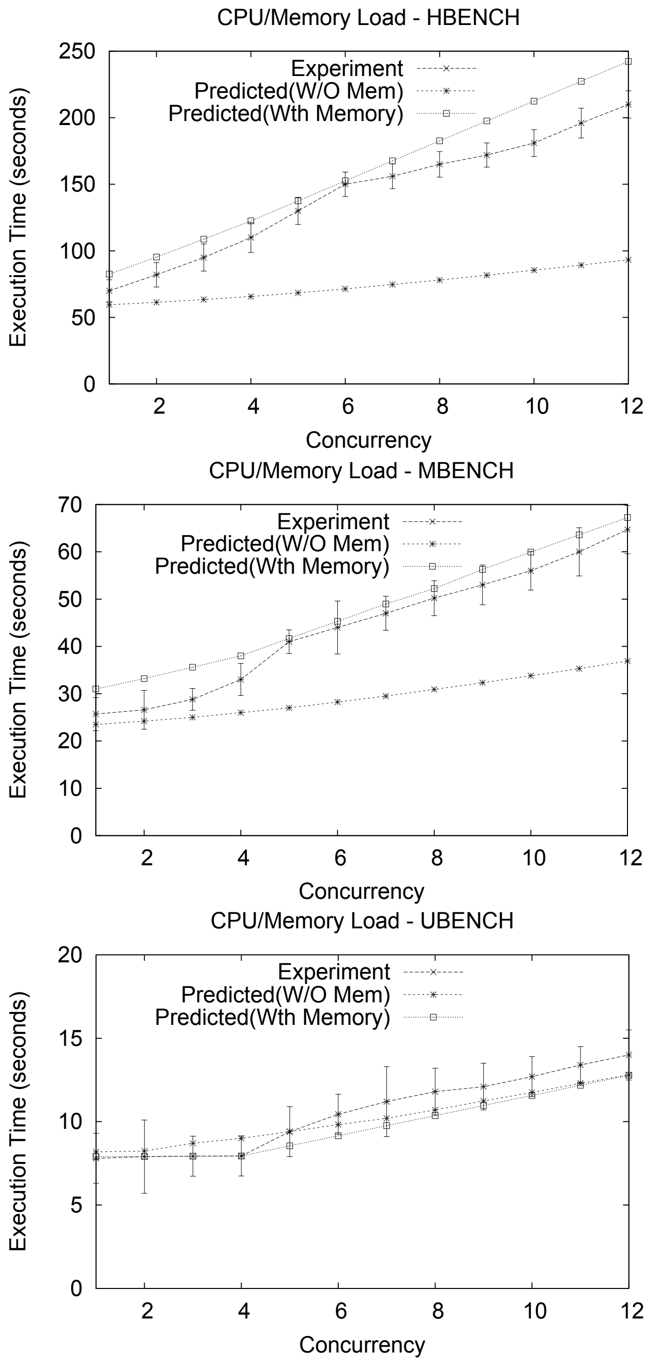


Figure 4.12: Average execution time vs. concurrency level for HBench, MBench and UBench for Workload 1. Top: UBench and MBench fixed at concurrency 4. Middle: UBench and HBench fixed at concurrency 4. Bottom: HBench and MBench fixed at concurrency 4.

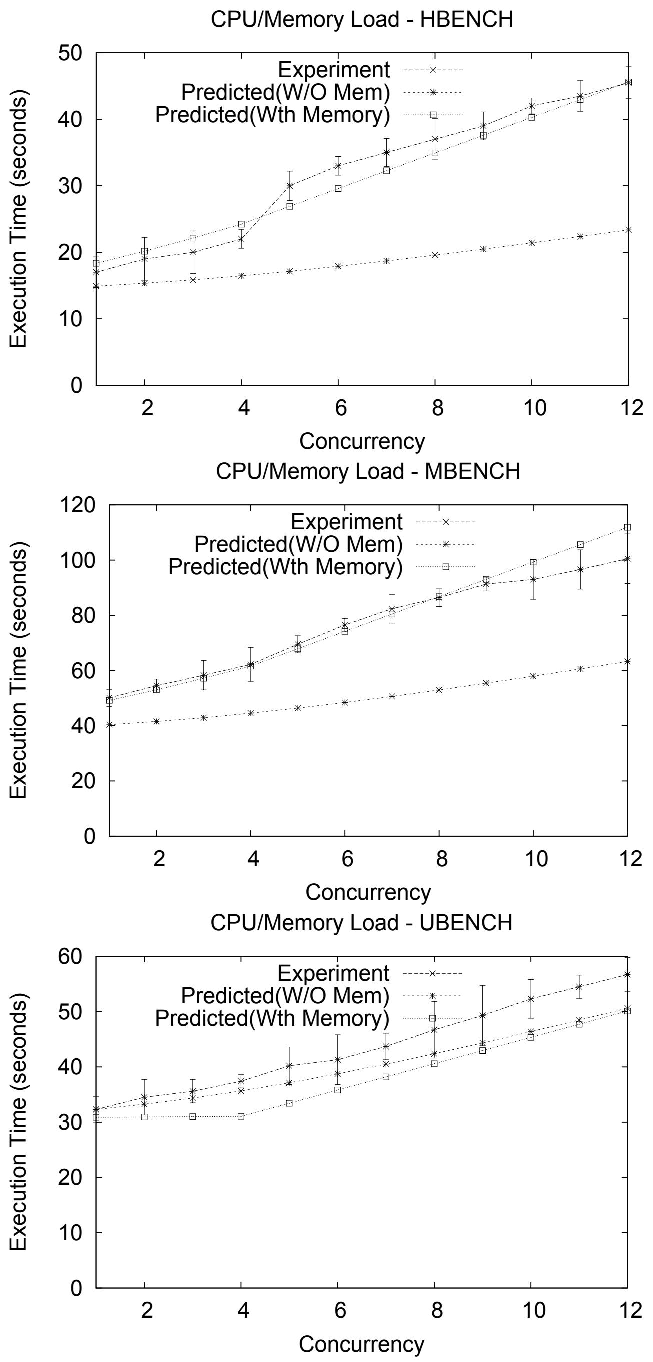


Figure 4.13: Average execution time vs. concurrency level for HBench, MBench and UBench for Workload 2. Top: UBench and MBench fixed at concurrency 4. Middle: UBench and HBench fixed at concurrency 4. Bottom: HBench and MBench fixed at concurrency 4.

Table 4.6: Input parameters for the multi-class experiments with disk activity on a 4-core machine.

Parameter	UBench	MBench	HBench
$D_{\mathrm{Core-I}}$	10.8 sec	22.3 sec	71.2 sec
D_{Disk}	0 sec	2.5 sec	4.0 sec
PercStalledBackendCycles	3%	30%	50%

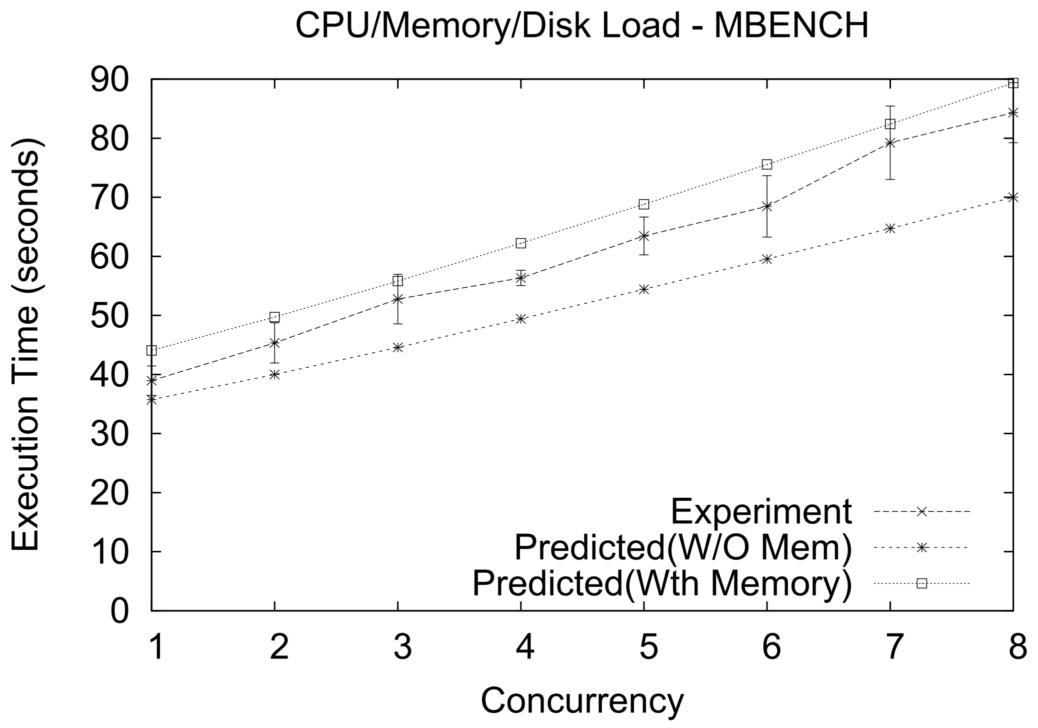


Figure 4.14: MBench average execution time vs. Mbench concurrency level. UBench and HBench fixed at concurrency 4.

4.8 Related Work

Approximate MVA models for closed QNs with multiple-server queuing stations were developed in [2, 65, 70]. These models consider that all resources in a multi-server queuing station have the same processing speed regardless of concurrency level and do not consider the effects of memory contention. Three operational laws for a single queue with parallel servers were derived by Kelly et. al. [44]. These results also do not consider any contention for memory resources.

Fedorova et. al. discuss the possible reasons for contention in multi-core systems and provide new methods for mitigating contention through scheduling algorithms [29]. The authors in [68] use a control theory approach to partition processing cores, shared cache, and off-chip memory bandwidth between concurrently executing applications. A global resource broker is used to manage per application demand. Levesque et. al. discuss the AMD chips and also discuss methods to collect performance data and use that data to analyze the performance impact of multi-core processors [49]. That work uses micro benchmarks and other benchmarks to explore compiler switches for software optimization. In [27], the author argues that performance counters available on all major current processors are crucial to the understanding of hardware performance. The paper describes Linux-based simple monitoring tools that allow one to pinpoint key bottlenecks in applications. The paper in [84] examines several Java applications and concludes that saturating the memory bandwidth of a multi-core system essentially degrades scalability. The authors postulate that Java's garbage collection scheme is detrimental on multi-core platforms.

Chen et. al. [16] study applications running on multi-core machines and examine the question of whether one can maximize resource usage while respecting application performance goals. The authors use queuing theory to predict the scalability of applications on multi-core machines. They do not however include memory contention in their model.

Towsley [73] provided a discrete time Markov model solution to modeling multiple processors and multiple bus/memory configurations. His methodology consisted of three steps: a) Replace the memory systems with a single aggregate queue whose service rate reflects the behavior of the memory system when there is no bus contention; b) Modify this queue to add contention; and c) Solve the resulting 2-queue system with CPU and memory. That work, which was done in 1983, has some similarities with our work in the sense that it also collapses the entire cache/memory system into a single queue. However, there are some significant differences: a) Unlike the work in [73] that uses simulation and exact solutions for validation, we use experiments, b) The model in [73] is single class while ours includes a multi-class formulation, c) Our solution has small computational and storage requirements because we rely on AMVA, and d) We use hardware performance counters to parameterize our model, making it more applicable.

In [19], the authors modeled shared bus multiprocessors using MVA and compared model results with trace-driven simulation results. The authors used `awk`, `compress`, `nroff`, and `sort` as benchmark programs. They focus their work on cache misses and collected memory trace data for the benchmark programs to generate bus request statistics. These statistics were then fed into their Customized MVA (CMVA) algorithm. That paper does not specifically mention how to handle multiple classes of jobs and does not compare analytic results with experimental results.

The authors in in [25] provided MVA models for CPU, I/O, and memory. They provided several what-if scenarios and discussed cache coherency. The results are based on simulation and not actual experimentation. That paper presented a detailed study on the I/O portion of the total execution time of a process and also studied the architectural effects of symmetric multi-processing machines on TPC-C (www.tpc.org) benchmarks for a single class environment.

Lastly, in [69] the authors developed and validated an analytic model for evaluating shared-memory systems with ILP processors. Even though this work predates multi-core computers, it shows how application performance can be studied and inferences drawn by using analytical models. One of the main contributions of that paper is a trace-driven simulation method to generate parameters an order of magnitude faster than a detailed execution-driven simulator.

4.9 Concluding Remarks

Memory contention in multi-core machines may be a significant portion of the execution time of an application, especially for machines with a large number of cores. This chapter presented a two-level approximate single- and multi-class queuing network analytic model to predict the execution time of applications running on multi-core machines. The first level captures contention for memory and the other incorporates that contention into an application-level model that captures contention for cores. Our model also considers SMP and NUMA architectures.

We used hardware counters provided by modern Intel processor chips to obtain the parameters for the memory contention model. To that end, we focused on back-end-stall cycles, which provide a good approximation for cycles spent by the CPU due to memory access during the load/store phase of instruction execution.

We developed a micro-benchmark written in C that allowed us to vary the amount of memory allocation, the amount of memory accessed, and the extent of I/O activity. Linux's heavy use of buffer cache, whereby the OS fetches complete file data if system memory is available, required us to flush the cache buffer before every run. We intend to investigate the effect of this eager fetch of file data on the performance model. We have also ran experiments using well-known Linux benchmarks, HBench, UnixBench and SPECCPU2006 to further validate our model.

Our many experiments showed that ignoring the effect of memory contention when modeling applications with any amount of appreciable memory demand produces erroneous results. The results of our single and multiple-class experiments demonstrate that our model is capable of estimating with an acceptable accuracy the execution time of jobs running on multi-core machines. Moreover, we showed how one can easily obtain input parameters to our model using data readily available in the performance counters of modern processors.

Our experiments used single-threaded applications. Detail investigation of the impact of multi-threaded applications is left as future work. This may require using QN models of software contention [55]. Our work is very relevant for Hadoop-like systems [90] wheret most cluster nodes are multi-core computers. The models presented here are used to predict the execution time of MapReduce jobs as explained in the following chapters of this dissertation.

Chapter 5: Epochs - An algorithm for calculating execution times for finite time interval

5.1 Introduction

The work described in this chapter is published in [20]. Since this dissertation deals with the problem of design and modeling of schedulers responsible for job scheduling, one problem that needs solving is the ability to calculate the execution time of a job, when other jobs join and leave the executing node. This section discusses the use of closed Queuing Network (QN) models during finite intervals to estimate the execution time of jobs submitted to a computer system. Queuing theory has extensively studied the problem of estimating job execution times in steady state conditions both in the case of single queues or queuing networks (see [9,46,55,63]). Early results on queuing theory were derived assuming certain stochastic assumptions (e.g.,steady-state equilibrium, Poisson arrivals, exponentially distributed service times). Some of these results were later generalized to more general distributions [9], but still the steady-state equilibrium assumption was required in these cases.

However, the results obtained under stochastic assumptions proved to be quite robust even when these assumptions were violated. Buzen explained why in his formulation of operational analysis of computer system performance [13]. Operational analysis establishes mathematical relationships between variables that can be measured during a finite time interval. If the relationships are always true they are called operational laws and if they require some assumptions they are called operational theorems. The validity of the assumptions in operational theorems can also be assessed by taking measurements during the same finite interval during which the relationships are established. Examples of operational assumptions are: (1) Flow Balance (i.e., the number of arrivals is equal to the number of departures during a given time interval, (2) Homogeneous Arrivals (i.e., the arrival rate does not depend on the queue size), (3) Homogeneous Service Times (i.e., the mean time between completions does not depend on the queue size), and (4) One-Step Behavior (i.e., a queue length can only vary by increments of $\pm$ 1) [13]. Note that steady-state implies flow balance but the converse is not true [14]. Buzen and Denning showed that most real systems satisfy homogeneity approximately [14].

Buzen and Denning have derived in [12,14] the operational counterpart of the Mean Value Analysis (MVA) [63] equations for solving closed QN models. They showed that the MVA equations are valid for finite time intervals if flow balance, one-step behavior, and homogeneous service times are met. As will be seen later in this section, this is a key aspect on which this work is dependent upon.

The steady state treatment of queuing systems considers open or closed systems. In the case of open queuing systems, the job arrival process is characterized by an inter-arrival time distribution and in the case of closed systems, the workload is characterized by a job population vector that indicates the steady state number of concurrent jobs of each class. Contention for resources among jobs leads to waiting times that are used to determine the steady state execution time of jobs (either average and in some cases higher moments or distributions).

This work considers a different kind of problem, the understanding of which is better illustrated with the help of Fig. 5.1. The figure shows a computer system (top right) and a model of that system (bottom right), which represents the processors, I/O devices and their respective queues. The system model can be solved using simulation techniques [66] or analytic models such as analytic queuing networks [9,55,63].

The left-hand side of Fig. 5.1 shows three typical methods for characterizing the arrival process of jobs to a computer system. Method (a) consists of a job trace that can be either replayed against the real system in order to derive actual measurements or used as input in trace-driven simulation studies. Method (b) consists of generating job inter-arrival times (and their features) as random numbers that follow a desired probability density function. This is a typical method used in discrete event simulation. Finally, method (c) considers the mathematical expression that characterizes job inter-arrival time distributions. This method is used in deriving solutions for analytical models of computer systems.

