

AN EXPERIMENTAL STUDY OF MONOLITHIC SCHEDULER
ARCHITECTURE IN CLOUD COMPUTING SYSTEMS

BY

GOURAV KHANEJA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

ABSTRACT

Scheduling in large scale computing clusters is critical to job performance and resource utilization. As the cluster size grows to thousands of machines and scheduling needs become complex and varied, scheduling in cloud-scale clusters presents unique challenges. To encourage the development of innovative schedulers, there is a need for an experimental framework to analyze scheduling performance over large clusters, using relatively modest resources. In this thesis, we present an experimental scheduler testbed to study job scheduling in emulated cloud-scale clusters. We show that the performance of the scheduler in an emulated cluster models closely the same in a real cluster of the same size. We use the testbed to evaluate the monolithic scheduler architecture, a popular scheduling architecture, in a 6000 node emulated cluster over realistic workload. We conclude that scheduling algorithms should embrace randomness in order to beat resource contention. We infer that scheduling in the monolithic architecture is a network I/O intensive process. We calculate the optimal value of design parameters for the monolithic architecture for Google workload.

Hadoop YARN is a popular open-source cluster management framework which can be seen as an implementation of the monolithic scheduler architecture. We evaluate the three default scheduling policies in Hadoop YARN: Capacity, Fair and Fifo, over realistic workload. Based on our experiments, we observe that Fifo scheduling results in unbalanced load across cluster machines and is not suitable for enterprise clusters. We study the trade-offs exploited by Capacity and Fair scheduler: while the Fair scheduler offers less scheduling delay by avoiding head-of-the-line blocking problem, it may drop applications in case the load increases. On the other hand, the Capacity scheduler does not drop any application but errs on the side of higher scheduling delay.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Roy H. Campbell for his invaluable guidance which has made this study possible. I would like to thank him for providing me the freedom to shape the direction of research projects and giving me the opportunity to be a part of Systems Research Group (SRG). It had been a wonderful learning experience and have taught me a great deal about academic research.

I would like to thank Faraz Faghri and Read Sprabery for their key inputs in the design of experimental testbed. I am also thankful to Shadi Abdollahian, Mayank Pundir, John Bellessa and all the members of Systems Research Group for the vibrant research environment. I would like to thank Professor Cristina Abad and Professor Brighten Godfrey for insightful and thoughtful discussions. I would like to thank Sreevatsan Raman from Cask Data Inc for giving me the opportunity to work on Hadoop YARN.

My masters study has been financially supported by University of Illinois at Urbana-Champaign, Systems Research Group and Intel Corporation, for which I am truly thankful. SRG has provided us with more than enough computing resources for carrying out vital experiments which make up the core of this study.

Finally, I would like to thank my parents and my brothers for their love, support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER 1 INTRODUCTION	1
1.1 Technical Contributions	4
1.2 Thesis Outline	5
CHAPTER 2 CLUSTER SCHEDULER ARCHITECTURES	6
2.1 Background	6
2.2 Overview of Scheduler Architectures	6
2.3 Monolithic Scheduler Architecture	8
CHAPTER 3 SCHEDULER TESTBED: DESIGN AND IMPLEMENTATION	11
3.1 Workload Generator and Google Traces	11
3.2 Scheduler	16
3.3 Cluster Emulator	17
CHAPTER 4 EVALUATION OF MONOLITHIC SCHEDULER ARCHITECTURE	20
4.1 Monolithic Scheduler: Design and Implementation	21
4.2 Heartbeat Interval	23
4.3 Path Limit	26
4.4 Scheduling Constraints	26
4.5 Scheduling Algorithm	29
4.6 Components of Scheduling Delay	30
4.7 Verification of Cluster Emulation	34
CHAPTER 5 EVALUATION OF HADOOP YARN SCHEDULERS	39
5.1 Overview of Hadoop YARN Architecture	39
5.2 Experimental Set-up	42
5.3 Experimental Results	43

CHAPTER 6 RELATED WORK	62
6.1 Cluster Schedulers	62
6.2 Analysis of Scheduling Workload	66
CHAPTER 7 CONCLUSION	68
7.1 Future Work	69
REFERENCES	71

LIST OF TABLES

3.1	Scale of scheduling workload in Google traces.	13
3.2	Attributes of jobs in Google traces.	13
3.3	Attributes of tasks in Google traces.	14
4.1	Variation of failure rate with heartbeat interval.	23
4.2	Failure rate for different scheduling algorithms.	30

LIST OF FIGURES

1.1	Cumulative distribution of number of tasks in jobs in Google traces.	2
1.2	Cumulative distribution of task durations in Google traces.	3
2.1	Overview of scheduler architectures. Popular implementation of each architecture is mentioned in parenthesis.	7
2.2	Design of Monolithic Scheduler Architecture.	10
3.1	Architecture of Scheduler Testbed. Orange/ dashed box represents process, while solid/ Blue box represents machine.	12
3.2	Cumulative distribution of number of scheduling attempts of tasks in Google traces.	15
3.3	Memory usage and cpu load of Cluster Emulator to emulate 6000 nodes during a four hour experiment. The machine consists of 32 logical cores and 90 GB of memory. The Emulator collects cluster resource utilization data every second and stores it in memory, which results in constantly increasing memory usage.	19
4.1	Scheduler cpu load for different heartbeat intervals.	24
4.2	Cumulative distribution of job-wise scheduling delay for different heartbeat intervals.	25
4.3	Cumulative distribution of job-wise scheduling delay for different path limits. <i>No limit</i> means that there is no maximum bound on the number of requests that can be served concurrently.	27
4.4	Cluster cpu utilization for different path limit values. <i>No limit</i> means that there is no maximum bound on the number of requests that can be served concurrently.	28
4.5	Effect of scheduling constraints on scheduler cpu load.	29
4.6	Cluster cpu utilization for different scheduling algorithms.	31
4.7	Relationship between job and task delay distribution for Google workload for our implementation of monolithic scheduler.	32
4.8	Distribution of network and scheduler delay that make up the total task delay.	33