This body of work shows how to combine methods (a) and (b), typically used in simulation studies, with analytic system models. The advantages of this approach are:

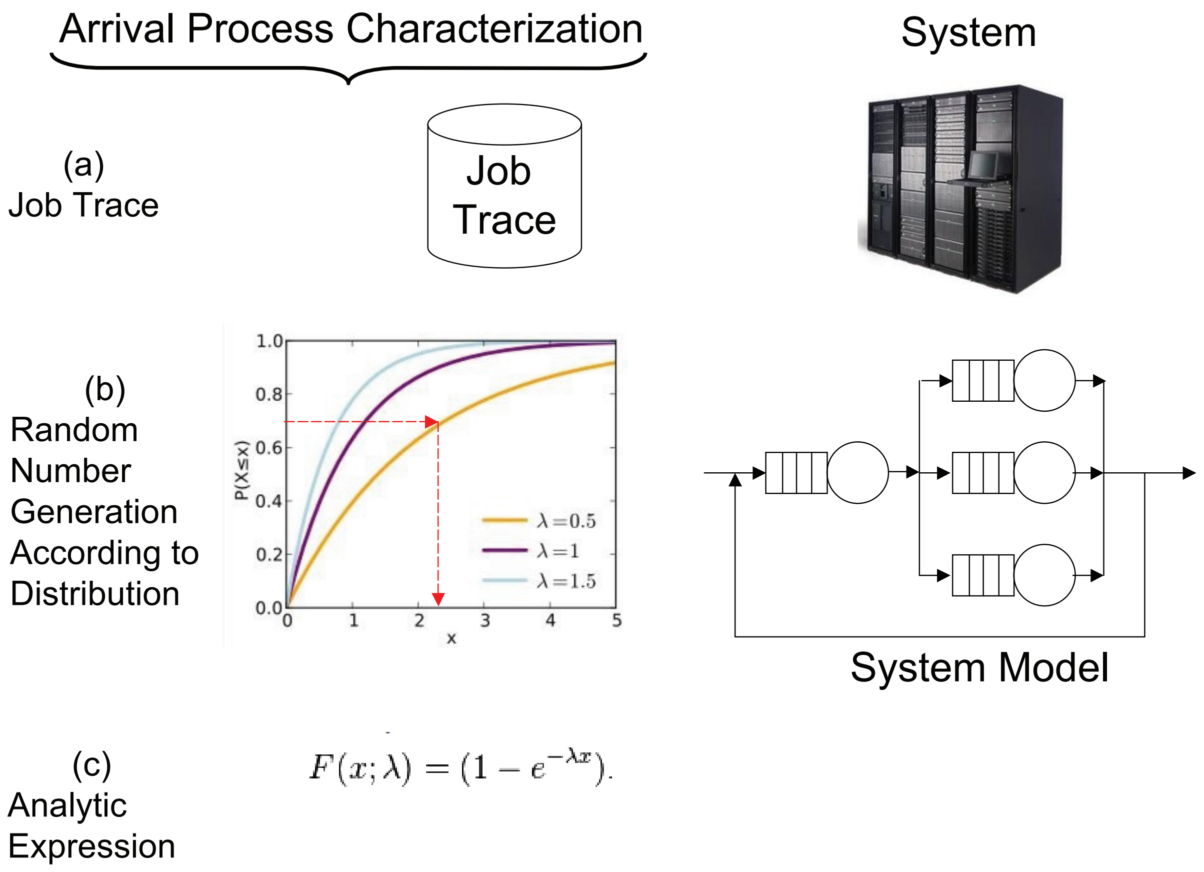

Arrival Process Characterization
System
(a) Job Trace
Job Trace
(b) Random Number Generation According to Distribution
P(X≤x)
x
λ=0.5
λ=1
λ=1.5
System Model
(c) Analytic Expression
$F(x;\lambda) = (1 - e^{-\lambda x})$.

Figure 5.1: Methods for characterizing arrival processes in modeling.

- The system model can be solved through efficient analytic methods (e.g., Mean Value Analysis) instead of more complex and time-consuming simulations.
- The analytic models can be employed in situations for which they were not designed for (e.g., using a job trace as input). Note that the conventional approach for dealing with job traces as input to analytic models requires processing the trace in order to fit a known distribution for the job inter-arrival times. A difficulty may arise if this distribution does not meet the assumptions required by the analytic model.

An algorithm is presented here which is called the *Epochs* algorithm, that estimates the execution times of jobs in a job stream. These jobs are executed by a computer system and contend for its resources. The algorithm is validated against experimental results using both jobs derived from a micro-benchmark and well-known benchmarks. The validation shows that the absolute relative error between measurements and predictions by the Epochs algorithm is below 10% most of the time and is at most 15%. These errors are considered to be acceptable for execution time prediction.

The rest of this discussion is organized as follows. Section 5.2 describes the notation and formalizes the problem description. Section 5.3 presents the Epochs algorithm. The next section discusses validation results with a micro-benchmark and with three UNIX benchmark programs. Section 5.5 shows how the Epochs algorithm can be used to assess schedulers in a cluster of computers. Section 5.6 discussed the computational complexity of the Epochs algorithm. Section 6.8 discusses related work. Finally, section 5.8 presents some concluding remarks.

5.2 Notation and Problem Description

Consider a known stream $\mathcal{S} = \{(J_1, t_1), \cdots, (J_n, t_n), \cdots, (J_N, t_N)\}$ of jobs J_n that arrive at times $t_n, (n = 1, \cdots, N)$. Let $t_1 \leq t_2 \leq \cdots t_N$ without loss of generality. Each job J_n is characterized by a vector $\vec{D}_n = (D_{1,n}, \cdots, D_{K,n})$ of service demands at resources $1, \cdots, K$. The service demands of a given type of job are not deterministic but are average values obtained during multiple executions of each type of job. The differences in service demands for each job type in the stream may be attributed to differences in the execution path due to differences in the input data and/or due to a variability in the conditions in which measurements were taken.

The number of jobs in the stream is finite and the job arrival instants are assumed to be known as the Epochs algorithm is presented. This latter assumption is later relaxed. There are no steady state considerations. However, similarly to the steady state analysis, there is contention for the use of resources and this contention has to be considered in order to compute the execution times of the jobs in the stream.

The problem described here could be addressed by using trace-driven discrete simulation in which $\mathcal{S}$ is the input trace. In this approach, arriving jobs join queues, receive service at the various simulated system resources, and leave after receiving all the required service at the system resources. The service demands at each resource are drawn from some distribution at each job arrival. The approach presented in this section replaces the stochastic simulation of job behavior at the computer system by analytic models. However, job arrivals is still “trace-driven” or obtained through random number generation.

Figure 5.2 illustrates the concepts presented here. Time is divided into time intervals of finite duration called *epochs*. The first epoch starts by definition at t_1, the time at which the first job(s) arrive. The end of an epoch is characterized by one of two events: (1) the arrival of a new job or jobs (if more than one job arrives at the same time) or (2) the completion of a job. The last epoch ends when the last job in execution ends. Therefore, each epoch has a constant workload mix, i.e., a set of jobs running concurrently.

Let us denote the subsequent epochs as $E_1, \cdots, E_i, \cdots, E_M$ and let $e(E_i)$ be the end time of epoch E_i $(i = 1, \cdots, M)$ such that $e(E_1) < e(E_2) < \cdots < e(E_M)$. Let $e(E_0)$ be defined as t_1. The end time of epoch i is the start time of epoch $i + 1$ for $i = 1, \cdots, M - 1$. The time it takes to execute all jobs in the stream $\mathcal{S}$ is $e(E_M) - e(E_0)$.

With respect to Fig. 5.2, jobs 1-4, arrive at times t_1, t_2, t_3, and t_4, respectively, and end at times $ej(J_1), ej(J_2), ej(J3)$, and $ej(J_4)$, respectively. There are seven epochs: $E_1, \cdots, E_7$.

Let $\mathcal{W}_i = \{J^i_{i_1}, J^i_{i_2}, \cdots, J^i_{j_i}\}$ be the workload mix during epoch E_i. The jobs in $\mathcal{W}_i$ compete for the use of the K resources. Therefore, during each epoch, the jobs in that epoch's workload mix spend some time using the system resources (i.e., spending some of their service demands at these resources) and waiting to use these resources. The fraction of its service demand that a job is able to use during a given epoch is a function of the contention it finds from the other jobs in that workload mix.

The workload mixes at the various epochs of Fig. 5.2 are: $\mathcal{W}_1 = \{J_1\}; \mathcal{W}_2 = \{J_1, J_2\}; \mathcal{W}_3 = \{J_1, J_2, J_3\}$; $\mathcal{W}_4 = \{J_1, J_2, J_3, J_4\}; \mathcal{W}_5 = \{J_2, J_3, J_4\}; \mathcal{W}_6 = \{J_3, J_4\}$; and $\mathcal{W}_7 = \{J_4\}$. Thus, the concurrency levels at the various epochs of Fig. 5.2 are 1, 2, 3, 4, 3, 2, and 1, respectively.

The problem which needs to be solved is: Given a stream of jobs $\mathcal{S}$ and their corresponding service demands, find the execution time of each job in $\mathcal{S}$. The execution time of a job J_n is the difference between its end time $ej(J_n)$ and its start time $sj(J_n)$. Note that both $ej(J_n)$ and $sj(J_n)$ must coincide with an epoch transition time because a new epoch starts when a job arrives or completes. Clearly, the execution time of a job depends on how much service at each device the job can accomplish within each epoch. The accomplished service demand within an epoch depends

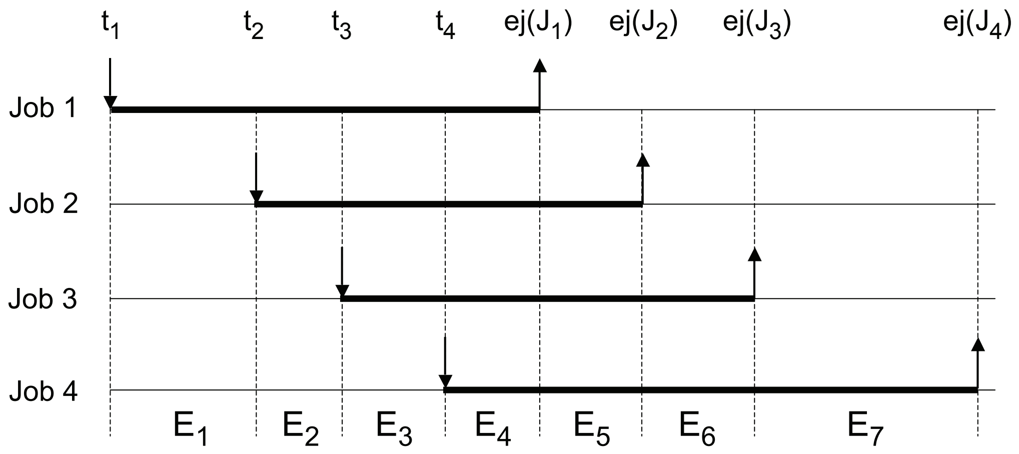


Figure 5.2: Concept of Epochs.

on the contention at various resources caused by the jobs that are part of the workload mix of that epoch.

Some additional notation is given below:

- $d(E_i)$: duration of epoch i.
- $se(J_n)$: index of start epoch of job J_n. For example, $se(J_2) = 2$ in Fig.5.2. Note that $sj(J_n) = e(E_{se(J_n)-1})$.
- $ee(J_n)$: index of end epoch of job J_n. For example, $ee(J_2) = 5$ in Fig.5.2. Note that $ej(J_n) = e(E_{ee(J_n)})$.
- $et(J_n)$: execution time of job J_n. $et(J_n) = ej(J_n) - sj(J_n)$.
- $D^i_{k,n}$: portion of service demand at resource k for job J_n accomplished during epoch E_i. Then,

$$D_{k,n} = \sum_{i=se(J_n)}^{ee(J_n)} D^i_{k,n} \tag{5.1}$$

- $R^i_{k,n}$: residual service demand at resource k for job J_n at the beginning of epoch E_i

5.3 The Epochs Algorithm

Explained now is the estimation of $ej(J_n)$ using the *Epochs* algorithm described in this section. The process consists of estimating $D^i_{k,n}$ for every job J_n in the workload $\mathcal{W}_i$ for epoch E_i. This value can be obtained by solving a multi-class closed QN model for epoch E_i. The parameters for this model are as follows:

- R_i: number of classes for the model in epoch E_i. $R_i = \mid \mathcal{W}_i \mid$. Each class corresponds to exactly one job in $\mathcal{W}_i$.

- $\vec{N}_i = (N_1, \cdots, N_{R_i})$: population vector for the QN model for epoch E_i. By definition, this vector is equal to $(1, \cdots, 1)$ because each class corresponds to exactly one job in $\mathcal{W}_i$.

- $\vec{R}^i_n$: vector of residual service demands for job J_n at the beginning of epoch i. This vector can be computed as the difference between the original vector of service demands for that job and what the job has already accomplished in terms of its service demands from its starting epoch until the epoch preceding epoch E_i.

$$\vec{R}^i_n = \vec{D}_n - \sum_{v=1}^{i-1}(D^v_{1,n}, \cdots, D^v_{K,n}) \tag{5.2}$$

- $\mathbf{R}_i$: matrix of residual service demands for the model at epoch i. Each column corresponds to a job in $\mathcal{W}_i$ and each row corresponds to each of the K resources. The values in column n that correspond to job J_n come from the vector $\vec{R}^i_n$.

- $\mathcal{M}(R_i, \vec{N}_i, \mathbf{R}_i)$: closed QN model with parameters R_i, $\vec{N}_i$ and $\mathbf{R}_i$. The solution to this model returns the execution times $T_{i,r}$ of class r $(r = 1, \cdots, R_i)$ (i.e., the execution time of the job corresponding to class r).

The execution times $ej(J_n) - sj(J_n)$ for all jobs in $\mathcal{S}$ are computed through Algorithm 7, the Epochs algorithm. The inputs to the algorithm are the job stream, $\mathcal{S}$, and the vector of service demands $\vec{D}_n$ for each job J_n. Lines 5-6 initialize the epoch count i, the workload mix for epoch

E_1, the value of the variable LastArrival to the earliest job arriving time (i.e., t_1), and the residual service demand values as equal to the service demands for all jobs in epoch 1.

Then, the algorithm loops (lines 7-50) while the set of jobs in the workload for epoch i is not empty. Line 13 invokes an AMVA solver (see e.g., [55]) during epoch i, a finite time interval, to compute the execution times of the jobs present at the beginning of that epoch. As indicated above, the MVA equations are valid for finite intervals if flow balance, one-step behavior, and homogeneous service times are satisfied. According to Buzen and Denning, many real systems satisfy service homogeneity assumptions [14]. One step-behavior is satisfied by our definition of an epoch, which starts when either a new job arrives or a job leaves. Flow balance is also satisfied by definition because the workload mix is constant during any epoch, which implies that no jobs arrive or leave during an epoch.

In line 15, the algorithm computes the minimum execution time (MinEnd) of the jobs executing in the current epoch and, in line 16, the NextArr function is used to return the next job arrival time (NextArrival) by inspecting the job stream $\mathcal{S}$ after instant LastArrival.

The algorithm then determines if the current epoch ends due to a job completion (line 19) or due to a job arrival (line 33) by comparing the values of MinEnd and LastArrival. In the former case, the end time of the job(s) completing at the end of the epoch is computed (line 24) and the accomplished demand for all jobs in the workload mix during the current epoch is computed in line 26 as:

$$D^i_{k,n} = R^i_{k,n} \cdot \frac{e(E_i) - e(E_{i-1})}{T_{i,n}} \;\; \forall\, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i \tag{5.3}$$

Equation (5.3) says that the fraction of service demand accrued by a job during an epoch is proportional to the proportion of the total execution time of that job with respect to the epoch duration.

The workload for the next epoch is computed by removing the job(s) that completed from the workload of the current epoch (line 31). In the latter case, the end time of the current epoch is set to the arrival time of next job(s) to arrive in the job stream (line 34). As before, the accomplished demand for all jobs in the workload mix during the current epoch is computed (line 38) as in line 26 and the workload for the next epoch is computed by adding to the current workload the arriving job(s) (line 43). Lines 45-49 take care of the case in which there is an idle period in the job stream.

Table 5.1 illustrates the execution of the algorithm on a simple example with two jobs J_1 and J_2 and three epochs. Job J_1 has starting service demands at the CPU and disk equal to 2 sec and 4 sec, respectively. The starting service demands at the CPU and disk for job J_2 are 3 sec and 5 sec, respectively. Row 2 of the table indicates the event that triggers the start of an epoch. Row 3 indicates the workload mix at each epoch. Row 4 indicates the duration of each epoch. Row 5 shows the end time of each epoch. Rows 6 and 7 and 11 and 12 show the residual service demands at the CPU and disk for jobs J_1 and J_2, respectively, at the start of each epoch. Rows 8 and 13 illustrate the execution times of jobs J_1 and J_2, respectively, as if no other event were to start a new epoch and these jobs would continuously execute as indicated by the workload mix for that epoch. Rows 9 and 10 and 14 and 15 indicate the accomplished service demands for jobs J_1 and J_2, respectively, at each epoch.

Table 5.1: Example of the operation of the Epochs Algorithm

Row No.		Epoch 1	Epoch 2	Epoch 3
2	Start Event	Arr J1	Arr J2	End J1
3	$\mathcal{W}_i$	J1	J1, J2	J2
4	$d(E_i)$	3	4.69	4.98
5	$e(E_i)$	3	3+ 4.69 = 7.69	7.69 + 4.98 = 12.67
6	$R^i_{\text{cpu},1}$	2	$2-1=1$	-
7	$R^i_{\text{disk},1}$	4	$4-2=2$	-
8	$T_{i,1}$	6	4.69	-
9	$D^i_{\text{cpu},1}$	$2\times 3/6=1$	$1\times 4.69/4.69=1$	-
10	$D^i_{\text{disk},1}$	$4\times 3/6=2$	$2\times 4.69/4.69=2$	-
11	$R^i_{\text{cpu},2}$	-	3	3 - 1.13 = 1.87
12	$R^i_{\text{disk},2}$	-	5	5 - 1.89 = 3.11
13	$T_{i,2}$	-	12.44	4.98
14	$D^i_{\text{cpu},2}$	-	$3\times 4.69/12.44=1.13$	$1.87\times 4.98/4.98=1.87$
15	$D^i_{\text{disk},2}$	-	$5\times 4.69/12.44=1.89$	$3.11\times 4.98/4.98=3.11$

Epoch E_1 ends due to the arrival of job J_2 at time equal to 3 sec. Epoch 2 now starts and jobs J_1 and J_2 execute simultaneously during a period of time. During epoch E_1, job J_1 was able to accomplish half of its service demands at the CPU and disk. The residual service demands for that job at the beginning of epoch E_2 are 1 sec and 2 sec, respectively. The solution of the AMVA

Algorithm 7 Epochs Algorithm: Compute Execution Times for a Job Stream

```
Inputs: $\mathcal{S} = \{(J_1, t_1), \cdots, (J_n, t_n), \cdots, (J_N, t_N)\}$ and
$\vec{D}_n = (D_{1,n}, \cdots, D_{K,n}) \; \forall \; J_n$
Output: $et(J_n) \; \forall J_n \in \mathcal{S}$
/* Initialization */
$i \leftarrow 1; \mathcal{W}_i = \{J_n \mid (J_n, t_1) \in \mathcal{S}\}$; LastArrival $\leftarrow t_1$
$R^1_{k,n} \leftarrow D_{k,n} \;\; \forall \, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i$
while $\mathcal{W}_i \neq \emptyset$ do
    /* Build matrix of service demands for epoch $i$ */
    $\vec{D}^i_n \leftarrow \vec{D}_n - \sum_{v=1}^{i-1}(D^v_{1,n}, \cdots, D^v_{K,n}) \; \forall \; J_n \in \mathcal{W}_i$
    $\mathbf{D}_i \leftarrow \mathrm{BuildDemands}(\vec{D}^i_n \; \forall \; J_n \in \mathcal{W}_i)$
    /* Solve the closed QN model for epoch $i$ */
    $R_i \leftarrow \mid \mathcal{W}_i \mid$
    $(T_{i,1}, \cdots, T_{i,R_i}) \leftarrow \mathcal{M}(R_i, \vec{N}_i, \mathbf{D}_i)$
    /* Determine end of epoch $i$ */
    MinEnd $\leftarrow \min_{r=1}^{R_i} T_{i,r}$ /* minimum execution time */
    NextArrival $\leftarrow$ NextArr (LastArrival, $\mathcal{S}$) /* next job arrival time */
    LastArrival $\leftarrow$ NextArrival
    if NextArrival $>$ MinEnd then
        /* epoch $i$ ends due to job completion */
        $e(E_i) \leftarrow e(E_{i-1}) +$ MinEnd
        /* Find set of completing jobs */
        EndingJobs $\leftarrow \{J_n \mid J_n \in \mathcal{W}_i \wedge T_{i,n} = \mathrm{MinEnd}\}$
        /* Compute end time of ending jobs */
        $et(J_n) \leftarrow e(E_i) - t_n \; \forall J_n \in$ EndingJobs
        /* Compute accomplished demand for all jobs in $\mathcal{W}_i$ */
        $D^i_{k,n} \leftarrow R^i_{k,n} \times [e(E_i) - e(E_{i-1})]/T_{i,n}$
                $\forall \, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i$
        /* Update residual service demands */
        $R^{i+1}_{k,n} \leftarrow R^i_{k,n} - D^i_{k,n} \;\; \forall \, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i$
        /* Adjust workload mix for next epoch */
        $\mathcal{W}_{i+1} \leftarrow \mathcal{W}_i -$ EndingJobs
    else
        /* Epoch $i$ ends due to new jobs arrivals */
        $e(E_i) \leftarrow$ NextArrival
        /* Find set of next jobs to arrive */
        ArrivingJobs $\leftarrow \{J_n \mid (J_n, t_n) \in \mathcal{S} \wedge t_n = \mathrm{NextArrival}\}$
        /* Compute accomplished demand for all jobs in $\mathcal{W}_i$ */
        $D^i_{k,n} \leftarrow R^i_{k,n} \times [e(E_i) - e(E_{i-1})]/T_{i,n}$
                $\forall \, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i$
        /* Update residual service demands */
        $R^{i+1}_{k,n} \leftarrow R^i_{k,n} - D^i_{k,n} \;\; \forall \, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i$
        /* Adjust workload mix for next epoch */
        $\mathcal{W}_{i+1} \leftarrow \mathcal{W}_i \bigcup$ ArrivingJobs
    end if
    $i \leftarrow i + 1$ /* increment epoch count */
    if $(\mathcal{W}_i = \emptyset) \wedge (\mathrm{NextArrival} \leq t_N)$ then
        /* there is an inactive period before the next epoch */
        $\mathcal{W}_i \leftarrow \{J_n \mid (J_n, \mathrm{NextArrival}) \in \mathcal{S}\}$
    end if
end while
```

model for epoch E_2 indicates that the execution times for jobs J_1 and J_2 are 4.69 sec and 12.44 sec, respectively. Because no other job arrives, epoch E_2 ends when job J_1 ends at time 3 + 4.69 = 7.69 sec. The duration of epoch E_2 is then 4.69 sec. The residual service demands at the CPU and disk for job J_2 are 1.87 sec and 3.11 sec respectively at the start of epoch E_3. During that epoch, job J_2 runs by itself and takes 4.98 sec to complete. Thus, the execution time of job J_2 is 4.69 + 4.98 = 9.67 sec.

The Epochs algorithm was described as requiring that the entire job stream $\mathcal{S}$ be known in advance, as is the case of a trace in a trace-driven simulation. Minor modifications in the algorithm allow the job arrival process to be determined through stochastic generation of job types and job inter-arrival times from a given distribution as would be done in a typical discrete event simulation. The modifications are as follows:

- Remove the requirement that the job stream $\mathcal{S}$ be known as input, but continue to require that the job types be known and randomly generated and that their service demands be known.
- Replace line 16 of the Epochs algorithm by NextArrival $\leftarrow$ NextArr (LastArrival, IntArrival-TimeDistrib) where the function NextArr determines the next arrival instant by generating a job inter-arrival time from a given distribution IntArrivalTimeDistrib and adding the generated value to LastArrival.

5.4 Experimental Validation

As a first validation of the Epochs algorithm, a micro-benchmark program was written in C to provide us control and flexibility in designing jobs with varying characteristics (see pseudo-code in Algorithm 8). The program alternates between writing to a file and performing CPU operations (in this case, computing π using the Monte Carlo method). The main loop is repeated Repeat-Count times and within each loop 50% of the time I/O is done and the other 50% a CPU-Intensive computation takes place.

The micro benchmark was parameterized to generate four different types of jobs. Each job was run in isolation 100 times and average CPU and I/O service demands were computed. Two hardware configurations were used to run the experiments with the benchmark:

Algorithm 8 Micro-benchmark Pseudo Code

```
    Input: RepeatCount
    /* Open a temp file in direct and truncate mode */
    f ← OpenTempFile();
    for $i = 0$ to RepeatCount do
5:      /* Compute random number between 0 and 1 */
        r ← GenerateRandomNumber(0,1);
        if r > .5 then
            /* write block of size 2048 bytes five times */
            performDiskIO;
10:     else
            /* CPU activity */
            r ← GenerateRandomNumber (0, RepeatCount/100)
            for all $i = 1$ to $r$ do
                Calculate $\pi$ using Monte Carlo with iteration count equal to RepeatCount
15:         end for
        end if
    end for
    CloseFile (f);
    printTimingInfo;
```

1. *Virtual Machine*: A RHEL 2.6+ kernel based CentOS VM running on one core of a i7-3740QM processor running at 2.7GHz. This VM has 4GB of memory.

2. *Physical Machine*: A machine running CentOS Linux 2.6+ kernel. This machine has a 32-core Xeon(R) CPU E5-2665 running at 2.4 GHz and is organized as 2 NUMA nodes with a total of 132 GB memory. Only one of the 32 cores was used to run the benchmark.

The service demands obtained by running the four types of jobs at the VM and physical machine environments are shown in Tables 5.2 and 5.3, respectively.

Table 5.2: Service Demands for Jobs on the Virtual Machine

Job Id $\rightarrow$	Job1	Job2	Job3	Job4
CPU	53.4	18.7	6.3	26.96
Disk	6.6	4.5	3.4	5.14
RepeatCount	10000	7000	5000	8000

Table 5.3: Service Demands for Jobs on the Physical Machine

Job Id $\rightarrow$	Job1	Job2	Job3	Job4
CPU	60.38	28.5	4.87	11.05
Disk	9.02	8.3	3.24	5.45
RepeatCount	9000	7000	4000	5000

Table 5.4: Epoch data for the physical machine - Micro-benchmark - Scenario1

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1,J2
3	10	15	5	Arr J3	J1,J2,J3
4	15	32.5	17.5	End J3	J1,J2
5	32.5	77.3	44.8	End J2, Arr J4	J1, J4
6	77.3	104.3	27.0	End J4	J1
7	104.3	118.0	13.7	End J1	J1

Table 5.5: Execution times for the physical machine - Micro-benchmark - Scenario 1

Job No.	Measured Execution Time	Predicted Execution Time	% Relative Error
Job 1	116.9 $\pm$ 2.3	112.6	3.7
Job 2	69.1 $\pm$ 0.8	68.4	1.0
Job 3	24.9 $\pm$ 0.07	27.0	-8.4
Job 4	17.2 $\pm$ 0.8	17.5	-1.7

The measured execution times reported in the tables that follow represent averages over 10 runs for the virtual machine configuration and over 15 runs for the physical machine one. The tables also report 95% confidence intervals for these averages.

Two scenarios were run on the physical machine configuration using the micro-benchmark and the four jobs derived from it. The epoch data for the first scenario, which has seven epochs, is shown in Table 5.4. The table also shows the start time and end time of each epoch, their duration, the event that triggered the start of the epoch, and the workload mix in each epoch. The arrival times of jobs J_1-J_4 were predetermined. The other values in the table were determined through measurements obtained by the execution of the four jobs. The values of Table 5.4 represent a single run of Scenario 1 for illustration purposes.

Table 5.5 shows the average measured execution times for the four jobs for scenario 1 on the physical machine and their corresponding 95% confidence intervals. Column 3 shows the predicted execution times computed with the Epochs algorithm. The last column of the table shows the percent relative error, whose absolute value varies between 1.7% and 8.4%.

The epoch data for the second scenario on the physical machine is shown in Table 5.6. One of the differences between this scenario and the previous is that two instances of each of the four job types were used. The second instance is indicated in the table with a "-2" suffix (e.g., $J1$-2). This execution leads to 16 epochs and a larger concurrency level than the previous scenario. For example, there are eight concurrent jobs in execution in epoch 8.

Table 5.6: Epoch data for the physical machine - Micro-benchmark - Scenario 2

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1, J2
3	10	15	5	Arr J3	J1, J2, J3
4	15	20	5	Arr J4	J1, J2, J3, J4
5	20	25	5	Arr J1	J1, J2, J3, J4, J1-2
6	25	30	5	Arr J2	J1, J2, J3, J4, J1-2, J2-2
7	30	35	5	Arr J3	J1, J2, J3, J4, J1-2, J2-2, J3-2
8	35	40	5	Arr J4	J1, J2, J3, J4, J1-2, J2-2, J3-2, J4-2
9	40	40.5	.5	End J3	J1, J2, J4, J1-2, J2-2, J3-2, J4-2
10	40.5	71.8	31.3	End J3-2	J1, J2, J4, J1-2, J2-2, J4-2
11	71.8	91.6	19.8	End J4	J1, J2, J1-2, J2-2, J4-2, J2-2, J4-2
12	91.6	110.8	19.2	End J4-2	J1, J2, J1-2, J2-2
13	110.8	147.3	36.5	End J2	J1, J1-2, J2-2
14	147.3	168.2	20.9	End J2-2	J1, J1-2
15	168.2	209.3	41.1	End J1	J1-2
16	209.3	220.6	11.3	End J1-2	J1-2

Similarly to Table 5.5, Table 5.7 shows experimental and predicted execution times for each of the jobs. As it can be seen, most of the absolute values of the percent relative error are below 3.4%. Only one value has an error of 15%.

Table 5.8 shows epoch data for a virtual machine scenario with jobs from the micro-benchmark. There are seven epochs in this scenario. The measured execution times and execution times predicted by the Epochs algorithm for the data in Table 5.8 are shown in Table 5.9. As the table indicates, the absolute relative error varies from 4 to 11.6%.

The Epochs algorithm was then validated using jobs from real Unix benchmarks. In particular, three jobs were used from the benchmark: Nbench [101], Bonnie++ [99], and Dbench [100]. Nbench is a synthetic computing benchmark program intended to measure a computer's CPU, FPU,

Table 5.7: Execution times for the physical machine - Micro-benchmark - Scenario 2

Job Number	Measured Exec. Time	Predicted Exec. Time	% Relative Error
Job 1	207.2 ± 2.24	204.4	1.4
Job 1-2	198.7 ± 1.03	195.7	1.5
Job 2	142.2 ± 2.60	137.4	3.4
Job 2-2	142.0 ± 2.37	138.3	2.6
Job 3	24.9 ± 0.44	25.5	-2.4
Job 3-2	43.3 ± 0.75	36.8	15
Job 4	71.4 ± 0.43	71.6	-0.3
Job 4-2	68.8 ± 0.39	70.8	-2.9

Table 5.8: Epoch data for the virtual machine environment - Micro-benchmark Scenario

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1,J2
3	10	15	5	Arr J3	J1,J2,J3
4	15	36.9	21.9	End J3	J1,J2
5	36.9	57.2	20.3	End J2, Arr J4	J1, J4
6	57.2	114.8	57.6	End J4	J1
7	114.8	115.5	0.7	End J1	J1

Table 5.9: Execution times for the virtual machine environment - Micro-benchmark Scenario

Job Number	Measured Exec. Time	Predicted Exec. Time	% Relative Error
Job 1	114.8 ± 0.6	110.2	4.0
Job 2	61.8 ± 0.88	57.6	6.6
Job 3	50.6 ± 0.52	47.3	6.5
Job 4	24.8 ± 1.02	21.9	11.6

Table 5.10: Service Demands for UNIX benchmark jobs on the Virtual Machine Environment

Job Names →	Nbench (J1)	Bonnie++ (J2)	Dbench (J3)
CPU	25.0	8.2	5.5
Disk	0.0	9.8	4.5

and memory system speeds, and includes various types of sorts, bit manipulation, compression and encryption algorithms, LU decomposition, floating-point emulation, neural network, and a task allocation algorithm. Bonnie++ is a benchmark suite aimed at performing a number of simple tests of hard drive and file system performance. Dbench is a tool to generate I/O workloads to either a file system or to a networked CIFS or NFS server.

Each of these three benchmark programs were run in isolation a large number of times and the average service demands were computer at the CPU and disk on the virtual machine configuration. The results are shown in Table 5.10, which also indicates that these three jobs will be referred heretofore as J1, J2, and J3.

Table 5.11 shows epoch data for an experiment using the three UNIX benchmark jobs described above. There are 12 epochs in this scenario and two instances of jobs J1, J2, and J3 arriving at difference time instants. During epoch 6, both instances of the three jobs are running concurrently.

Table 5.11: Epoch data for Unix Benchmark Jobs in the Virtual Machine Environment

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1, J2
3	10	15	5	Arr J3	J1, J2, J3
4	15	20	5	Arr J1-2	J1, J2, J3, J1-2
5	20	25	5	Arr J2-2	J1, J2, J3, J1-2, J2-2
6	25	39.97	14.97	Arr J3-2	J1, J2, J3, J1-2, J2-2, J3-2
7	39.97	50.47	10.5	End J3	J1, J2, J1-2, J2-2, J3-2
8	50.47	57.23	6.76	End J2	J1, J1-2, J2-2, J3-2
9	57.23	67.33	10.1	End J3-2	J1, J1-2, J2-2
10	67.33	69.38	2.05	End J2-2	J1, J1-2
11	69.38	79.59	10.21	End J1	J1-2
12	79.59	79.59	0	End J1-2	None

Table 5.12 shows the average execution times and their 95% confidence intervals as well as the

job execution times predicted by the Epochs algorithm. As it can be seen, the absolute relative error varied from 2.7% to 11.3%.

We created 10 jobstreams with 50 job arrivals consisting of the three Unix benchmark programs (NBench, Bonnie++ and DBench). The interarrival times are exponentially distributed with arrival rates ranging from .01/sec to .1/sec. Figure 5.3 shows percent relative error as a function of the arrival rate for the job streams. The relative error percent is calculated as the absolute error based on the makespan of the jobstream predicted by the Epochs algorithm and the response time found by experimentation on a single server. The graph shows that the maximum error is 18%, with the average error staying around 10%.