4.9	Job delay distribution for emulated and real clusters of small and large size.	35
4.10	Task delay distribution for emulated and real clusters of small and large size.	36
4.11	Variation of scheduler cpu load with time for real and emulated clusters of small and large size.	37
4.12	Variation of number of failed requests (cumulative) with time for emulated and real clusters of small and large size.	38
5.1	Hadoop YARN Architecture.	40
5.2	Cumulative distribution of AM delay for Capacity, Fair and Fifo Scheduler. This is a job-wise delay distribution.	44
5.3	Cumulative distribution of total Scheduling delay for Capacity, Fair and Fifo Scheduler. This is a job-wise delay distribution.	45
5.4	Cumulative distribution of Allocation delay for Capacity, Fair and Fifo Scheduler. This is a task-wise delay distribution.	46
5.5	Cumulative distribution of Task-Start delay for Capacity, Fair and Fifo Scheduler. This is a task-wise delay distribution.	47
5.6	Variation of standard deviation of cpu usage across nodes with time, for YARN schedulers.	48
5.7	Variation of total cpu utilization of cluster with time, for YARN schedulers.	49
5.8	Variation of number of running applications with time, for YARN schedulers.	50
5.9	Variation of number of healthy nodes in the cluster with time, for YARN schedulers.	51
5.10	Variation of cumulative number of failed applications with time, for YARN schedulers.	52
5.11	Variation of scheduling delay with trace speed for Capacity scheduler.	55
5.12	Variation of scheduling delay with trace speed for Fifo scheduler.	56
5.13	Effect of trace speed on scheduling delay for Fair scheduler.	57
5.14	Effect of trace speed on application failures for Fair scheduler.	58
5.15	Effect of task duration on cluster cpu utilization for Fair scheduler.	59
5.16	Effect of task duration on running application count for Fair scheduler.	60
5.17	Effect of task duration on node failures for Fifo scheduler.	61

CHAPTER 1

INTRODUCTION

Building and maintaining large clusters of commodity machines is an expensive and power consuming task, which is why it is important to utilize them well. To increase the utilization, such clusters are shared between a wide variety of computing applications, including but not limited to batch data analytics frameworks like MapReduce [1], graph processing frameworks like Pregel [2], real-time streaming frameworks like Storm [3], web requests frameworks and a wide variety of data-stores like Spanner [4], Dremel [5], Cassandra [6] and HBase [7]. In May 2011, Google released a 29 days long cluster trace from one of their sizable multi-tenant cluster, consisting of 12,583 machines [8]. Charles Reiss et al. [9] analyzed the trace and concluded that the most notable workload characteristic is the heterogeneity of jobs in terms of number of tasks in the jobs (Figure 1.1), run-time duration of constituting tasks (Figure 1.2), cpu & memory requirements, hardware & kernel constraints and inter-arrival period between jobs. Figure 1.1 shows the cumulative distribution of the number of tasks in jobs. While 75% of the jobs have only one task, a small number of jobs contains the majority of the tasks, giving rise to a long tail. Figure 1.2 shows the distribution of the running duration of all tasks. The duration of the longest task is almost four order of magnitude larger than that of the shortest task. Apart from heterogeneity in workload, cluster machine configurations are also dynamic and heterogeneous in nature.

Such a heterogeneity in workload (combined with heterogeneity in cluster machines) significantly reduces the effectiveness of slot and core based scheduling. Besides, majority of the jobs contain a small of number of short duration tasks which require quick scheduling decisions. Figure 1.2 shows that more than 50% of tasks completes within 16 minutes. On one hand, there are interactive query based jobs which are latency sensitive, while on the other hand, there are complex jobs with thousands of tasks with specific

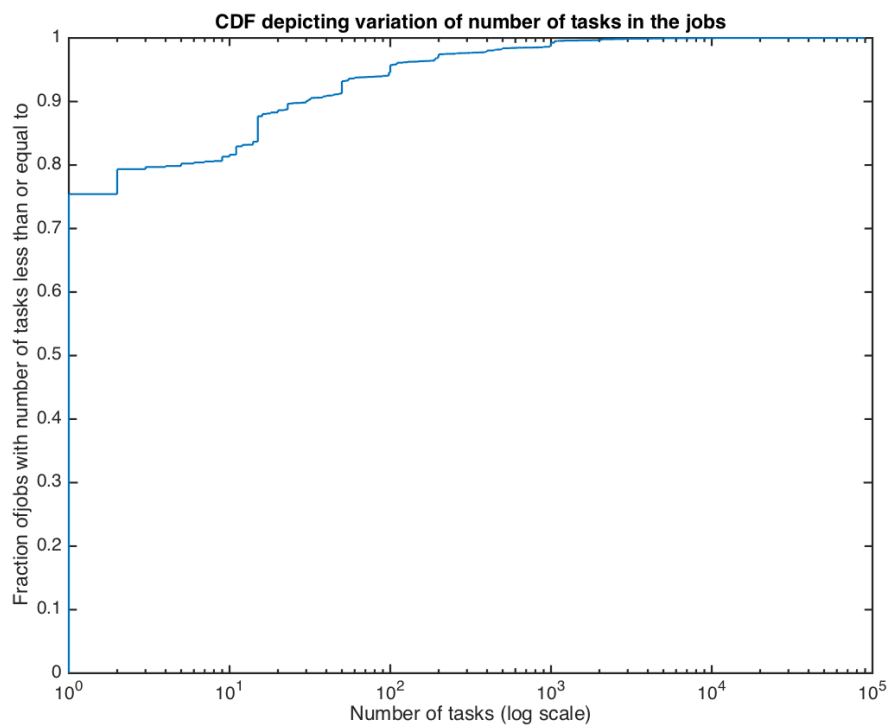


Figure 1.1: Cumulative distribution of number of tasks in jobs in Google traces.