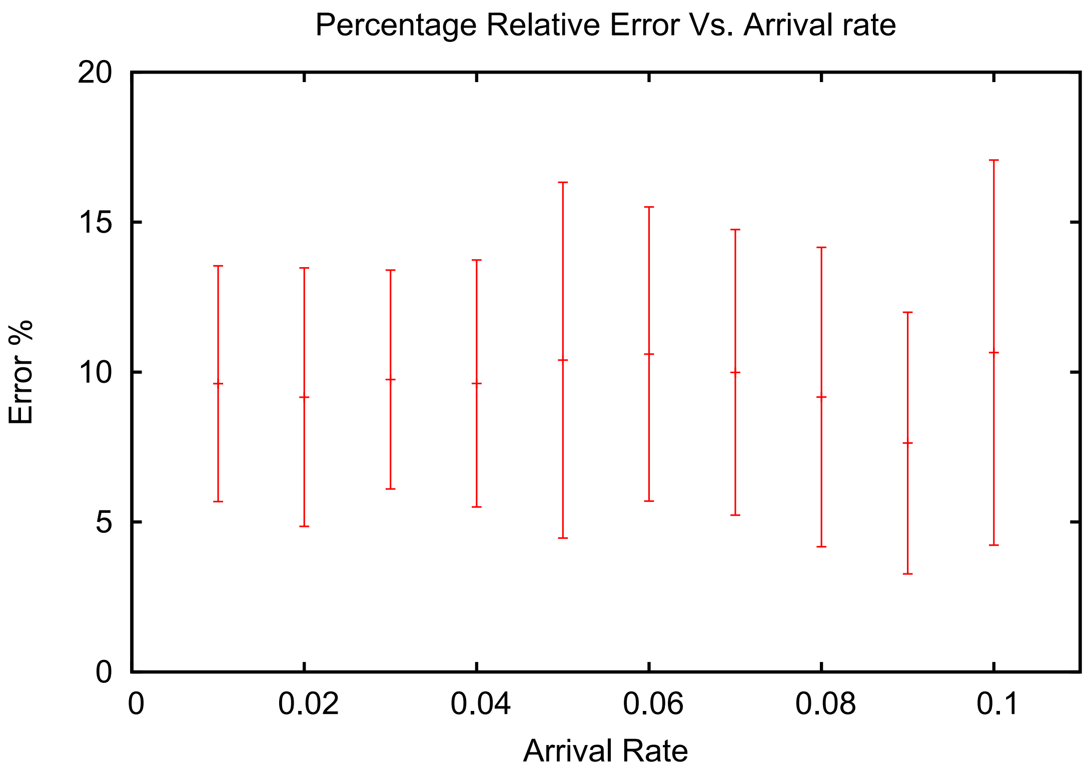


Figure 5.3: Error percentage (between predicted and observed makespan) as a function of arrival rate.

5.5 Applications of the Epochs Algorithm to Scheduling

One of the applications of the Epochs algorithm is on the performance evaluation of server clusters that receive a stream of jobs that are scheduled according to some scheduling policy. Figure 5.4 shows several servers each receiving a sub-stream of the global job stream. The stream of jobs received by each server is determined by a scheduler.

Table 5.12: Execution times for Unix Benchmark Jobs in the Virtual Machine Environment

Job Number	Measured Exec. Time	Predicted Exec. Time	% Relative Error
Job 1	67.6 ± 1.08	69.4	-2.7
Job 1-2	67.8 ± 1.24	64.6	4.7
Job 2	51.3 ± 2.04	45.5	11.3
Job 2-2	53.0 ± 1.43	47.3	10.75
Job 3	27.8 ± 0.83	29.9	-7.5
Job 3-2	30.3 ± 1.2	32.2	-6.2

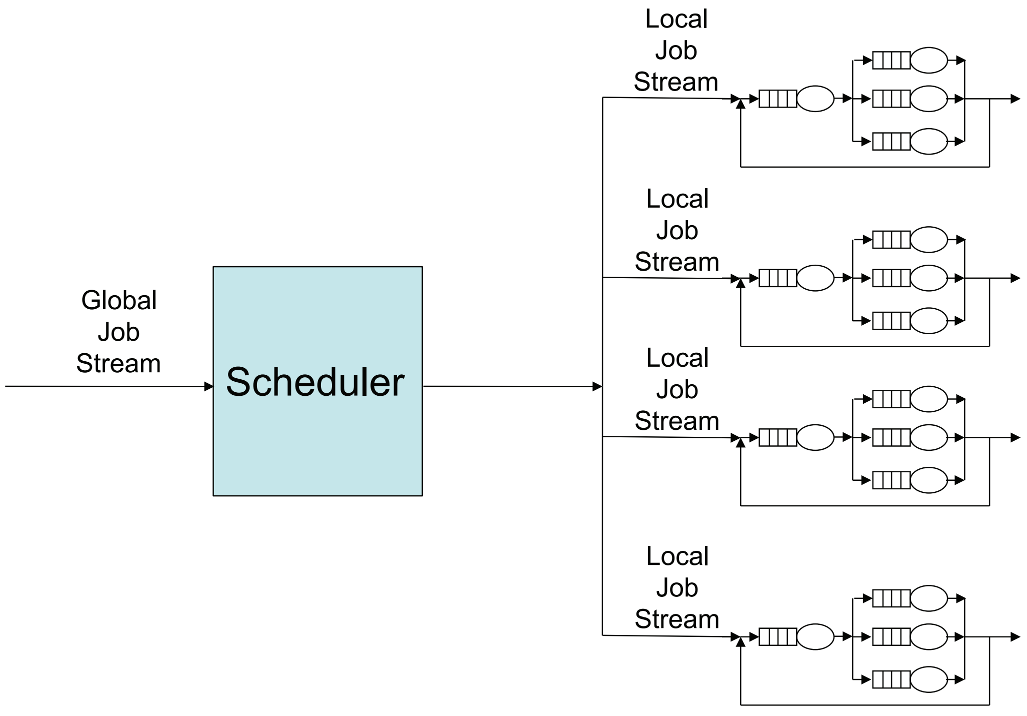


Figure 5.4: Use of the Epochs Algorithm for Assessing Schedulers in Server Clusters.

Modeling the scheduler using analytic models is very difficult in general. There is a vast body of literature on analytic modeling of schedulers for single queues or for server clusters in which servers are modeled as single queues. Harchol-Balter [34] presents a good survey on the topic. However, one can either simulate the scheduler or implement the scheduler in a way that generates the local sub-streams for each server. Then, the Epochs algorithm can be applied to each server (explained in more detail in the next chapter of this dissertation). For example, Fig. 5.5 shows the makespan (i.e., the time to complete all the jobs in a job stream) for 10 different job streams randomly generated from jobs chosen from Nbench, Bonnie++, and Dbench, for an average arrival rate of 0.167 jobs/sec.

The four scheduling policies used in Fig. 5.5 are:

- Round Robin (RR): The scheduler chooses the servers in the cluster in a round robin fashion. This scheduling scheme is oblivious to the utilization of any server resource (either CPU or disk).
- Least Response Time (LRT): This scheduling algorithm finds the machine on which the incoming job is predicted to have the least response time. Since the scheduler has the exact states of all the jobs running on all servers, it can estimate the response time of an incoming job if it were added to any node in the cluster.
- Least Maximum Utilization First (LMUF): The server with the minimum utilization for the resource with the highest utilization is the one that receives an incoming job. The utilization of the resources at each server is calculated as a snapshot at the time the new job arrives to be scheduled.
- Least Maximum Utilization First-Threshold (LMUF-T): Similar to LMUF except that a job is not sent to a server if the utilization of the resource with the highest utilization (i.e., the bottleneck) exceeds a certain threshold. In that case, the job is queued at the scheduler. When a job completes at any machine, the scheduler attempts to send the queued job again to one of the servers. The goal of LMUF-T is to bound the contention at each node. However, jobs will wait at the scheduler. It may be more advantageous to wait a bit at the scheduler and then be assigned to a less loaded machine.

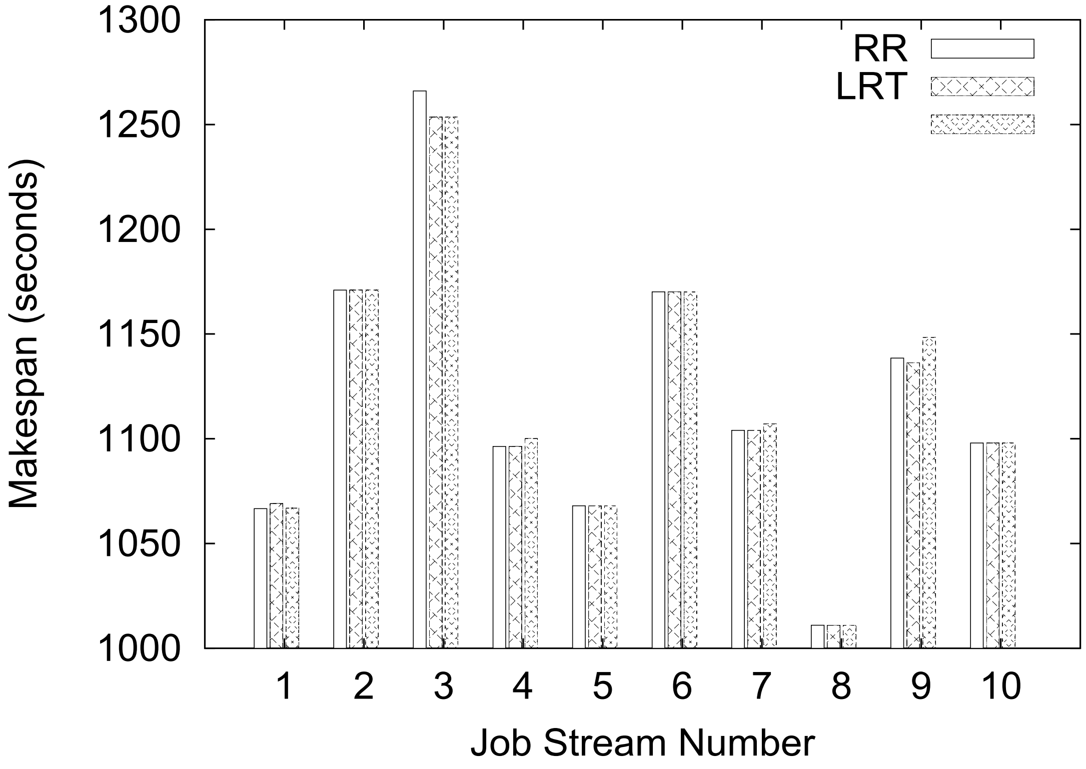


Figure 5.5: Makespan vs. jobs stream for various schedulers.

Each of the four schedulers was implemented taking as input a global job stream that arrives at a server cluster. Then, a scheduler uses its scheduling policy to decide where the arriving jobs should be sent, generating the job streams for each server.

5.6 Computational Complexity

The computational complexity of the Epochs algorithm can be estimated from the computational complexity of the Approximate Mean Value Analysis (AMVA) algorithm (see [55]). Due to the use of the Bard-Schweitzer approximation [64], the computational complexity of AMVA is proportional to $\text{Niter} \times K \times R$ where Niter is the number of iterations needed by AMVA to converge to a given tolerance, K is the number of devices (i.e., processors and I/O devices), and R is the number of job classes. The Epochs algorithm requires that AMVA be run in each epoch. The number of classes in each epoch is equal to the number of jobs in the workload mix of that epoch. Thus, the computational complexity of the Epochs algorithm is proportional to $\text{NE} \times \text{Niter} \times K \times \bar{J}$, where NE is the number of epochs and $\bar{J}$ is the average concurrency level in the workload mix of each epoch.

Let us estimate the computational complexity of the Epochs algorithm for reasonable values of the parameters discussed above. AMVA is known to converge rapidly in most cases. Depending on the value of the tolerance used, it is not uncommon for Niter to be less than 100. The number of devices K is typically small, e.g., of the order of 5. The average concurrency level $\bar{J}$ is also a relatively small number, say no more than 100.

Experiments were run in which AMVA was executed several times for a model with 5 devices, 100 classes with population equal to 1 each (as is the case in the Epochs algorithm), and service demands randomly generated with values between 1 and 100 sec for each device and each class. The average execution time of an AMVA Java implementation on a laptop with an Intel 7 processor was 147 msec with a 95% confidence interval equal to 9.2 msec. The average number of iterations needed to achieve convergence for a tolerance of 5×10^{-4} was 39 with a 95% confidence interval equal to 1.5. Thus, the computational complexity of the Epochs algorithm is proportional to $\text{NE} \times 39 \times 5 \times 100 = 19,500 \times \text{NE}$ for the parameter values discussed above. The average execution time of the Epochs algorithm on the machine used in our experiments would then be $147 \times$ NE msec. The value of NE depends on the length of the trace. Hence, the analysis of a trace that implied in 1,000 epochs would take 147 sec, i.e., a bit over two minutes. Thus, the computational complexity of the Epochs algorithm is well within the reach of modern computers.

5.7 Related Work

Combining different modeling techniques to evaluate the performance of a computer system has been done by many before with the goal of obtaining the benefits of more than one technique. For example, Mehdipour et al. have combined simulation and analytic models for processor design [53]. Norton has introduced the Simalytic technique, which combines simulation methods with analytic models [24]. Menascé has shown how to combine Generalized Stochastic Petri Nets (GSPN) and Queuing Networks (QN) to reduce the size of the state space of GSPNs [54].

The idea of hybrid simulation/analytic models is not new. In fact, an interesting taxonomy on the types of hybrid models is presented in [67]. In that paper the authors present some examples of hybrid models. All examples, except for one, are for single-queue systems. The exception is for a computer system with admission control due to a limited multiprogramming degree. The authors in [67] suggest simulating the computer system for different values of the degree of multiprogramming n, and modeling the entire system as a load dependent single queue in which the service rate is a function of n.

This work differs from previous ones because it takes inputs that are typical in simulation modeling and uses them as inputs to analytic QN models.

5.8 Concluding Remarks

The traditional approach for specifying the workload in analytic queuing network models uses the following methods for specifying the workload: (a) average job arrival rates in the case of open job classes and (b) job population in the case of closed classes [9, 55, 63]. Besides these two types of workload intensity parameters, the service demands for each job class have to be specified. This work presents a novel technique that consists in driving analytic queuing network models with job traces (real or synthetically generated), which specify job arrival instants within the trace and the types of the arriving jobs. Service demands are associated with job types.

The method presented here, called the Epochs algorithm, is based on scanning a job trace and determining finite-duration time intervals called epochs in which a certain workload mix is active. The duration of each epoch is estimated using the Mean Value Analysis equations in each epoch in order to determine when the current epoch terminates due to the completion of a job. The basis of the approach used in the Epochs algorithm lies in the fact that the MVA equations are valid for finite duration intervals if certain operational assumptions such as flow balance, one-step behavior, and homogeneous service times are satisfied [14]. The advantage of the method presented here over traditional methods for specifying the workload in analytic models is that any distribution-independent job trace can be used as input as long as the operational assumptions are met. These assumptions are much more general and easier to verify than the stochastic assumptions used in traditional analytic models.

The job execution time predictions were validated experimentally using a micro-benchmark developed by the authors and with real programs from well-known Unix benchmarks. The results indicated that the relative absolute error stays below 10% in most cases and is at most 15% for the cases examined.

Chapter 6: A Contention-Aware Hybrid Evaluator for Assessing Schedulers for Big Data Applications

6.1 Introduction

The work described in this chapter has been published in [4]. With the algorithm described in the previous chapter, it is now possible to use the Epochs algorithm to start deriving performance related values (throughput, response times) for a slew of jobs hitting a cluster of computers in a simulated as well as in a real execution environment. The main discussion point of this chapter is a novel method which is inherently aware of contention on computing resources. This method, called TDAM (Trace Driven Analytic Model), relies on the implementation of the scheduler under evaluation and on analytic closed queuing network (QN) models to assess resource contention experienced on the cluster nodes.

Many enterprises today run Big Data applications on a cluster of heterogeneous machines. Apache Hadoop [90] and associated software stacks are examples of such software platforms and products. As has been discussed before, Internet organizations like Amazon, Yahoo, LinkedIn, and Facebook run thousands of Hadoop jobs on a routine basis on clusters comprising of thousands of server nodes. These jobs have varied completion time requirements since some are ad-hoc quick query jobs, some are medium size data mining jobs, and some are very large (in terms of resource requirements and completion times) analytical processing jobs.

Hadoop developers have made available several different schedulers over the years to schedule MapReduce jobs to suit a particular organizational needs. These schedulers need extensive testing and verification for correctness. Most job schedulers in the Hadoop ecosystem have complex XML configuration files to set up queues and quotas. But, more importantly, testing and validation is needed to check if the schedulers are appropriate for the intended workload mix.

The efficacy of a job scheduler can be assessed in many different ways. These are (1) *Experimentation:* Select a representative mix of real jobs, setup a real cluster, run the jobs using a given scheduler and measure the completion times. This method is very onerous mainly because obtaining a suitable free cluster for experimentation is often very difficult. (2) *Simulation:* Simulate a representative mix of real jobs running through a simulated scheduler and using simulated servers. This method is complex because not only the scheduler but the processing and I/O resources of the servers have to be simulated in software. (3) *Analytic modeling:* Develop analytic models of the scheduler, servers and their queues using the proper arrival distributions. This is not trivial because modeling the behavior of even moderately complex scheduling disciplines and their interaction with the server models for heterogeneous workloads may not be mathematically tractable.

Many research papers [75, 77, 80] have been published during the last few years proposing a variety of methods to predict the completion time of MapReduce jobs. However, they do not consider the contention experienced by the nodes as a result of more than one task executing concurrently on a compute node. Especially with the notion of number of slots on a TaskTracker not being fixed anymore [71], it is extremely important that resource contention be estimated correctly. The work described in this section attempts to bridge the gap in today's MapReduce research which considers the completion time of a task to be independent of the number of tasks executing at the same node.

The main contribution of this section is to introduce a mathematically sound method to assess schedulers for server clusters, such as the ones running Hadoop MapReduce platforms, one of the prime movers for Big Data applications. This method, called TDAM (Trace Driven Analytic Model), is a hybrid method that relies on the implementation of the scheduler under evaluation and on analytic closed queuing network (QN) models to assess resource contention at the cluster nodes. The implemented scheduler under test takes as input a synthetic or a real trace of jobs of various types (in terms of resource usage) and schedules (not necessarily as dictated by the trace) them on "servers", which are modeled by analytical QNs. This allows the user to examine how different scheduling methods affect the completion time of a stream of jobs. By using an implementation of the scheduler on a real or synthetic job trace we avoid the complexity of modeling the behavior of scheduling policies. The analytic QN models capture the congestion of CPU and I/O resources at the various servers. These QN models are solved using an operational analysis formulation [12] of mean value analysis [55] equations for finite time intervals.

We applied the TDAM method to assess a few common cluster scheduling policies under different workload types. We implemented the TDAM method inside Hadoop's Mumak [97], a job-trace simulator distributed with Hadoop. Implementing TDAM in Mumak makes it aware of the contention experienced by tasks while executing concurrently with other tasks at server nodes. That way, the contention-aware Mumak (CA-Mumak) is able to predict task and job completion times much more accurately instead of relying on Mumak's fixed task completion times provided in the trace it uses as input. The choice of Mumak as a trace simulator for Hadoop job traces is incidental and the TDAM methodology can be applied to any number of Hadoop job simulators available today. Even though we have discussed the applicability of the TDAM methodology with respect to Hadoop and MapReduce, the theory and the models that are used in TDAM are also applicable to any scenario where job schedulers are used in a cluster of computers.

The rest of this chapter is organized as follows. Section 6.2 discusses the motivation and need for assessing different job schedulers. The following section presents the TDAM method. Section 6.4 uses TDAM to assess four different scheduling policies on different types of workloads (Hadoop MapReduce job trace and also non-Hadoop synthetic job trace). Section 6.6 uses CA-Mumak to assess the impact of actual Hadoop schedulers (e.g., FIFO, Capacity, and Fair) on a real Hadoop job under various scenarios in which cluster size and job characteristics vary. This section also discusses some experimental result from implementing the Epoch's algorithm right inside the Hadoop FIFO scheduler making the scheduler contention aware. Section 6.8 discusses related work and lastly section 5.8 presents some concluding remarks and future work.

6.2 The Need for Robust Job Scheduler Assessment

Predicting job completion time needs congestion information on computer nodes. All the resources of a modern machines behave like queues. Processes that need to use these resources have to wait their turn. As a result of this phenomenon, when multiple processes run concurrently on a node, they experience an increase in their execution times compared to what it would be if they were running in isolation. Assume that a CPU bound process takes 10 secs. to complete when run by itself on a uniprocessor machine. Now, if five of these processes were started at the same time on the same machine, each will take 50 secs to complete and due to fair processor scheduling on most OSs today, and thus all the processes will finish around the same time after 50 seconds. This makes sense intuitively and indeed is trivial when we consider only 1 CPU (hence 1 queue). In case of a more realistic scenario when multiple resources like many CPUs, disks, and memory banks are present, all producing queuing effects on modern machines, the prediction of performance metrics such as response time, throughout, and resource utilization can be made from the service demands by using queuing-theoretic analytic models [12, 55].

A big challenge in today's Hadoop and similar cluster platforms is to manage the allocation of resources to different jobs so that a service level objective (SLO) can be advertised and met. This requires (1) a quick, efficient, and realistic way to assess the schedulers before they go into production; and (2) an efficient method to estimate resource congestion at the servers. Our solution to this challenge is the TDAM method, which allows any type of job scheduler to be efficiently evaluated on any job trace. It is conceivable that even for a single enterprise, depending on the time of day and workload mix, one scheduling scheme outperforms another. Dynamically choosing schedulers based on the workload mix and overall resource utilization (which requires collecting and analyzing data from every node and other cluster characteristics like network bandwidth and topology) is the subject of self-configuring autonomic job scheduling [62]. We do not discuss autonomic aspects of dynamic scheduling in any detail in this chapter.

Originally, Hadoop was designed to run large batch jobs infrequently. The out-of-the-box FIFO scheduler packaged with Hadoop was sufficient for that purpose. There was really no need to worry about resource utilization to complete jobs in a timely manner. However, other schedulers were developed as the number of jobs increased many fold and organizations started to use Hadoop not only for batch analytics but also for small ad-hoc queries. The *Capacity Scheduler* was developed for more efficient cluster sharing. The *Fair Scheduler* was introduced to maintain fairness between different cluster users. Despite the presence of many schedulers, there is really no way to test with any accuracy the performance of the schedulers for a set of jobs with varying characteristics and possibly a compute cluster with heterogeneous servers without actually running the jobs on a real cluster. Many other Hadoop schedulers have been developed since then [39]. These schedulers did not provide any method to assess their effects on a job trace, but more importantly there was no way to know how they would behave under moderate or heavy workloads without actually running real jobs.

One of the primary goals of our research is to show how to apply the TDAM method to help enterprises assess the effectiveness of scheduling disciplines on the performance of Big Data applications. Towards that end, we have augmented Hadoop's Mumak simulator with the TDAM method and made Mumak contention aware.

6.3 The TDAM Framework

In this section we refer to jobs as single task jobs. In the context of MapReduce, the proper terminology would be tasks. Figure 6.1 depicts the basis for the TDAM framework. The scheduler, which in TDAM is an actual implementation of the scheduler, takes a global stream of jobs (J1, J2, J3, and J4 in the figure) as input and schedules them into the various servers of the cluster. This creates a local job stream for each server. In TDAM, servers are not actually implemented but they are represented by analytic queuing network (QN) models (see Server QN in the figure for each server). A QN is a collection of queues, where each queue consists of a device (e.g., CPU, disk), represented by a circle, and a waiting line, represented by a rectangle in front of the circle. The output of a queue can feed the input of another queue creating a network of queues. In order to solve a QN we need two types of parameters: workload intensity (e.g., the number of concurrent jobs in execution at the server) and service demands for each job type at each device. For example, the CPU service demand for a given job type is the total CPU time for jobs of that type. The solution of a QN determines the average execution time of jobs of each type. Figure 6.1 shows that the top server has a local job stream in which only jobs J1 and J3 appear and the bottom server has a local job stream with jobs J2 and J4 only. The scheduler decides which jobs are assigned to each server.

An important aspect of TDAM is that servers are modeled analytically using a workload provided by the implemented scheduler. The second important aspect of TDAM is that the contention

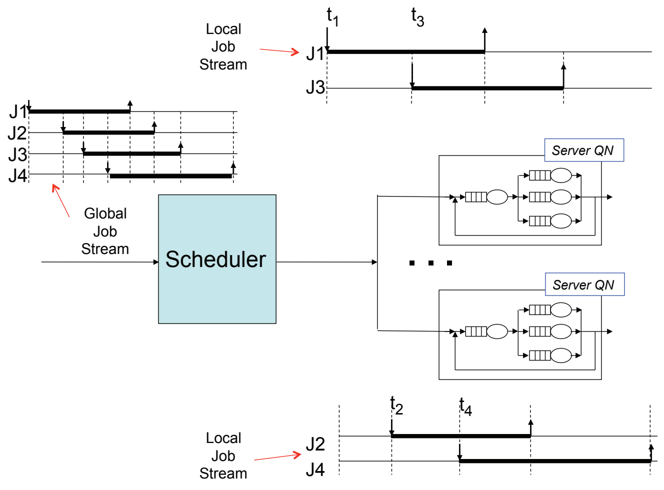


Figure 6.1: The TDAM Framework.

for resources at each server is modeled by the QNs that represent each server. Therefore, TDAM is contention aware.

When we look at the local job stream for each server, we see that the concurrency level varies as jobs arrive and finish. For example, the timeline for the top server in Fig. 6.1 shows that time is divided into three different intervals, which we call *epochs*. During the first one, only job J1 is executing. During the second epoch, jobs J1 and J3 are executing concurrently. In the last epoch, that starts after job J1 completes, only job J3 is executing. In order to model this type of situation, we need to extend the MVA algorithm in order to estimate how much of each job's service demand was executed at each epoch. This extension is explained in detail in chapter 5 of this dissertation.

The scheduler implementation maintains global cluster information regarding which jobs are executing at each server and how much work each job has already performed. This allows the scheduler to implement a variety of contention-aware scheduling disciplines as we show in the rest of this section.

Another important advantage of TDAM is that it can be used to assess any scheduler on a cluster of any size. The reason is that the scheduler is implemented and not modeled or simulated and the servers are modeled analytically using extremely fast solution methods. TDAM can be used in any cluster, including those running MapReduce jobs. For our purposes, we implemented TDAM in Mumak and made this popular Hadoop hob trace simulator contention aware.

6.4 Scheduler Assessment Using TDAM

This section discusses the results of assessing four different scheduling policies using TDAM. The workload consists of streams of tasks derived from well-known Unix benchmark programs. The choice of the scheduling schemes is intended to showcase the variety of insights that one can glean regarding the performance parameters of the executing node, like throughput, response time, utilization and queue lengths, which are essential to predict completion times of individual tasks. The four non-preemptive scheduling policies discussed here are:

- Round Robin (RR): Chooses the servers in the cluster in a round robin fashion. This scheduling scheme is oblivious to the utilization of any server resource (either CPU or disk).
- Least Response Time (LRT): Selects the server on which the incoming task is predicted to have the least response time. Since the scheduler has the exact states of all the tasks running on all the servers, it can predict the response time of the incoming task if it were added to any node in the cluster.
- Least Maximum Utilization First (LMUF): Assigns the incoming task to the server with the minimum utilization for the bottleneck resource (i.e., the resource with the highest utilization). The utilization of a resource at each server is calculated as a snapshot at the time the new task arrives to be scheduled.
- Least Maximum Utilization First-Threshold (LMUF-T): Similar to LMUF except that a task is not sent to a server if the utilization of its bottleneck resource exceeds a certain threshold. In that case, the job is queued at the scheduler. When a job completes at any node, the scheduler attempts to send the queued job again to one of the servers. The goal of LMUF-T is to bound the contention at each node by having jobs wait at the scheduler. It may be more advantageous to wait a bit at the scheduler and then be assigned to a less loaded machine.

For the workload used to compare the schedulers, we created job streams by randomly selecting jobs from one of the three benchmarks: Bonnie++ [99], Nbench [101], and Dbench [100]. Inter-arrival times were assumed to be exponentially distributed, even though this assumption is not required by TDAM. Any arbitrary arrival process that satisfies the homogeneous arrival assumption can be used [12]. The job stream files created, along with the scheduling scheme and the number of servers in the cluster are the main input parameters to the experimental runs.

We consider both single-task jobs and multi-task jobs. In a multi-task job, the various tasks of a job run on the same or different nodes of a cluster and the job is deemed to have completed only when all its tasks have completed. MapReduce jobs are examples of multi-task jobs [90].

First, we consider the effect of different scheduling schemes on the makespan of various single-task jobs. Then, we consider the impact of the CPU utilization threshold in the LMUF-T scheduling policy. Lastly, in the subsection we discuss the results of running multi-task jobs on a heterogeneous cluster.

6.4.1 Effect of Scheduling Policy on the Makespan

This subsection considers how the scheduling policy affects the makespan, i.e., the time needed to execute all jobs in a job stream. Table 6.1 shows the characteristics of the jobs used in the evaluation carried out in this section. For single-task jobs, we made very minor modifications to three benchmark programs (Bonnie++ [99], Nbench [101] and Dbench [100]) and measured their CPU and disk service demands (see Table 6.1). Changing the input parameters to these benchmark programs allowed us to obtain two sets of service demand values (e.g., for Bonnie++, the two sets of values for CPU and disk demands are [8.2 sec, 9.8sec] and [16.4 sec, 19.6 sec]). The job inter-arrival times are exponentially distributed with averages of 3 sec and 6 sec. Thus, as shown in Table 6.1, we obtained four different workloads by combining two service demand sets and two average inter-arrival time values.

Table 6.1: (CPU, Disk) service demands (in sec) for benchmark jobs Bonnie++, Nbench, and Dbench, and two values of the workload intensity

Workload	1	2	3	4
Job $\downarrow$	Inter-arrival time:6 sec		Inter-arrival time:3 sec	
Bonnie++	(8.2, 9.8)	(16.4, 19.6)	(8.2, 9.8)	(16.4, 19.6)
Nbench	(25, 0)	(50, 0)	(25, 0)	(50, 0)
Dbench	(5.5, 4.5)	(11, 9)	(5.5, 4.5)	(11, 9)

For each workload in Table 6.1, we created 10 job streams by randomly selecting, with equal probability, the type of job at each arrival instant. Figure 6.2 depicts the makespan for all 10 streams for workload 1 (other workload charts depict similar results but are omitted to save space) and for each of the four scheduling disciplines. The utilization threshold used in LMUF-T is 70% for all workloads. This value is used here for illustrative purposes only. Note that the y-axis does not start at zero so that the differences between the schedulers are easier to visualize. The following conclusions can be drawn from the data. First, LMUF-T provides the worst (i.e., the highest) makespan in all cases. This is a consequence of LMFU-T not sending jobs to a server when the utilization of its bottleneck device exceeds the threshold T. While LMUF-T is inferior than the other policies for a 70% threshold, there may be good reasons to use this policy: (1) the energy consumption of a server increases with its utilization, and (2) running servers at high utilizations may reduce their reliability and their lifetimes. Second, LRT and LMUF provide similar makespans at the 95% confidence level for most job streams and workloads. The reason is that there is a strong correlation between the response time at a node and the utilization of its bottleneck device. Finally, RR is worse than LRT and LMUF for jobs with higher service demands and/or higher workload intensity values. This is an expected result because RR is oblivious to the workload or load on the server.

We performed one-factor ANOVA [41] at the 95% confidence level for the data shown for workload 1 (and other workloads not shown). LMUF-T was shown to be significantly different at the 95% confidence level than the three other scheduling disciplines. We also applied the Tukey-Kramer [41] procedure to the three other scheduling disciplines and found statistically significant differences among them in several of the tested workload.

6.4.2 Threshold Effect on LMUF-T

We now discuss the effect of the bottleneck threshold value used by LMUF-T. In that case, the total execution time of a job consists of two components: wait time at the scheduler and time spent at a node, which includes time spent using resources and time waiting to use node resources.

Figures 6.3(a) and 6.3(b) show the waiting time at the scheduler, server time, and total time

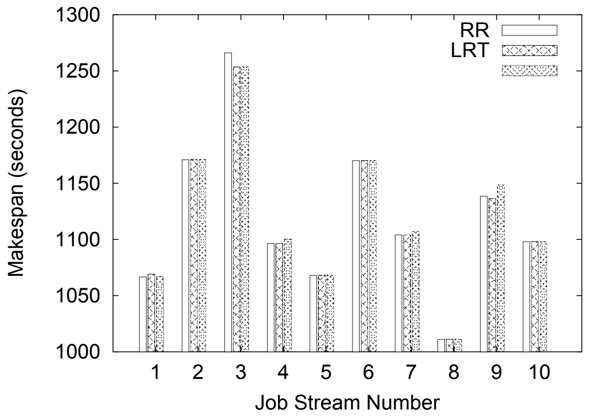


Figure 6.2: Makespan vs. job stream for single-task jobs in workload 1.

for workloads 1 and 3 versus the CPU utilization threshold for Bonnie++ and Nbench, respectively. These workloads contain a mix of the three types of jobs in Table 6.1. However, each graph only shows average values for a specific job within the multi-job workload. Concentrating on Figure 6.3(a), as the CPU utilization threshold approaches 100%, the wait time (the line at the bottom of the chart) goes to zero as expected because no jobs will be queued at the scheduler. However, as seen in the figures (not all the graphs are shown to save space), the server time increases due to increased congestion at the servers. For each type of program (meaning different service demands), the range of threshold values that provides the best total execution time is not the same. For Bonnie++, this range is [0.6, 0.9]; for Nbench it is [0, 0.4] and for Dbench (not shown) the lowest execution time is reached for a CPU utilization threshold equal to 100%. Figure 6.3(b) uses the same service demands as in Figure 6.3(a) but with an average arrival rate twice as large. This corresponds to workload 3 in Table 6.1. There is a marked difference in behavior between the graphs of Figure 6.3. As we can see from the figures, because the arrival rate has doubled, server congestion increases significantly when the effect of scheduler throttling is reduced by using a high utilization threshold. For example, the best utilization threshold range is [0.1, 0.4] for Nbench (as well for Bonnie++ and Dbench).

Figure 6.4(a) shows the CPU utilization for a typical server in the cluster using RR scheduling for workload 2. Figure 6.4(b) shows similar data for LMUF-T scheduling with a 70% threshold for the same workload. In the RR case, the CPU utilization quickly reaches 100% and stays there. However, in the LMUF-T case, the average CPU utilization stays at a lower level and never stays

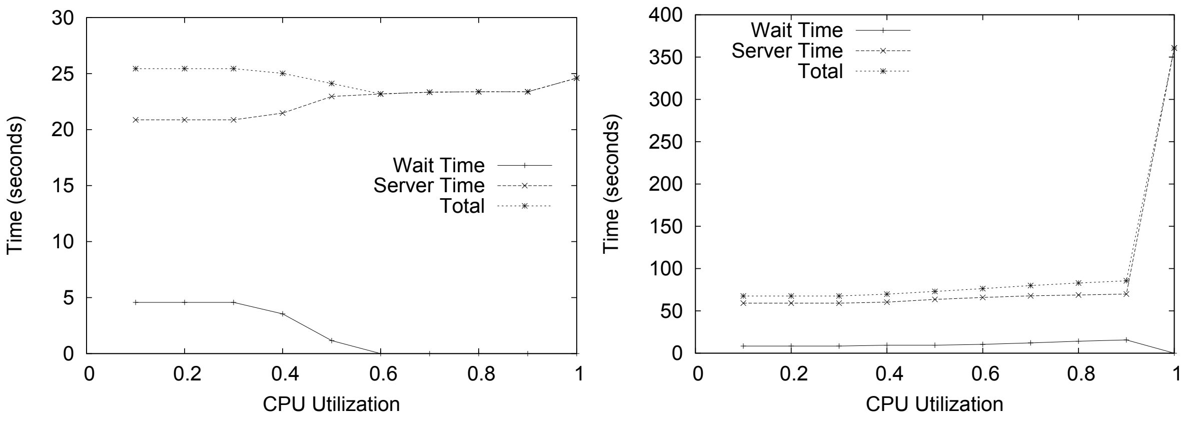


Figure 6.3: Average execution time, wait time, and server time vs. CPU utilization threshold. Left: (a) Bonnie++ and workload 1. Right: (b): Nbench and workload 3.