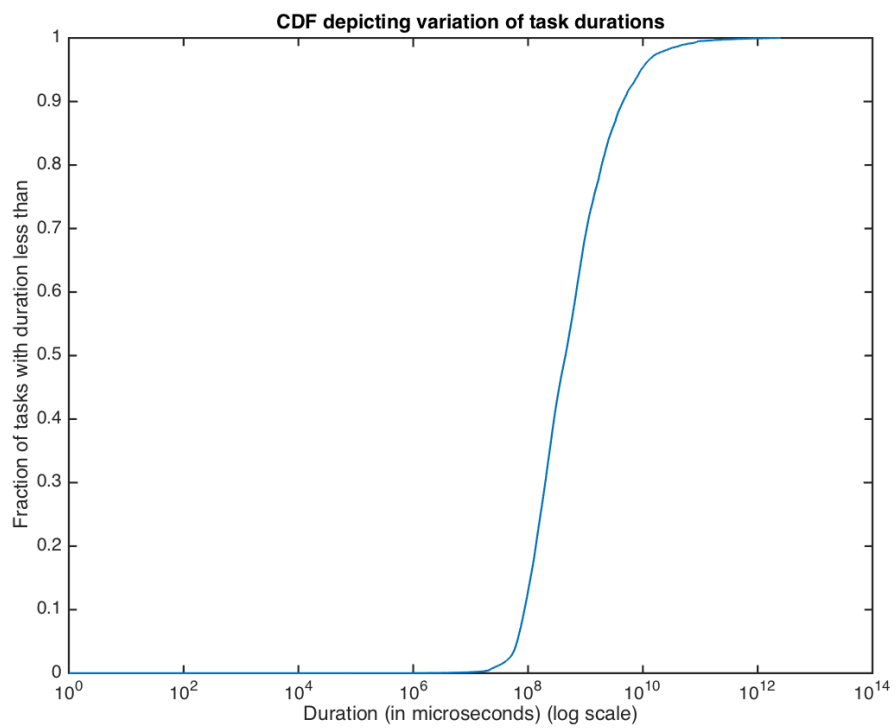


Figure 1.2: Cumulative distribution of task durations in Google traces.

scheduling needs. Apart from fast scheduling decisions, schedulers need to enforce global policies, respect job priorities and ensure fairness.

To tackle the unique scheduling problem in cloud computing, research community has proposed a variety of radically different scheduler architectures and provided their implementations. A brief survey of popular scheduling architectures is presented in Chapter 6. Some popular scheduler implementations from different architectures include Mesos [10], Yet Another Resource Negotiator (YARN) [11], Omega (Google) [12], Sparrow [13] and Apollo (Microsoft) [14]. To evaluate existing architectures with different design parameters and encourage the development of innovative scheduling architectures, there is a need to compare scheduling design and algorithms in a way that is not tied to a specific implementation. In this thesis, we are trying to fill that gap and present an open source experimental testbed for scheduler evaluation under realistic workload based on Google traces. We used the testbed to evaluate *monolithic* scheduler architecture [15], a popular scheduler architecture. We also evaluated scheduling policies in Hadoop YARN, a popular implementation of monolithic scheduler architecture.

1.1 Technical Contributions

We briefly describe the contributions of this study as follows.

- We build an experimental testbed for evaluating scheduler architectures under varying realistic workloads (Chapter 3). The testbed replays traces from Google clusters to generate workload for experiments. It consists of a software based *Cluster Emulator* to emulate large scale clusters using relatively modest resources. We verify that the performance of scheduler in an emulated cluster is a strong indicative of the same in a real cluster of the same size. We also provide an abstract implementation of scheduling components, which can be used to implement different scheduler architectures.
- Using the testbed, we thoroughly study the performance of the monolithic scheduler architecture [15] (Chapter 4) on a 6000 node emulated cluster with workload generated by replaying Google traces. We study

the effect of heartbeat interval and scheduling constraints on scheduling delay and cluster resources utilization. We analyze the performance impact of different scheduling algorithms. We also study the impact of different concurrency levels in scheduler to handle job requests. We present a component-wise analysis of scheduling delay.

- We evaluated the default scheduling policies in Hadoop YARN, a popular open source implementation of the monolithic architecture. We evaluated the three YARN schedulers: Capacity, Fair and Fifo on a 22 node real cluster over workload generated by replaying Google traces. Since YARN contains a per-node daemon called *Node Manager (NM)*, we evaluated YARN on a real cluster instead of using cluster emulation. We use the Workload Generator from the testbed to create YARN clients.

1.2 Thesis Outline

The rest of the thesis is organized as follows: In Chapter 2, we provide an overview of scheduler architectures, followed by a brief description of monolithic architecture. In Chapter 3, we describe the design and architecture of experimental testbed, along with a brief description of Google cluster traces. In Chapter 4, we present a thorough experimental evaluation of monolithic architecture. In Chapter 5, we present a thorough experimental evaluation of the three schedulers in Hadoop YARN: Capacity, Fair and Fifo. We describe the related work in Chapter 6. We conclude in Chapter 7.

CHAPTER 2

CLUSTER SCHEDULER ARCHITECTURES

In this chapter, we provide an overview of popular cluster scheduler architectures in cloud computing literature. We then briefly describe the design of monolithic scheduler architecture which is the focus of this study.

2.1 Background

Scheduling workload [8] [15] consists of a series of *jobs*, which consists of one or more *tasks*, each of which can run as a (possibly multi-threaded) process on a machine. To schedule a job, one needs to map each task, in a job, to a machine which has enough resources (cpu cores and memory) available to run that task. Apart from resource requirements, tasks may specify constraints to run on machines with specific properties, such as machines with GPUs or machines with specific kernel versions. Jobs are annotated with priorities and user-names. It may be desirable to share cluster resources fairly between different users. Besides, a scheduler may preempt low priority tasks in order to provide resources for high priority tasks.

A *scheduling agent* is a program which receives job requests and maps tasks to machines. A scheduling agent generally maintains a data structure, called *cluster state*, which represents the resources available in the cluster. Cluster state needs to be periodically synchronized with the actual available resources. A scheduler architecture may consist of one or more scheduling agent(s).

2.2 Overview of Scheduler Architectures

In his PhD thesis, Konwinski [15] provides a taxonomy of scheduler architectures. Figure 2.1 gives a schematic overview of different scheduler architec-

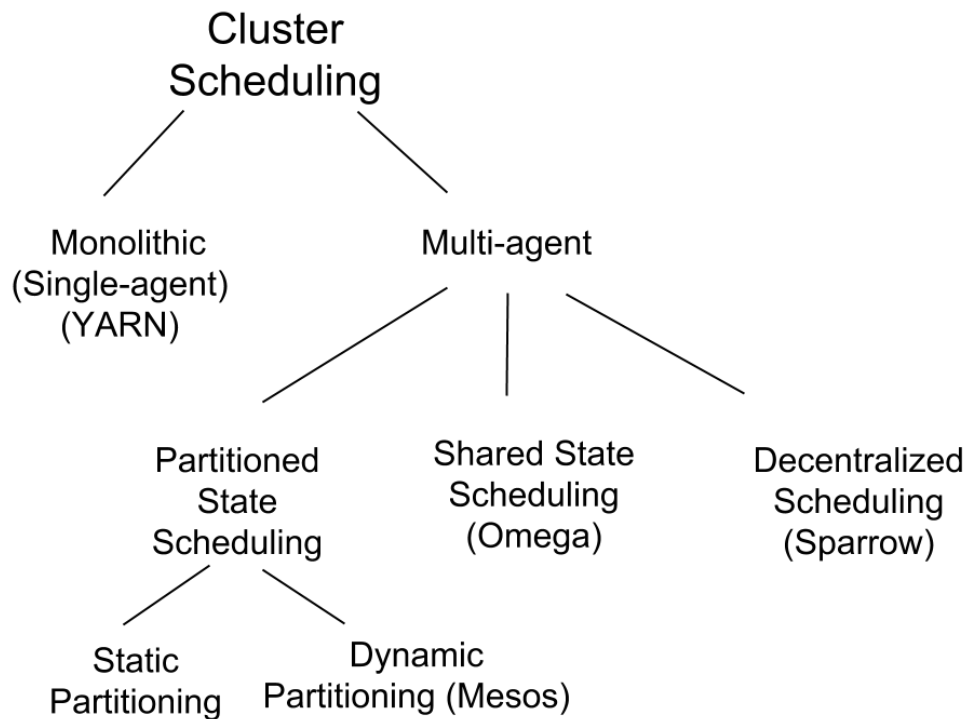


Figure 2.1: Overview of scheduler architectures. Popular implementation of each architecture is mentioned in parenthesis.

tures. Broadly speaking, cluster scheduler architectures can be classified as *single-agent* or *multi-agent*. A single-agent architecture runs a single instance of scheduling agent which has an exclusive access to all the machines in the cluster. The single scheduling agent handles all the job requests, possibly using thread level parallelism. It is easy to implement inter-job constraints and enforce global policies with such a design. Single-agent architecture is also referred to as monolithic scheduler architecture or simply monolithic architecture. Schedulers implementing monolithic architecture are referred to as *monolithic schedulers*.

A multi-agent architecture consists of two or more scheduling agents which share the cluster machines. The job requests can be divided between the agents through specific policies. For example, a simple round robin policy may be used for load balancing or division of requests may be carried out according to the job type. Although such an architecture is scalable over multiple machines, it needs to address the problem of synchronization of

cluster state between multiple agents. In this section, we will introduce three multi-agent architectures: *partitioned state*, *shared state* and *decentralized*.

In a partitioned state architecture, cluster resources are partitioned between scheduling agents according to job demands and global policies. Such a partitioning eliminates interference between agents at the cost of potential decrease in cluster utilization. It is divided into two subtypes: *static* and *dynamic*. As the name suggests, in static partitioning, the partitions do not change. In dynamic partitioning, a central component is responsible to dynamically calculate the resource partitions between agents based on their requirements. Mesos [10] is an Apache project which is based on the principle of dynamic cluster partitioning.

Unlike partitioned state architecture, in a shared state architecture, all cluster resources are available to all scheduling agents. A resilient central copy of cluster state is maintained to avoid interference between agents. In order to claim a resource, an agent needs to update the central cluster state in an atomic transaction. In case of conflict where two agents are trying to claim the same resource, one of the transaction will fail. If an agent encounters a failed transaction, it re-calculates its requirements and tries again. The performance of such an architecture depends on the conflict / interference rate between scheduling agents, which in turn depends on the workload. Schwarzkopf et al. [12] have conducted experiments on shared state architecture and observed the conflict rate to be low for Google scheduling workload.