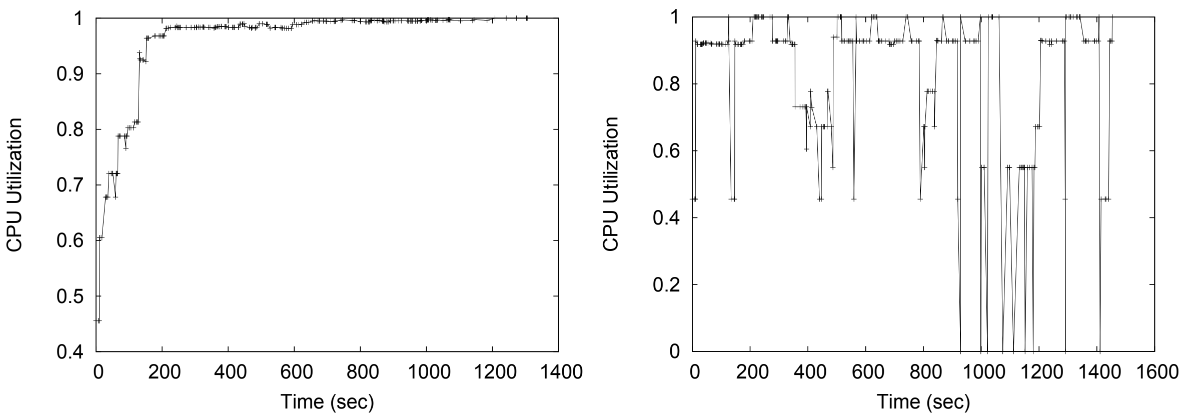


Figure 6.4: CPU utilization vs. time for workload 2. Left: (a) RR. Right: (b) LMUF-T.

fixed at 100% (an undesirable situation) as in Figure 6.4(a). This is because of the 70% threshold for CPU utilization used by LMUF-T. These sorts of insights into the running of jobs with various characteristics can be possible only if TDAM based tools are available to conduct ”what-if” scenarios with scheduling schemes in an accurate fashion.

6.4.3 Assessing Synthetic MapReduce Job Traces

This subsection discusses the results from assessing a job trace consisting a set of MapReduce jobs on a cluster with heterogeneous servers.

A MapReduce job is considered finished only when all its tasks finish. Once a task is scheduled to run on a node, it can not be switched to another node (unless it fails and is retried by the framework). The workloads used are described in Table 6.2 and they differ in terms of their CPU and disk service demands. We assume that all the jobs arrive as indicated in the job-trace file. Based on the number of tasks per job and the number of jobs in Table 6.2, there are 50 small job tasks, 50 medium job tasks, and 50 large job tasks. We assume a cluster with 12 nodes, six of which are half as fast as the other six. The CPU and disk service demands in Table 6.2 correspond to the faster machines. The corresponding values for the slow machines are twice the values depicted in the table.

Table 6.2: Multi-task job characteristics.

Job Type	No. Tasks	No. Jobs	CPU (sec)	Disk (sec)
Small	5	10	5	1
Medium	25	2	10	5
Large	25	2	30	15

Experiments were performed using LMUF-T scheduling using three utilization thresholds: 0.0, 0.7 and 1.0. Table 6.3 shows the overall makespan for four different scheduling scenarios: (1) Any job can be scheduled on any machine, (2) Large jobs are only scheduled on slow machines and the other jobs are scheduled on any machine, (3) Small and medium jobs are scheduled on the slow machines and large jobs on the fast machines only, and (4) Small jobs are scheduled to the slow machines and medium and large jobs are scheduled into any machine.

The following observations can be made from the data shown in Table 6.3 (and additional data collected from the experiment and not shown here). First, for a given CPU utilization threshold value, the best makespan values are obtained either when any job can be scheduled on any machine (row 1) or when fast nodes are used exclusively by large jobs (row 3). Second, the data shows a clear impact of the threshold on server congestion. For example, for a CPU threshold of 0.0 (similar to the Hadoop approach of granting exclusive access to CPU cores to each task) there is very small server contention and large waiting times at the scheduler queue. For example, large jobs spend between 73% and 79% of their total time at the scheduler queue in any of the four scenarios. When the utilization threshold is 70% , there is a slight increase in server time due to added server congestion and a decrease in waiting time at the scheduler queue. As a consequence, the total times are lower than when the utilization threshold is zero. Third, the case in which there is no queuing at the scheduler favors small jobs over the other threshold values for all cases except for case 3. This is expected because in case 3 small and medium jobs can only use the slower machines and there is significant contention at these nodes because there is no admission control at the scheduler. On the other hand, large jobs have a much higher total time under the 100% threshold. This is due to the very high contention at the server nodes when compared with the other threshold values.

Even though we have shown results for three different utilization threshold values and for four scenarios for allocating tasks to the different types of nodes, the approach presented in this body of work can easily be applied to a wide variety of "what-if" scenarios. The reason is that the scheduler implementation is unchanged and the servers are modeled using closed queuing networks. The integration between the scheduler implementation and the closed QN models is done through the Epochs algorithm discussed in the previous chapter. A significant advantage of the TDAM methodology is that it allows for any scheduling discipline to be easily and accurately assessed for any size and type of cluster given that task/job contention for server resources is modeled analytically.

Table 6.3: Makespan (In Secs) for small, medium, large MapReduce jobs for different CPU utilization thresholds

	Overall Makespan ($U_{\text{cpu}} \leq 0.0$)	Overall Makespan ($U_{\text{cpu}} \leq 0.7$)	Overall Makespan ($U_{\text{cpu}} \leq 1.0$)
Any job to any machine	391	344	373
Large jobs to slow machines, other jobs to any machine	675	675	612
Small/Medium jobs to slow machines, large jobs to fast machines	405	343	273
Small jobs to slow machines, medium/large jobs to any machine	413	347	462

6.5 Design of CA-Mumak

Figure 6.5 shows a component-level diagram of CA-Mumak. The workload produced by the *Workload Generator* is a stream of jobs consisting of the list of tasks that compose the jobs and a few other interesting metadata about the tasks e.g., start and finish times. For example, MapReduce jobs are composed of map tasks and reduce tasks. Firstly, the job trace data is transformed into entities suitable for consumption by the *JobTracker Manager* and the *TaskTracker Manager* components of Mumak. Similar to what a standard Hadoop MapReduce framework does, Mumak uses JobTracker and TaskTracker classes to schedule tasks from a MapReduce job trace. However, unlike Hadoop where these components run as daemon processes, Mumak brings in the functionality of these pieces in its own process space. Mumak also tracks the finishing times of the tasks and eventually the entire job and records timings. The problem with the stock version of Mumak is that it has no awareness of the elongation of task execution times because it has no notion of task service demands (i.e., the total service time per task at each node resource) or the amount of resource contention that tasks undergo when multiple tasks execute on the same node. This is where the TDAM components come in. They bring in the knowledge of service demands (e.g., CPU and I/O) for the tasks of a particular job. Armed with this information, CA-Mumak can now not only schedule tasks in a variety of ways based on scheduler configuration, but is also able to utilize the analytical model engine that is part of TDAM to accurately predict the makespan of a set of jobs.

Continuing with the component diagram, the *Server View Manager* uses the well-known analytic closed queuing network (QN) algorithms [12, 55] and the server information to maintain current load state across the entire cluster as tasks come and leave worker nodes. The closed QN algorithm estimates the execution time of jobs in a job/task stream. Time is divided into intervals of finite duration called *epochs*. The first epoch starts when the first task arrives and the end of an epoch is characterized by either the arrival of a new task or the completion of a task. The QN algorithms predict the start time of the next epoch on any cluster node by being aware of the residual times of each task as tasks start and finish executing on the nodes. The *Residual Task Calculator* component computes the residual service time of each task in execution using the Epochs algorithm described in the previous chapter. The *Analytic Performance Model* block in the diagram implements the

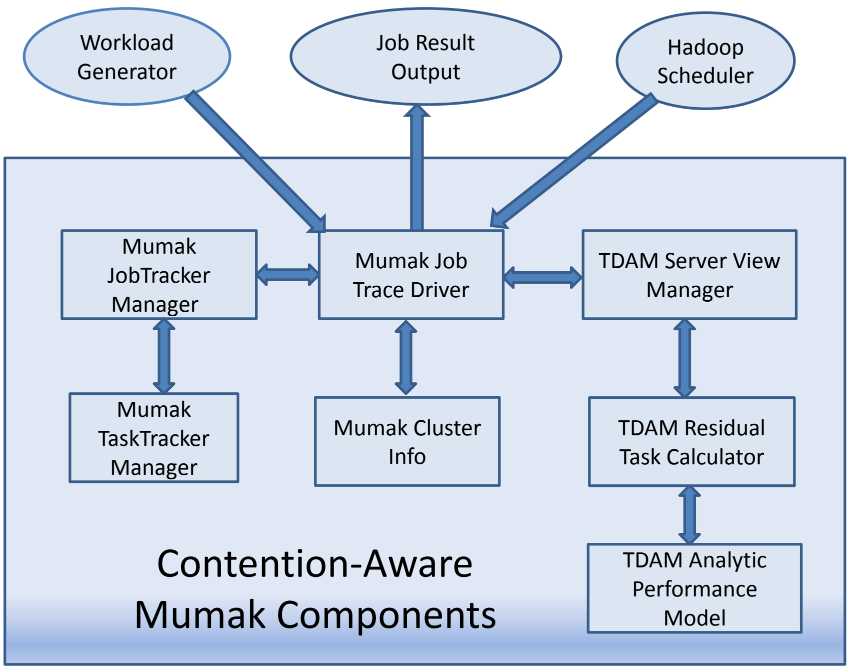


Figure 6.5: Components of TDAM-based CA-Mumak.

Approximate Mean Value Analysis (AMVA) algorithm [55]. In essence, the Epochs algorithm along with the AMVA calculation engine allows us to predict the completion time of individual tasks when a stream of jobs arrives to be executed on a cluster node.

As the simulation starts, Mumak reads in a Hadoop job-trace and the tasks for these jobs are executed virtually (since these jobs are not really executing on servers) and are scheduled to run on different nodes of a cluster utilizing the built-in JobTracker and TaskTracker inside of the simulator. Mumak also takes a cluster topology as input which consists of a certain number of TaskTrackers (execution nodes). Mumak then uses the configured Hadoop scheduler (e.g., FIFO) to schedule tasks on a TaskTracker with the understanding that as soon as the expected time of the task is complete, it would let the other components (e.g., JobTracker) know about the completion and thus be ready to receive the next task if required. The expected completion time of a scheduled task as determined from the trace by Mumak, is saved in the events to go-off after the timer expires in a typical event-calendar-driven simulator fashion. The idea behind CA-Mumak is to let Mumak schedule jobs from

Hadoop job-traces as it would normally do but augment the calculation of the completion time more accurately when the execution environment (e.g., the mix of co-executing tasks) changes. The third set of inputs to CA-Mumak is a set of service demands (CPU and I/O) for various tasks of the jobs. We measured these values by running the tasks in isolation and profiling resource demands.

Without the TDAM method, Mumak will simply consider the completion time noted in the trace for each particular task. But, the trace-specified completion time is only valid for a task if the same set of other tasks were running on the worker node when the task in question was in execution. In other words, if according to the trace file a task T1 took t minutes to execute on a machine, then this execution time t is valid only for the mix of tasks memorialized in the trace. If the task mix were to be different (either more or less time consuming) then the completion time t for task T1 is no longer valid. The TDAM method rectifies this situation because it keeps a complete view of all the tasks executing on each node in a dynamic fashion. Therefore, it is able to predict as tasks start and finish on each worker node, the execution time of a task with a high degree of accuracy utilizing queuing theory principles. It is therefore possible to estimate the makespan of jobs on different size clusters and also perform "what-if" scenarios using different schedulers and their configurations.

6.6 MapReduce Job Scheduler Assessment with CA-Mumak

This section discusses the results of the job scheduler assessment using stock and CA-Mumak and compares the results with job execution time values obtained by running MapReduce jobs on a real cluster using three different Hadoop job schedulers under MapReduce 1.0. We considered map-task-only jobs (we encounter these kind of MapReduce jobs in Big Data handling production environments performing bulk ingestion operations) for the results shown here. We assumed that all tasks are data local which is not the case in actuality for a large enough cluster. We have also not considered network contention. Additionally, the service demands will naturally change for a task when the input file to a job is smaller or larger than the size on which the service demands were calculated. We have therefore made sure during the experiments on a real cluster that the number of map tasks created and the data locality of the tasks are maintained as far as possible.

We first discuss the results of simulation runs using the three main Hadoop schedulers (FIFO, Fair, and Capacity) with stock and CA-Mumak. Tables 6.4-6.6 show the results of several job simulation runs on a 1- and 2-node cluster with 20 map slots on each TaskTracker. There are two types of jobs (column 1) and their service demands are: 100 sec of CPU and no I/O demand for type 1 jobs and 70 sec of CPU and 30 sec of I/O for type 2 jobs. Column 2 shows the number of jobs of each type. It is assumed that the TaskTrackers run on single core computers. Note that for stock Mumak the total makespan for the job with 10 tasks (row 1 of each table) is just 131 sec, which is clearly not possible since 10 map tasks are running at the same time and contending for the core. CA-Mumak shows a total time of around 1000 sec which makes sense since 10 tasks each with 100 sec of CPU service demand sharing a 1-core machine will finish in about 1000 (100×10) seconds. The other rows of each table show makespans for various other combinations of cluster size, job size, and number of jobs. It can be seen that in each case stock Mumak results show very optimistic completion times.

Table 6.4: Makespan using Hadoop's FIFO scheduler

Job Type	No. Jobs	Tasks per Job	No. Nodes	Mumak Time (sec)	CA-Mumak Time (sec)
1	1	10	1	131	1004
1	2	10,10	1	131, 161	1991,2003
1	2	10,10	2	116, 131	995, 1004
2	2,2	10,10	5	120, 126, 132, 138	795, 810, 816, 816

Table 6.5: Makespan using Hadoop's Fair scheduler

Job Type	No. Jobs	Tasks per Job	No. Nodes	Mumak Time (sec)	CA-Mumak Time (sec)
1	1	10	1	102	1002
1	2	10,10	1	102, 600	1989, 2001
1	2	10,10	2	102, 216	1002, 1155
2	2,2	10,10	5	102, 207, 351, 522	711, 849, 1002, 1062

We now discuss experiments based on a production level map-only MapReduce job that ran on a 7-node 4-core per node cluster. This job consists of 200 map tasks, is not very memory intensive, and is used for ingesting data from various sources (e.g., twitter historical feed, flight information)

Table 6.6: Makespan using Hadoop's Capacity scheduler

Job Type	No. Jobs	Tasks per Job	No. Nodes	Mumak Time (sec)	CA-Mumak Time (sec)
1	1	10	1	131	1004
1	2	10,10	1	131, 161	1991, 2003
1	2	10,10	2	116, 131	995, 1004
2	2,2	10,10	5	107,113, 119,125	782,797, 803,803

into a specialized store for visualization and analysis. The job-trace file included the start and finish times of each map task for the job. We measured the task CPU and I/O service demands for the job by running a few map tasks in isolation (easily achieved by making the job run on a single TaskTracker node having only 1 map slot). The CPU and disk service demands for the map tasks were found to be about 98 sec and 25 sec, respectively.

Table 6.7 shows the job's running time results obtained by running this job and by predicting these values through Mumak and CA-Mumak. We varied the number of TaskTrackers in the experiments (i.e., number of nodes in the cluster) from 3 to 5 to 7 and also varied the number of map slots available on each node from 2 to 4 to 8. The % relative error ([experiment - predicted]/experiment) is clearly better for CA-Mumak. For stock Mumak the % error range is between -26% and 51%, whereas for CA-Mumak the range is between 2% and 15%.

We experimented with two other similar jobs. One with 670 and another with 100 map tasks. The results are statistically similar to the results presented here. It is to be noted that during our experiments, when we varied the number of TaskTrackers, some of the tasks were not data-local anymore. We have not considered these variances and also have not considered network contention, which could be very high for a host of different MapReduce job types

Table 6.7: Contention Aware Mumak vs. Stock Mumak results with Hadoop FIFO scheduler

No. TaskTrackers	Map slots Per TaskTracker	Mumak Makespan (sec)	CA-Mumak Makespan (sec)	Actual Job Makespan (sec)	%Error on Mumak	%Error on CA-Mumak
3	2	3791	4592	5200±93	27.1	11.7
3	4	1934	2807	3100±46	37.6	9.5
3	8	1202	2216	2454±36	51.0	9.7
5	2	3107	2702	2868±55	-8.3	5.8
5	4	1844	1652	1844±25	0.0	10.4
5	8	1028	1232	1454±29	29.3	15.3
7	2	2816	2027	2235±45	-26.0	9.3
7	4	1613	1332	1429±39	-12.9	6.8
7	8	1109	986	1010±32	-9.8	2.4

6.7 Scheduler Assessment with Contention Aware (CA-FIFO) Hadoop Scheduler

6.7.1 CA-FIFO Design and Implementation

This section discusses our work of embedding contention awareness directly into the FIFO MapReduce scheduler. The software pattern and paradigm used to enhance FIFO scheduler to be contention aware is valid for enhancing many other schedulers in Hadoop MapReduce 1.0 as well as MapReduce 2.0 (YARN).

In chapter 3 of this dissertation, we presented the pseudo code for the Hadoop FIFO scheduler in its original form. In algorithm 1 presented below, we present the FIFO scheduler enhanced with the TDAM method (a combination of analytical performance models and the Epochs algorithm discussed in the previous chapter). The main idea is that the modified scheduler is now intimately aware of the resource utilization and the state of the load on the executing nodes across the entire cluster (e.g., TaskTracker on which the scheduler has been asked to schedule a task from the jobs waiting on the queue), and is therefore able to make decisions as to whether it should at all schedule a task or possibly schedule a task which would not be normally scheduled. In Chapter 3 Algorithm 1, we presented the Hadoop's stock FIFO scheduler pseudocode. We present an enhanced version of the original Hadoop FIFO scheduler which is contention aware in Algorithm 9 called CA-FIFO.