In decentralized architecture, all scheduling agents have access to the entire cluster and work independently of each other. Each cluster machine consists of multiple slots with specific resources. Each slot in a machine maintains a queue of tasks that want to use the slot resources, which are executed in FIFO order. Scheduling agents may query cluster machines for the length of their task queues in order to make intelligent scheduling decisions. Ousterhout et al. [13] present an implementation of decentralized architecture, called Sparrow.

2.3 Monolithic Scheduler Architecture

Monolithic schedulers belong to single-agent class of architecture classification, where a single scheduling agent handles all the requests. Figure 2.2

describes the design of monolithic schedulers. The single scheduling agent maintains long lived TCP connection to each cluster machine, which are used to receive periodic heartbeats. The heartbeat may contain resource usage and health status of the machine. It is used to keep the in-memory cluster state up-to-date. The heartbeat interval is critical to the performance of the scheduler. In Section 4.2, we evaluate how the performance of monolithic schedulers changes with the heartbeat interval ?

A monolithic scheduler receives job requests and may execute them either in a FIFO manner (*single-path* monolithic scheduler) or may use thread-level parallelism (*multi-path* monolithic scheduler). A FIFO execution of requests may suffer from *head-of-the-line* blocking problem, where a complicated scheduling decision delays the execution of simpler scheduling requests. We evaluate the performance difference between single-path and multi-path schedulers in Section 4.3.

A scheduler may implement preemption of low priority tasks in case sufficient resources are not available for a high priority job. It should also ensure fairness of resource allocation between users.

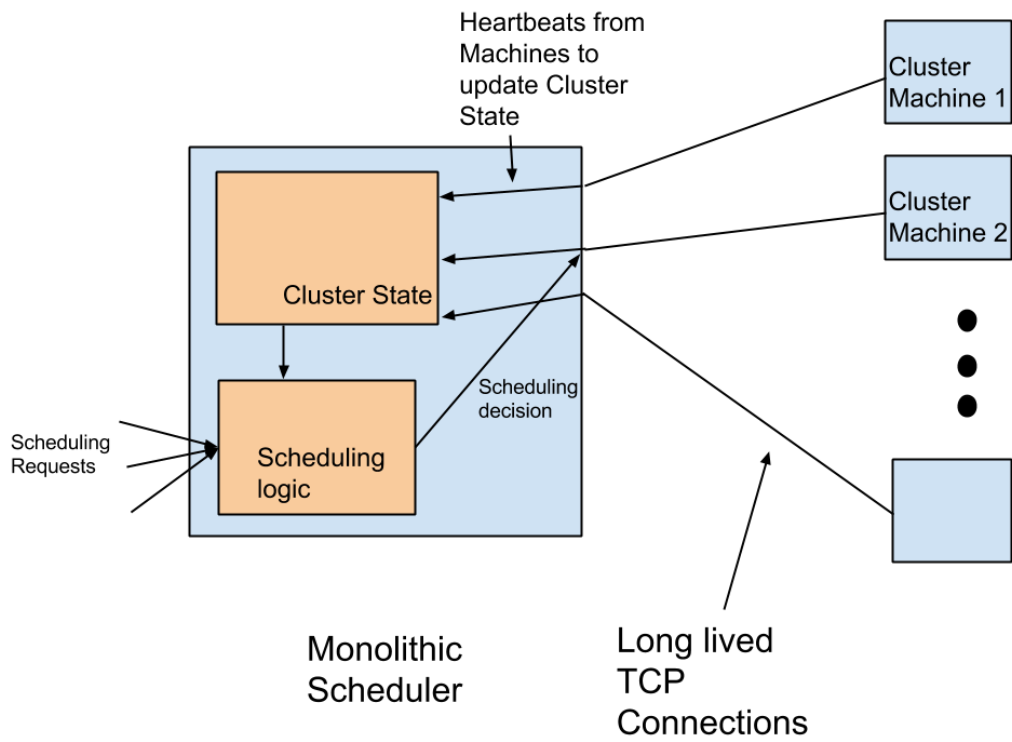


Figure 2.2: Design of Monolithic Scheduler Architecture.

CHAPTER 3

SCHEDULER TESTBED: DESIGN AND IMPLEMENTATION

We have developed an experimental testbed to facilitate performance testing of different scheduler architectures over large emulated clusters with diverse workloads. We used the testbed to analyze how different design parameters affect the performance of monolithic schedulers ? The testbed can be divided into three components:

- A *Workload Generator* to generate job requests to be scheduled on cluster nodes.
- An abstract implementation of scheduler modules, which can be inherited by a specific scheduler implementation, which is to be tested.
- A *Cluster Emulator* to emulate large clusters (consisting of thousands of machines) with significantly fewer resources.

Figure 3.1 shows the architecture of the testbed with monolithic scheduler implementation. The testbed is written in Java and spans about 7000 lines of code. The source code is open for comments and contributions [16] [17]. In the rest of this chapter, we would describe each component and their interactions with each other.

3.1 Workload Generator and Google Traces

The main task of Workload Generator is to generate job requests for scheduling agent(s) by replaying a user-specified trace, with a given speed for a given time. Different traces and request generation models can be plugged into Workload Generator. For precise simulation of job inter-arrival periods, the entire trace is read in to the memory before starting the experiment. Each job request runs as a thread (orange/dashed box in Figure 3.1) and

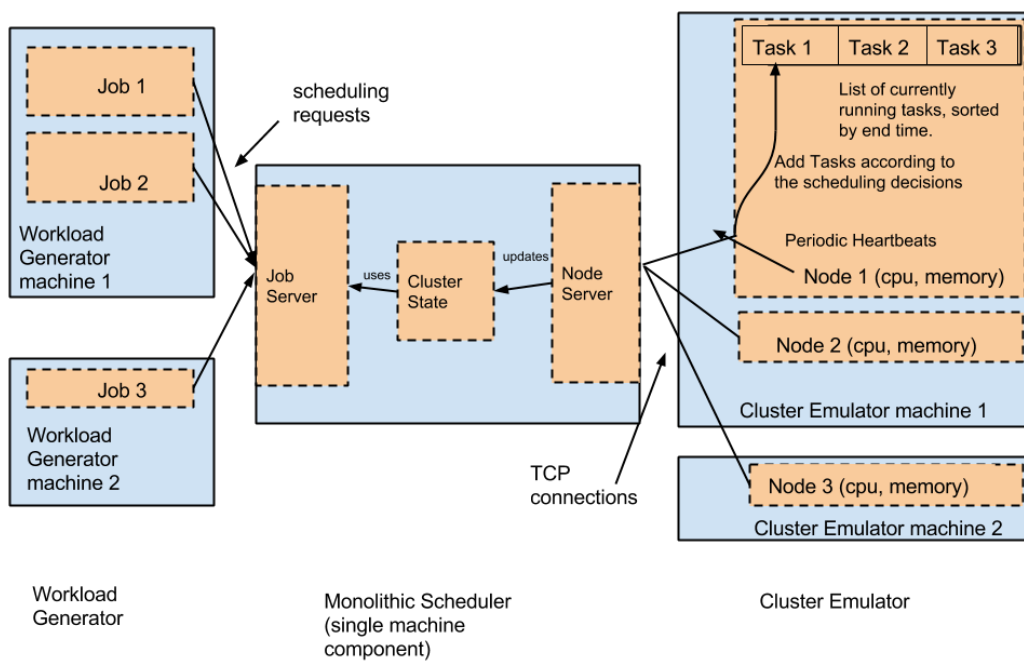


Figure 3.1: Architecture of Scheduler Testbed. Orange/ dashed box represents process, while solid/ Blue box represents machine.

Table 3.1: Scale of scheduling workload in Google traces.

Trace duration	29 days
Cluster size	12,583 nodes
Number of unique jobs	672,074
Number of unique task	25,424,731
Number of tasks with at least one scheduling constraint	1,405,572
Number of unique constraints	17

logs scheduler response, scheduling decisions, task and job delays. All job threads share a single buffered file writer for writing logs, which is protected by locks for thread safety.

For the experiments in this report, we are using scheduling workload from Google cluster traces, published in May 2011 [8]. Charles Reiss et al. [9] thoroughly analyzed the trace and observed heterogeneity of workload as the most notable characteristic. Table 3.1 shows some important statistics about the scale of trace. The trace consists of jobs, which contains one or more tasks. Each task in a job has cpu and memory requirements and is tagged with a submission, scheduled and finish timestamp. Some of the tasks specify scheduling constraints, which restrict the set of machines the task can run on. For example, task t_1 should only run on machines with GPUs, or task t_2 should run on machines with kernel version greater than 2.6.1. Table 3.2 and Table 3.3 summarize characteristics of jobs and tasks in Google traces respectively. The trace also describes the configuration of machines in the cluster in an anonymized form. Note that this is a simplified description of trace format, suitable for further discussion in this report. For a detailed description, reader is recommended to the technical report describing Google traces by Reiss et al. [8].

Table 3.2: Attributes of jobs in Google traces.

Field	Description
<i>timestamp</i>	Timestamp of the event
<i>job id</i>	unique Job identifier
<i>event type</i>	enum{submit, schedule, finish}

To keep the analysis and characterization tractable, we have made three simplifying assumptions as follows.

- According to the trace, the machines in the cluster are added, removed

Table 3.3: Attributes of tasks in Google traces.

Field	Description
<i>timestamp</i>	Timestamp of the event
<i>job id</i>	parent Job identifier
<i>index</i>	Task index within the job
<i>event type</i>	enum{submit, schedule, finish}
<i>cpu</i>	Resource request for CPU cores
<i>memory</i>	Resource request for Memory in MB
<i>constraints</i>	A set of scheduling constraints.

and updated (in terms of hardware configuration or kernel versions) over the time. Although there are 8966 addition, 10556 removal and 7380 update events, the number of machines in the cluster remains fairly constant. For all the experiments in this report, we assume that the cluster consists of a constant set of 6000 heterogeneous machines for the entire duration of the trace. We plan to address this assumption in future experiments.

- A task may fail and need to be rescheduled. Thus, a task may be scheduled more than once. Figure 3.2 shows the variation of scheduling attempts of tasks. Although 90% of the tasks have one scheduling attempt, a long tail results in significantly large number of re-scheduling requests. Although such a distribution affects the scheduler workload and performance, we will ignore re-scheduling requests in the first version of Workload Generator so as to keep the analysis tractable. We plan to address this assumption in the future.
- Actual resource usage of tasks differ significantly from the amount requested and varies over time. However, since these variations do not directly affect the performance of the scheduler, we will not consider actual resource usage of tasks.

In order to model and characterize the workload, we identified a minimal set of dimensions which define jobs and tasks. We analyzed the traces and removed the following dimensions.