The scheduler object in Hadoop is actually owned by the JobTracker. When a TaskTracker has an empty slot, it asks the JobTracker with its normal heartbeat mechanism for a task (either map or reduce - we concentrate on the map tasks here). JobTracker in turns calls the *assign()* method in the scheduler which returns one or more tasks from the queue of jobs waiting to get serviced. Lines 1 to 10 in the algorithm is where CA-FIFO gathers data from the objects about the status of the TaskTracker, the total Maps that the TaskTracker can support, and the total number of running map tasks. At line 17, the scheduler gets the current utilization threshold of the CPU for this TaskTracker. Since the scheduler is contention-aware, it keeps a running tally of the tasks starting and finishing on a TaskTracker and thus, using the Epochs algorithm, it can calculate the residual response time of the remaining running tasks and the utilization of any device. Line 19 checks to see if the current CPU threshold is larger than the configured high-water mark for the CPU threshold. If that is the case, the main scheduling code is bypassed (lines 19-49). This means that the scheduler decided to postpone the assignment of a task until the CPU utilization falls below the specified threshold. If the current utilization is below the threshold, the scheduler attempts to schedule tasks from node local list of available tasks (line 30) and thereafter from remote node task lists (line 39) all the while trying to not schedule tasks that will take the utilization of the CPU above a specified value (lines 31 and 41). Even though in our discussion, we are concentrating on CPU utilization specifically, it is trivial to extend the discussion to any other resources that make up TaskTracker nodes in a Hadoop cluster. Once the task assignment is decided, the scheduler updates the model and calculates the residual time for all the existing tasks using the Epochs algorithm (lines 50-55).

As we have already seen, in a Hadoop cluster, the compute node resources are abstracted as Map and Reduce slots, which are the primary computing units. They are in most cases statically configured before the cluster is started. There is an implicit understanding that Map slots will be used for Map tasks and Reduce slots for Reduce tasks. Also, if a particular compute node is configured with, say, 8 Map slots, then in all cases, irrespective of the characteristic of a task executing on those slots, there will be 8 tasks executing, no more. There are, however, job distribution orders in which vastly different kinds of jobs and tasks are scheduled for execution. As an example, let us assume that there are tasks that are completely I/O and network bound, as is often seen with map task only ingest jobs. It is possible that by scheduling tasks on all available Map slots, the throughput and response time will both suffer compared to if some tasks were held back to bring the overall load on the compute nodes to a more acceptable level. It is however difficult to decide what an acceptable load is for a machine; the TDAM framework with analytic performance model provides the theoretical underpinning for that decision making. In our implementation of CA-MUMAK (Algorithm 9), we have not considered a varying number of map slots explicitly, but due to the fact that the scheduler provides admission control for the tasks based on the device utilization threshold, this effect is implicit. Under normal circumstances, a TaskTracker running on a X core machine is configured with X map slots. However, while using CA-FIFO scheduler, the TaskTrackers will be oversubscribed on the map slots (e.g., 2X or 3X). This means that the number of executing mappers can exceed the number of cores on the machine. The contention aware CA-FIFO scheduler will make sure that the load on the machine (e.g., with respect to CPU utilization) does not exceed a user specified value under any circumstances.

6.7.2 CA-FIFO Scheduler Results

We ran experiments on the new contention aware Hadoop FIFO (CA-FIFO) scheduler with the help of CA-Mumak as well as in a real Hadoop environment consisting of a 2- and a 5-node cluster. First, we discuss the simulation results with job streams with varying parameters (utilization threshold, number of TaskTrackers, number of jobs, number of tasks per job, number of mapper slots per TaskTracker) and present the jobstream makespan results. We fix some of the input parameters and vary a few others and discuss the implication of the results. Tables 6.8 and 6.9 show the makespan for a job stream with 30 tasks running on a 4-core processor (two TaskTrackers in Table 6.8 and five TaskTrackers in Table 6.9). The TaskTrackers would normally have 4 map slots (since these are 4-core machines) and would have a makespan of 530 and 248 seconds respectively. However, we note that, by scheduling more tasks at the same time (6, 8 or 10), we bring down the makespan considerably in many cases. We also note the difference in makespan when the scheduler performs admission control based on the utilization of a resource specified by the user. For example, a user can specify that tasks should only be scheduled when the CPU utilization is under 70% and the disk utilization under 75%. The job chosen for these experiments had a CPU service demand of 100 seconds (zero disk demand) and a total map task count of 30.

Table 6.8: CA-FIFO on CA-MUMAK on 2-node cluster - Makespan (in sec.) as a function of the number of map slots and utilization threshold (4-core processors per machine)

CPU Utilization	4 Map Slots	6 Map Slots	8 Map Slots	10 Map Slots
0.5	608	608	608	608
0.6	608	608	608	608
0.7	533	533	533	533
0.8	539	461	461	461
0.9	530	467	485	485
1.0	530	467	419	422

Table 6.9: CA-FIFO on CA-MUMAK on 5-node cluster - Makespan (in sec.) as a function of the number of map slots and utilization threshold (4 core processors per machine)

CPU Utilization	4 Map Slots	6 Map Slots	8 Map Slots	10 Map Slots
0.5	248	248	608	608
0.6	248	248	248	248
0.7	248	248	248	248
0.8	248	254	254	254
0.9	248	182	182	182
1.0	248	182	182	182

Next we see a couple of cases when we actually run the same MapReduce job on a 2- and 5-TaskTracker cluster. Each node has a 4-core processor and, therefore, normally each TaskTracker will have 4 Map slots. The CA-FIFO scheduler has a knob with which we can set the LMUF-T utilization threshold. We see in Tables 6.10 and 6.11 that by scheduling more than 4 tasks on each node (signified by columns S6, S8 and S10 for 6, 8 and 10 Map slots respectively), we can get a better makespan for the job (526 seconds vs. 644 seconds). It is also instructive to note that we can perform admission control on the scheduler by setting the utilization threshold of a device to a chosen value (e.g., 0.5 to 1.0 for CPU). When the value of the utilization threshold is set to a value lower than 1.0 (e.g., 100% utilization), the scheduler will not schedule a task even if it has been asked to assign a task. Instead, it waits for the next heart-beat from the JobTracker.

Table 6.10: CA-FIFO on 2-node cluster - Makespan (in sec.) as a function of the number of map slots and utilization threshold (4-core processors per machine)

CPU Utilization	4 Map Slots	6 Map Slots	8 Map Slots	10 Map Slots
0.5	713	712	712	712
0.6	712	712	712	712
0.7	637	637	637	637
0.8	643	575	565	565
0.9	644	571	589	589
1.0	644	523	523	526

Table 6.11: CA-FIFO on 5-node cluster - Makespan (in sec.) as a function of the number of map slots and utilization threshold (4 core processors per machine)

CPU Utilization	4 Map Slots	6 Map Slots	8 Map Slots	10 Map Slots
0.5	308	308	708	708
0.6	308	308	298	298
0.7	308	308	298	298
0.8	308	304	294	274
0.9	308	222	272	192
1.0	308	222	272	192

6.8 Related Work

During the last half decade or so, performance analysis and modeling of MapReduce jobs has received significant attention and several different approaches have been presented [36, 72]. A representative group from HP Labs and collaborating researchers have published numerous papers on how to improve the resource allocation of MapReduce programs and have presented proposals on how to model the map, reduce, and shuffle phases of these jobs [75, 77, 82, 83].

There is a significant body of work on scheduling for single queues. Harchol-Balter brings an excellent survey of analytic scheduling results for M/G/1 queues in Part VII of her recent book [34]. In [34], Harchol-Balter also looks at the problem of immediate dispatching of arriving tasks to a server farm. She considers that each server in the server farm is modeled as a single queue, i.e., the CPU and disk resources of the server are not individually modeled as we do here. Also, the work in [34] does not consider the possibility of queuing at the scheduler, as is done in LMUF-T. Several papers on job scheduling for parallel environments appear in [30].

In [36], the authors discuss a query system to answer cluster sizing problems using a combination of job profiles and estimations. The authors in [72] built a cost function based on the complexity of the map and reduce tasks. The profile of these tasks is calculated on a testbed environment. We also measure job characteristics (i.e., service demands) on a testbed environment. However, we do not require that the testbed be sized as the intended production site. In fact, a single node is sufficient to measure service demands. The ARIA paper [75] provides a solid foundation on how to analyze the different phases of MapReduce jobs. The authors first create job profiles from which they can ascertain the optimum allocation of map and reduce slots. Consequently, they create an SLO scheduler that incorporates the aforementioned model. However, that work does not consider resource contention due to multiple tasks running on the same node, as done here. In [3], the authors discuss a tool called Tresa that helps system administrators predict execution times when consolidating workloads in a data center. The tool automates workload characterization and uses MVA to predict execution times. There is no mention to the effect of scheduling policies on job execution times.

However, none of the above referenced papers and no other studies, to the best of our knowledge, consider the effects of resource contention when predicting the effect of scheduling policies on job completion times. In [80], the authors deal with speculative executions of tasks so that failing tasks do not degrade the running time of a job. The authors create a new scheduler called LATE (Longest Approximate Time to End) that is based on heuristics. In [38], the authors created a novel scheduler called "Maestro" that prevents excessive speculative executions of map tasks by scheduling the tasks in two waves which lead to a higher data-locality of the tasks.

Again, these efforts do not take into account resource contention at the node level. The identification of straggler jobs, at the heart of the work in [80], would benefit from knowing if a task is falling behind because the executing node is failing or if the job is having to contend with other high demand jobs. An analytical model could accurately predict the delay a task may experience due to resource contention thus improving the LATE scheduler heuristic. The authors in [17] created a MapReduce-based GIS workflow engine with its own scheduler called the MRGIS scheduler. This scheduler has a rather specialized way of scheduling tasks on Hadoop nodes and would greatly benefit by incorporating a performance model to track the congestion on worker nodes before tasks are actually scheduled.

Algorithm 9 Hadoop Contention Aware FIFO Scheduler Pseudo Code

```
Input: taskTracker
Output: scheduledTasks
/* Initilaize the model and epoch algo objects */
taskTrackerName = taskTracker.getTrackerName();
/* Initialize taskRedualTimeCalculator for taskTracker if needed */
**if** residualTimeCalculatorMap.get(taskTrackerName) is $\emptyset$ **then**
    trtc $\leftarrow$ createTaskRedualTimeCalculator();
    residualTimeCalculatorMap.put(taskTrackerName, trtc);
**end if**
/* Fetch cluster totalMapCapacity, totalRunningMaps etc. */
taskTrackerStatus $\leftarrow$ taskTracker.getStatus();
trackerMapCapacity $\leftarrow$ taskTrackerStatus.getMaxMapSlots();
trackerRunningMaps $\leftarrow$ taskTrackerStatus.getRunningMapCounts();
jobQueue $\leftarrow$ getJobQueue();
availableMapSlots $\leftarrow$ (trackerMapCapacity - trackerRunningMaps);
scheduledTasks $\leftarrow$ $\emptyset$
utilThreshold $\leftarrow$ trtc.maxUtilizedResourceValue();
/* the THRESHOLD_UTIL is configurable */
**if** utilThreshold $\neq$ THRESHOLD_UTIL **then**
    scheduledMaps:
    **for** $i = 0$ to availableMapSlots **do**
        /* Go thru each job in queue */
        **for all** job in jobQueue **do**
            **if** job.getStatus().getRunState() != JobStatus.RUNNING **then**
                continue;
            **end if**
        **end for**
        /* Try to schedule a node-local or rack-local Map task */
        task $\leftarrow$ job.obtainNewLocalMapTask();
        utilThreshold $\leftarrow$ trtc.maxUtilizedResourceValue(task.demands);
        **if** (utilThreshold $<$ THRESHOLD_UTIL) AND (task != null) **then**
            scheduledTasks.add(task);
            /* Break out after a ceiling is reached */
            **if** exceededMapPadding **then**
                break scheduledMaps;
            **end if**
        **end if**
        /* Try to schedule a non node-local or rack-local Map task */
        task $\leftarrow$ job.obtainNewNonLocalMapTask();
        utilThreshold $\leftarrow$ trtc.maxUtilizedResourceValue(task.demands);
        **if** (utilThreshold $<$ THRESHOLD_UTIL) AND (task != null) **then**
            scheduledTasks.add(task);
            /* Break out after a ceiling is reached */
            **if** exceededMapPadding **then**
                break scheduledMaps;
            **end if**
        **end if**
    **end for**
**end if**
**for all** task in scheduleTasks **do**
    /* update the model */
    taskResidualDemand $\leftarrow$ new TaskResidualDemand(clock, cpuDemand, diskDemand);
    taskResidualDemandList $\leftarrow$ new TaskResidualDemandList(taskResidualDemand);
    residualTaskCollection $\leftarrow$ new ResidualTaskCollection(taskName, taskResidualDemandList)
    trtc.addTaskToQueue(residualTaskCollection);
**end for**
```

Chapter 7: Concluding Remarks and Future Work

7.1 Concluding Remarks

In the chapter titled ”Modeling of Hadoop ’map-only’ MapReduce jobs,” we have discussed in detail the map phase of a MapReduce job. We then provided a model for predicting the completion time of the map tasks based on their service demands on CPU and disk using Mean Value Analysis (MVA).

In the chapter titled ”Predicting the Effect of Memory Contention in Multi-core Computers Using Analytic Performance Models,” we demonstrated that the effect of memory contention in multi-core machines may be a significant portion of the execution time of an application, especially for server-class machines with a large number of cores. This chapter presented a two-level approximate single-class queuing network analytic model to predict the execution time of applications running on multi-core machines. One level captures contention for memory and the other incorporates that contention into an application-level model, which takes into account contention for cores. We used hardware counters provided by most modern Intel processor chips to obtain the parameters for the memory contention model. To that end, we focused on back-end-stall cycles, which we contend provides a good approximation for cycles spent due to memory access in the CPU load/store side of the execution of micro operations. We have included as part of our experiments a home-grown micro-benchmark written in C. This has given us the freedom to vary the memory allocation amount, whether to access the memory or to perform disk I/O. Linux’s heavy use of buffer cache, in which the OS fetches complete file data if system memory is available, required us to flush the cache buffer before every run. We have not investigated the effect of this eager fetch of file data on the performance model. We further verified our results with well known LINUX benchmarks, HBench, UnixBench and SPECCPU2006 written in C to further validate our model. As part of the memory contention model work, we also presented a detailed procedure to capture memory contention effect on NUMA based systems. Finally, we discussed the multi-class extension to our single-class memory contention model.

In the chapter titled ”Epochs - An algorithm for calculating execution times for finite time interval,” we presented a new algorithm that estimates the execution times of jobs in a job stream. This algorithm enabled us to create and experiment with the TDAM method (chapter 6 titled “A Contention-Aware Hybrid Evaluator for Assessing Schedulers for Big Data Applications”), used to assess job traces of varied workload characteristics. The Epochs algorithm and the performance models at the heart of this algorithm are the foundation on which we were then able to enhance Hadoop Mumak to a contention aware Hadoop job trace simulator (CA-MUMAK). The same groundwork allowed for the Hadoop FIFO scheduler to be enhanced to a contention aware scheduler (CA-FIFO).

Job schedulers play a very important role in many large enterprise IT infrastructures. Many distributed applications run on multi-node clusters of heterogeneous servers. It is important to test the efficacy of a particular scheduling scheme in an extensive manner before it is put in a production environment. It is generally not feasible to perform live runs with real jobs and large compute clusters to test scheduling scenarios. The approach presented in this dissertation explores the ability to experiment with complex schedulers, which are actually implemented to process a global job trace, with a server cluster that is modeled as a collection of closed queuing networks, one per server. Therefore, no actual cluster is needed to evaluate a given scheduler algorithm. The glue between the scheduler implementation and the server analytic models is the Epochs algorithm, which was validated experimentally using real jobs. Since most clusters today are heterogeneous in nature due to the fact that machines are added as the need grows, it is not uncommon to find a cluster with 8, 16, and 24 core machines with either 32, 128 or 256 GB of RAM. The approach presented in this dissertation allows for the easy analysis of scheduling disciplines on Big Data jobs running on heterogeneous clusters.

It is an accepted fact that massively interconnected systems have become extremely difficult to manage, configure and fix even with expert level help. The dream of pervasive computing may not come true with millions of lines of poetentially suspect quality code running on thousands of systems interconnected on the World Wide Web. We are not even beginning to discuss subversive and malicious activities. In 2001, IBM coined the term Autonomic Computing [45] - computing systems that manage themselves. To provide fruition to this vision of self-* systems (a shortened acronym for Autonomic Computing since these systems are known to self-protect, self-configure, self-optimize and self-heal), it will take the collaboration of many disciplines of research. There has been a lot of research in this area in recent years [10, 11]. In the heart of an autonomic system are the Monitor, Analyze, Plan and Execute functions. Connected to all these functional blocks is another block called Knowledge. Altogether, this is called the MAPE-K loop where an autonomic manager will have feedback from the systems it is part of. With the help of this intelligent control loop (MAPE-K controllers can be either local or global), it will be able to manage the systems in an automatic way. The work in this dissertation started by creating performance models based on Closed QN network equations for Hadoop MapReduce applications, went on to make the models robust by considering the memory contentions on the task nodes followed by the creation of the Epochs algorithm, and culminated with the use of these artifacts to improve a Hadoop job simulator (MUMAK) and a Hadoop scheduler (FIFO) by making them contention aware. We hope that the components created by this dissertation can become part of the technologies used to create an autonomic manager for Hadoop based systems by dynamically varying the scheduling process (choose from a variety of schedulers) as a consequence of the variation in the workload.

7.2 Future Work

Extending the TDAM approach to model other phases of MapReduce jobs like shuffle, sort, and reduce is part of future work. Further validation of the memory contention model with additional CPU hardware counter events, to obtain more accurate values for memory-related contention, will be an useful research area. In the dissertation we have performed our experiments with a few types of Intel chips. It will be beneficial to validate our model on other Intel, AMD, and Power chips.