- For more than 99.8% of jobs, all constituting tasks request the same cpu and memory resources. Furthermore, only 0.4% of tasks change

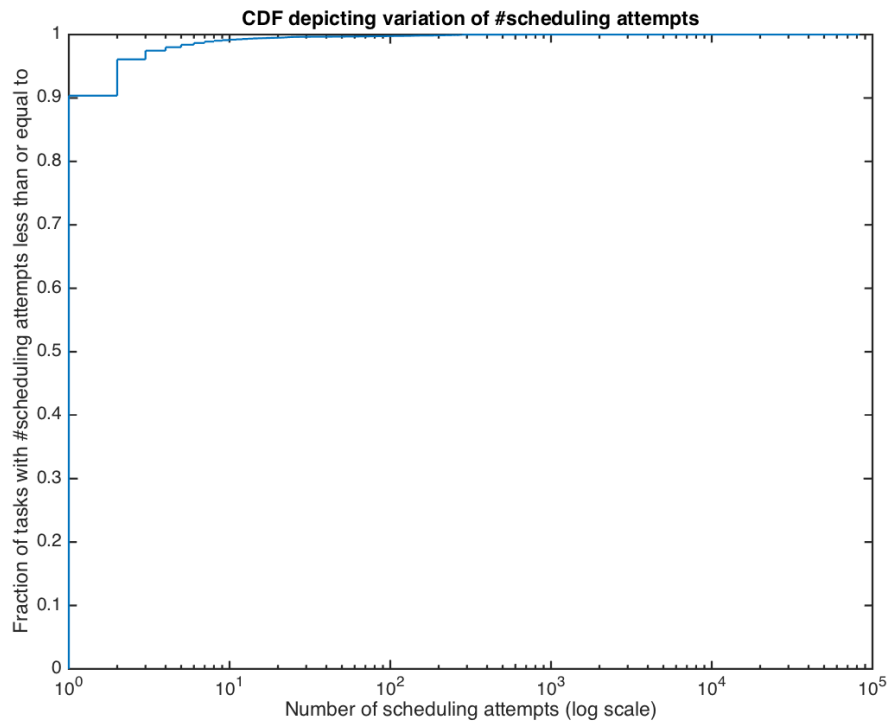


Figure 3.2: Cumulative distribution of number of scheduling attempts of tasks in Google traces.

their cpu and memory requirements during their lifetime. Therefore, we can safely represent resource requirements of all tasks in a job by a couple of values for cpu and memory.

- For 99% of the jobs, all constituting tasks arrive within 600 microseconds of each other. For 99.9% of the jobs, the interval becomes 3 milliseconds. Since this interval is negligible as compared to average job's inter-arrival period, we can safely represent arrival time of all tasks in a job with a single value.

Tasks in a job do not share the same run-time duration and therefore cannot be represented by a single value. In summary, a job can be represented by five attributes: (1) arrival time (2) cpu requirement (3) memory requirement (4) per task run-time duration (5) per task scheduling constraints.

3.2 Scheduler

The second component (middle one in Figure 3.1) is the scheduler implementation to be analyzed. The component can be changed to evaluate different architectures and design aspects. We have provided abstract implementation of some of the basic scheduling modules as follows, which can be inherited by different scheduler implementations.

- *Cluster state*: This module maintains in-memory data structures representing the current state (resource availability) of nodes in the cluster. These data structures are optimized to support fast scheduling decisions.
- *Node server*: This module is responsible for periodically collecting resource usage values from cluster nodes and updating cluster state. Current implementation of node server maintains TCP connection to each node in the cluster, through which it receives periodic heartbeat messages containing resource availability.
- *Job server*: This component is responsible for receiving job requests and making scheduling decisions for all tasks in a job. It needs a

pluggable *scheduling algorithm* to calculate the schedule by using information from cluster state. The scheduling algorithm is provided by a specific scheduler implementation.

In this report, we present results from experiments on monolithic scheduler architecture. We describe our implementation of monolithic scheduler in Section 4.1. In future, we will use the testbed to study other architectures.

3.3 Cluster Emulator

In order to evaluate schedulers on large scale clusters with tens of thousands of machines, we needed a way to emulate large number of machines from the point of view of scheduling agents, with fewer resources. The goal of emulation is to ensure that the performance of scheduling agent(s) in an emulated cluster strongly represents the same in a real cluster of the same size.

Cluster Emulator spawns multiple processes, each of which emulates a single cluster machine. We refer to such a process as *node process*. In Figure 3.1, orange/dashed box in Cluster Emulator machine, represents a node process. Each node process is assigned logical cpu and memory values (corresponding to cluster machines). A node process maintains a long-lived TCP connection with the scheduling agents(s) and sends periodic heartbeats, consisting of health reports and latest resource utilization/availability values. It maintains a list of tasks which are currently 'running' on the corresponding cluster machine. It receives scheduling decisions from agents and add tasks to the list if enough resources are available. Addition of a task to this list in the node process corresponds to the start of the execution of task on the corresponding cluster machine. When a task completes its execution (according to run-time duration), node process removes the task from the list and releases the resources. The run-time duration of the tasks are extracted from traces. The list is kept sorted according to end timestamp of tasks for efficient implementation.

Each node process consists of four executing threads. Unix kernel imposes a maximum limit on the number of threads. On Linux kernel version 3.5.0-43-generic (used in our experiments), this limit is 32,317 which corresponds

to a maximum of 8000 node processes per machine. In this paper, we emulated 6000 nodes using one machine consisting of 32 logical cores and 128 GB of memory. The emulator processes ran with a heap space of 90 GB. Figure 3.3 shows the cpu load and memory usage of the machine used for emulation. Note that cpu load is defined as the number of threads waiting for cpu, averaged over one minute. Since the threads in node processes are not cpu intensive, the cpu load of the emulation of 6000 nodes remains well below 15 for the entire duration of the experiment. Thus, 32 core machine used in experiments handles the emulation very well. Cluster Emulator collects resource utilization statistics of each node every second. It keeps the data in memory and aggregates it to get per-second cluster-wide resource utilization statistics, after the experiment has ended. This is why memory usage of Cluster Emulator constantly increases as the experiment continues. A memory of 90 GB is sufficient for a four hour experiment. In Section 4.7, we verify the validity of emulation by comparing scheduler performance metrics collected from real and emulated clusters of the same size.