We assume throughout this dissertation that the service demands of the jobs are known apriori, which is a reasonable assumption in captive data centers like Yahoo and Facebook. But for a hosted

Hadoop cluster available for general MapReduce in a publicly available cloud (e.g., Amazon EC2), the service demand of a job is obviously not available beforehand. However, for sufficiently large jobs, the scheduler can still learn the characteristics of the jobs by watching the run time of a few tasks that constitute job. This is an area ripe for future investigation.

Multi-threaded programs (e.g., Java by default is multi-threaded) have to deal with cache coherency problems, in which the time spent keeping caches synchronized cannot be ignored. Considering the effects of several levels of caches found in modern computers and capturing their access times, will be an useful extension to our memory contention model. All the experiments described in this dissertation were conducted on small clusters (2 to 5 nodes) with server quality hardware. Running experiments with medium size clusters (30-60 machines) should be considered. This will further enhance the validation process. We have tried contention-awareness enhancement only on the Hadoop FIFO scheduler. This work should be extended to make Fair scheduler and Capacity scheduler contention aware. Finally, as has been mentioned, all the experimental work in this dissertation was done on Hadoop version 0.22, also known as MapReduce 1.0. An important way in which this work can be taken forward by attempting to validate the software models and modules created for this dissertation on MapReduce 2.0 (YARN).

Bibliography

[1] Ahmad, Faraz, et al. “MapReduce with communication overlap (MaRCO).” Journal of Parallel and Distributed Computing 73.5 (2013): 608-620.

[2] Akyildiz, Ian F., and Gunter Bolch. “Mean value analysis approximation for multiple server queueing networks.” Performance Evaluation 8.2 (1988): 77-91.

[3] Ansaloni, Danilo, et al. “Find your best match: predicting performance of consolidated workloads.” Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ACM, 2012.

[4] Bardhan, Shouvik, and Daniel A. Menascé. “A contention aware hybrid evaluator for schedulers of big data applications in computer clusters.” Big Data (Big Data), 2014 IEEE International Conference on. IEEE, 2014.

[5] Bardhan, Shouvik, and Daniel A. Menascé, “Predicting the Effect of Memory Contention in Multi-core Computers Using Analytic Performance Models.” Computers, IEEE Transactions on 2014 (Volume:PP , Issue: 99)

[6] Bardhan, Shouvik, and Daniel A. Menascé, “The Anatomy of MapReduce Jobs, Scheduling, and Performance Challenges.” 2013 Int’l Conf. Computer Measurement Group, La Jolla, CA, November 5-8, 2013.

[7] Bardhan, Shouvik, and Daniel A. Menascé. “Queuing network models to predict the completion time of the map phase of mapreduce jobs.” Proceedings of the Computer Measurement Group International Conference. 2012.

[8] Bard, Yonathan. “Some extensions to multiclass queueing network analysis.” Proceedings of the Third International Symposium on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems. North-Holland Publishing Co., 1979.

[9] Baskett, Forest, et al. “Open, closed, and mixed networks of queues with different classes of customers.” Journal of the ACM (JACM) 22.2 (1975): 248-260.

[10] Bennani, Mohamed N., and Daniel A. Menascé. “Assessing the robustness of self-managing computer systems under highly variable workloads.” Autonomic Computing, 2004. Proceedings. International Conference on. IEEE, 2004.

[11] Bennani, Mohamed N., and Daniel A. Menascé. “Resource allocation for autonomic data centers using analytic performance models.” Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on. IEEE, 2005.

[12] Buzen, Jeffrey P., and Peter J. Denning. “Measuring and calculating queue length distributions.” (1979).

[13] Buzen, Jeffrey P. “Fundamental operational laws of computer system performance.” Acta Informatica 7.2 (1976): 167-182.

[14] Buzen, Jeffrey P., and Peter J. Denning. “Operational Treatment of Queue Distribution and Mean Value Analysis.” (1979).

[15] Chang, Fay, et al. “Bigtable: A distributed storage system for structured data.” ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

[16] Chen, Lydia Y., et al. “Achieving application-centric performance targets via consolidation on multicores: myth or reality?.” Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012.

[17] Chen, Qichang, Liqiang Wang, and Zongbo Shang. “MRGIS: A MapReduce-Enabled high performance workflow system for GIS.” eScience, 2008. eScience’08. IEEE Fourth International Conference on. IEEE, 2008.

[18] Chen, Yanpei, Sara Alspaugh, and Randy H. Katz. Design insights for MapReduce from diverse production workloads. No. UCB/EECS-2012-17. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 2012.

[19] Chiang, Mee-Chow, and Gurindar S. Sohi. “Experience with mean value analysis model for evaluating shared bus, throughput-oriented multiprocessors.” ACM SIGMETRICS Performance Evaluation Review 19.1 (1991): 90-100.

[20] Daniel A. Menascé, and Shouvik Bardhan. “Epochs: Trace-Driven Analytical Modeling of Job Execution Times.” Technical Report GMUCS-TR-2014-01, Computer Science Department, George Mason University, 2014.

[21] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: a flexible data processing tool.” Communications of the ACM 53.1 (2010): 72-77.

[22] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” Communications of the ACM 51.1 (2008): 107-113.

[23] Denning, Peter J., and Jeffrey P. Buzen. “The operational analysis of queueing network models.” ACM Computing Surveys (CSUR) 10.3 (1978): 225-261.

[24] Dumke, Reiner, ed. “Performance engineering: state of the art and current trends.” Vol. 2047. Springer Science & Business Media, 2001.

[25] Du, Xing, et al. “Architectural effects of symmetric multiprocessors on TPC-C commercial workload.” Journal of Parallel and Distributed Computing 61.5 (2001): 609-640.

[26] Eager, Derek L., Daniel J. Sorin, and Mary K. Vernon. "Analytic modeling of burstiness and synchronization using approximate mva". Technical Report 1391, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1998.

[27] Eranian, Stephane. "What can performance counters do for memory subsystem analysis?" Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08). ACM, 2008.

[28] Erlang, Agner Krarup. Sandsynlighedsregning og telefonsamtaler. 1909. Later in French: Calcul des probabilite conversations telephoniques. 1925

[29] Fedorova, Alexandra, Sergey Blagodurov, and Sergey Zhuravlev. "Managing contention for shared resources on multicore processors." Communications of the ACM 53.2 (2010): 49-57.

[30] Frachtenberg, Eitan, and Uwe Schwiegelshohn. "Job scheduling strategies for parallel processing." Springer Berlin/Heidelberg, 2009.

[31] Ganapathi, Archana, et al. "Predicting multiple metrics for queries: Better decisions enabled by machine learning." Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on. IEEE, 2009.

[32] Ganapathi, Archana, et al. "Statistics-driven workload modeling for the cloud." Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on. IEEE, 2010.

[33] Guide, Part. "Intel 64 and IA-32 Architectures Software Developer.s Manual." (2010).

[34] Harchol-Balter, Mor. "Performance Modeling and Design of Computer Systems: Queueing Theory in Action." Cambridge University Press, 2013.

[35] Hennessy, John L., and David A. Patterson. "Computer architecture: a quantitative approach". Elsevier, 2012.

[36] Herodotou, Herodotos, Fei Dong, and Shivnath Babu. "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics." Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[37] Herodotou, Herodotos. "Hadoop performance models." arXiv preprint arXiv:1106.0940 (2011).

[38] Ibrahim, Shadi, et al. "Maestro: Replica-aware map scheduling for mapreduce." Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on. IEEE, 2012.

[39] Isard, Michael, et al. "Quincy: fair scheduling for distributed computing clusters." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.

[40] Isci, Canturk, et al. "Improving server utilization using fast virtual machine migration." IBM Journal of Research and Development 55.6 (2011): 4-1.

[41] Jain, Raj. "The art of computer systems performance analysis." John Wiley & Sons, 2008.

[42] Kavulya, Soila, et al. “An analysis of traces from a production mapreduce cluster.” Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE, 2010.

[43] Kayi, Abdullah, et al. “Experimental evaluation of emerging multi-core architectures.” Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007.

[44] Kelly, Terence, et al. “Operational analysis of parallel servers.” Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on. IEEE, 2008.

[45] Kephart, Jeffrey O., and David M. Chess. “The vision of autonomic computing.” Computer 36.1 (2003): 41-50.

[46] Kleinrock, Leonard. Computer applications, volume 2, queueing systems. Wiley, 1976.

[47] Krevat, Elie, et al. “Applying performance models to understand data-intensive computing efficiency.” No. CMU-PDL-10-108. CARNEGIE-MELLON UNIV PITTSBURGH PA PARALLEL DATA LABORATORY, 2010.

[48] Lee, Gunho, Byung-Gon Chun, and Randy H. Katz. “Heterogeneity-aware resource allocation and scheduling in the cloud.” Proceedings of HotCloud (2011): 1-5.

[49] Levesque, John, et al. “Understanding and mitigating multicore performance issues on the AMD Opteron architecture.” Lawrence Berkeley National Laboratory (2007).

[50] Levinthal, David. “Performance analysis guide for intel core i7 processor and intel xeon 5500 processors.” Intel Performance Analysis Guide (2009).

[51] Little, John DC. “A proof for the queuing formula: $L = \lambda W$.” Operations research 9.3 (1961): 383-387.

[52] Liu, Huan. “Cutting mapreduce cost with spot market.” 3rd USENIX Workshop on Hot Topics in Cloud Computing. 2011.

[53] Mehdipour, Farhad, et al. “A combined analytical and simulation-based model for performance evaluation of a reconfigurable instruction set processor.” Proceedings of the 2009 Asia and South Pacific Design Automation Conference. IEEE Press, 2009.

[54] Menascé, Daniel A. “A methodology for combinining GSPNs and QNs.” Computer Measurement Group Conference. 2011.

[55] Menascé, Daniel A., et al. “Performance by design: computer capacity planning by example.” Prentice Hall Professional, 2004.

[56] Menascé, Daniel A., et al. “Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures.” Journal of Parallel and Distributed Computing 28.1 (1995): 1-18.

[57] Menascé, Daniel A., Virgilio A.F. Almeida, and Larry W. Dowdy. “Capacity Planning for Web Services: metrics, models, and methods.” Upper Saddle River: Prentice Hall PTR, 2002

[58] Menascé, Daniel A., Stella C.S. Porto, and Satish K. Tripathi. “Processor assignment in heterogeneous parallel architectures.” Parallel Processing Symposium, 1992. Sixth International. IEEE, 1992.

[59] Ozturk, Ozcan. “Multicore education through simulation.” Microelectronic Systems Education, 2009. MSE’09. IEEE International Conference on. IEEE, 2009.

[60] Peng, Lu, et al. “Memory performance and scalability of Intel’s and AMD’s dual-core processors: a case study.” Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE International. IEEE, 2007.

[61] Polo, Jorda et al. “Resource-aware adaptive scheduling for mapreduce clusters.” Middleware 2011. Springer Berlin Heidelberg, 2011. 187-207.

[62] Ramirez, Alain E., Barbara Morales, and Tariq M. King. “A self-testing autonomic job scheduler.” Proceedings of the 46th Annual Southeast Regional Conference on XX. ACM, 2008.

[63] Reiser, Martin, and Stephen S. Lavenberg. “Mean-value analysis of closed multichain queuing networks.” Journal of the ACM (JACM) 27.2 (1980): 313-322.

[64] Schweitzer, Paul J. “Approximate analysis of multiclass closed networks of queues.” Proceedings of International Conference on Stochastic Control and Optimization. 1979.

[65] Seidmann, Abraham, J. Paul, and Sarit Shalev-Oren. “Computerized closed queueing network models of flexible manufacturing systems: A comparative evaluation.” Large Scale Systems 12 (1987): 91-107.

[66] Severence, Frank L. “System modeling and simulation: an introduction.” John Wiley & Sons, 2009.

[67] Shanthikumar, J. G., and R. G. Sargent. “A unifying view of hybrid simulation/analytic models and modeling.” Operations research 31.6 (1983): 1030-1052.

[68] Sharifi, Akbar, et al. “METE: meeting end-to-end QoS in multicores through system-wide resource management.” Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. ACM, 2011.

[69] Sorin, Daniel J., et al. “Analytic evaluation of shared-memory systems with ILP processors.” ACM SIGARCH Computer Architecture News. Vol. 26. No. 3. IEEE Computer Society, 1998.

[70] Suri, Rajan, Sushanta Sahu, and Mary Vernon. “Approximate mean value analysis for closed queuing networks with multiple-server stations.” Proceedings of the 2007 Industrial Engineering Research Conference. 2007.

[71] Tang, Shanjiang, Bu-Sung Lee, and Bingsheng He. “Dynamic slot allocation technique for MapReduce clusters.” Cluster Computing (CLUSTER), 2013 IEEE International Conference on. IEEE, 2013.

[72] Tian, Fengguang, and Keke Chen. “Towards optimal resource provisioning for running mapreduce programs in public clouds.” Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE, 2011.

[73] Towsley, Don. “An approximate analysis of multiprocessor systems.” Proceedings of the 1983 ACM SIGMETRICS conference on Measurement and modeling of computer systems. ACM, 1983.

[74] Tripathi, S. K., and Menascé Daniel A. “Scheduling issues in heterogeneous multiprocessor systems.” Transputers. Vol. 92. 1992.

[75] Verma, Abhishek, Ludmila Cherkasova, and Roy H. Campbell. “ARIA: automatic resource inference and allocation for mapreduce environments.” Proceedings of the 8th ACM international conference on Autonomic computing. ACM, 2011.

[76] Verma, Abhishek, Ludmila Cherkasova, and Roy H. Campbell. “Play it again, SimMR!.” Cluster Computing (CLUSTER), 2011 IEEE International Conference on. IEEE, 2011.

[77] Verma, Abhishek, Ludmila Cherkasova, and Roy H. Campbell. “Resource provisioning framework for mapreduce jobs with performance goals.” Middleware 2011. Springer Berlin Heidelberg, 2011. 165-186.

[78] Vianna, Emanuel, et al. “Modeling the performance of the Hadoop online prototype.” Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on. IEEE, 2011.

[79] Yang, Hailong, et al. “MapReduce workload modeling with statistical approach.” Journal of grid computing 10.2 (2012): 279-310.

[80] Zaharia, Matei, et al. “Improving MapReduce Performance in Heterogeneous Environments.” OSDI. Vol. 8. No. 4. 2008.

[81] Zhang, Yanyong, et al. “Performance implications of failures in large-scale cluster scheduling.” Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, 2005.

[82] Zhang, Zhuoyao, et al. “Performance modeling and optimization of deadline-driven pig programs.” ACM Transactions on Autonomous and Adaptive Systems (TAAS) 8.3 (2013): 14.

[83] Zhang, Zhuoyao, Ludmila Cherkasova, and Boon Thau Loo. ”Exploiting Cloud Heterogeneity to Optimize Performance and Cost of MapReduce Processing.” Special Issue on Performance and Resource Management in Big Data Applications,ACM SIGMETRICS,March 2015

[84] Zhao, Yi, et al. “Allocation wall: a limiting factor of Java applications on emerging multi-core platforms.” ACM SIGPLAN Notices 44.10 (2009): 361-376.

[85] From Wikipedia, *perf: Linux profiling with performance counters*, https://perf.wiki.kernel.org/index.php/Main_Page

[86] http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[87] Hbench - http://www.eecs.harvard.edu/vino/perf/hbench/readme.html

[88] LMBench - http://www.bitmover.com/lmbench/

[89] UnixBench - https://code.google.com/p/byte-unixbench/

[90] Hadoop, Documentation and open source release, http://hadoop.apache.org/core/

[91] Accumulo, http://accumulo.apache.org/

[92] Cassandra, http://cassandra.apache.org/

[93] Cloudera, http://www.cloudera.com/

[94] White House PDF - http://www.whitehouse.gov/sites/default/files/microsites/ostp/

[95] Corona, https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920

[96] Flume, http://flume.apache.org/

[97] MapReduce, Apache, Hadoop. "Mumak Map-Reduce Simulator." (2009). https://issues.apache.org/jira/browse/MAPREDUCE-728

[98] Hive, http://hive.apache.org/

[99] http://www.coker.com.au/bonnie++/

[100] http://dbench.samba.org/

[101] http://www.tux.org/ mayer/linux/bmark.html

[102] https://hbase.apache.org/

[103] http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

[104] http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

[105] http://www.ibm.com/smarterplanet/us/en/business_analytics/article/it_busines_intelligence.html

[106] http://odbms.org/download/Pro Hadoop Ch. 6.pdf

[107] Oozie, http://oozie.apache.org/

[108] http://answers.oreilly.com/topic/459-anatomy-of-a-mapreduce-job-run-with-hadoop/

[109] Pig, http://pig.apache.org/

[110] Sqoop, http://sqoop.apache.org/

[111] http://hadoop.apache.org/docs/r0.18.3/mapred_tutorial.html

[112] http://developer.yahoo.com/blogs/hadoop/anatomy-hadoop-o-pipeline-428.html

[113] YARN, http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[114] http://www.quora.com/Facebook-Engineering/How-is-Facebooks-new-replacement-for-MapReduce-Corona-different-from-MapReduce2

Curriculum Vitae

Shouvik Bardhan is a software developer with over 25 years of experience in designing and implementing software in a myriad of domains. He currently works with GISFederal as a Software Engineer involved in the development of cloud based software system for the US Department of Defense. Prior to that he held software positions at HPTi, Electrosoft Services Inc., Landmark Systems, Applied Information Sciences and completed several consulting engagements as an independent contractor. He holds a Bachelors of Science in Computer Science and Engineering from Indian Institute of Technology, Kharagpur, India. He also holds a Masters of Science in Computer Science from The Johns Hopkins University, Baltimore, USA. He has published six technical papers in journals and conference proceedings.