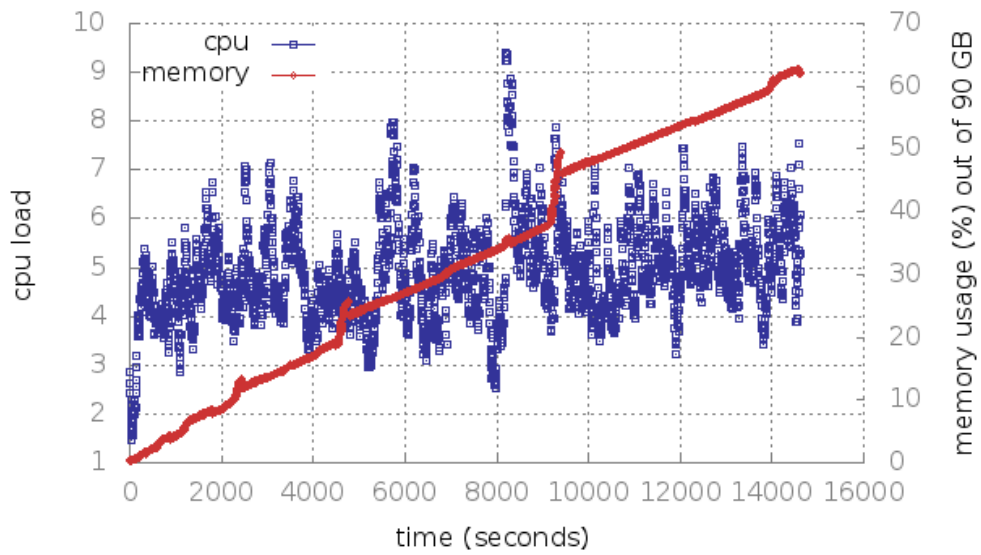


Figure 3.3: Memory usage and cpu load of Cluster Emulator to emulate 6000 nodes during a four hour experiment. The machine consists of 32 logical cores and 90 GB of memory. The Emulator collects cluster resource utilization data every second and stores it in memory, which results in constantly increasing memory usage.

CHAPTER 4

EVALUATION OF MONOLITHIC SCHEDULER ARCHITECTURE

We used the testbed (described in Chapter 3) to thoroughly evaluate monolithic scheduler architecture. We used one machine for each component: Workload Generator, monolithic scheduler and Cluster Emulator. We used Dell 320 machines, with four quad core processors, giving a total of 32 logical cores after enabling hyper-threading. Each machine consists of 128GB RAM, 64GB SSD, 512GB of storage and are connected to each other with 1 gigabit per second Ethernet. The machines run Ubuntu distribution with kernel version 3.5.0-43-generic.

For all experiments, we replayed workload from Google traces for a duration of 1 hour (3600 seconds), unless otherwise stated. To facilitate shorter experiment durations while covering a major portion of trace, we replayed the traces with a speed of 100x for all experiments, unless otherwise stated. All experiments are carried out with a 6000 nodes emulated cluster, unless otherwise stated. We verify the validity of cluster emulation in Section 4.7.

Each experiment was ran twice and as expected, the results from the two runs were highly correlated for all collected metrics. In this chapter, we report results from the first run of each experiment.

For all experiments, we measured the following metrics:

- *Scheduling delay*: For each job, we measured the total time taken by the scheduler to calculate it's schedule i.e. assign a node to each task, as perceived by the job client (Workload Generator). We refer to this delay as scheduling delay of the job.
- *Cluster cpu utilization*: We define resource utilization of the cluster as a ratio of the total resources being used to total resources available. We will only report cluster cpu utilization since it is strongly correlated with cluster memory utilization.
- *Scheduler cpu load*: We measure scheduler cpu load every second. We

use 1-minute load average (average number of jobs waiting to use cpu in last 1 minute) of Linux Top command to get cpu load. We will use the terms scheduler load and scheduler cpu load interchangeably.

- *Failure rate*: We keep track of the percentage of jobs failed to be scheduled on the nodes. We refer to this percentage as failure rate. A job fails if at least one of it's constituting task is not scheduled.

4.1 Monolithic Scheduler: Design and Implementation

We implemented the monolithic scheduler architecture in Java. For each machine in the cluster, scheduler contains a thread (referred to as *node thread*) which maintains a TCP connection to the corresponding node. This connection is used to receive heartbeats (containing health report and resource usage) from the machine. We study the effect of heartbeat interval on scheduler performance in Section 4.2. Each node thread keeps the updated resource availability values for the corresponding machine, which is protected by locks for thread safety. The set of all node threads makes up the cluster state (Section 3.2).

Scheduler listens on a given port for job requests in a thread called *job server* (Section 3.2). For each received request, job server spawns another thread called *request handler*, which serves the request by assigning a node to each of it's task. The job server maintains a thread pool of request handler threads. We study the effect of the size of this thread pool in Section 4.3. A request handler uses a *scheduling algorithm* to calculate a schedule for the job (assignment of a node to each task). We use a default scheduling algorithm shown in Algorithm 1 for all experiments, unless otherwise stated. We study the effect of scheduling algorithm in Section 4.5.

The scheduling decisions are sent to the machines through TCP connections of corresponding node threads. The machines may accept or reject the scheduling decisions, depending on the resources available. A stale or inconsistent cluster state may result in rejection of scheduling decisions. Request handler re-runs scheduling algorithm for tasks which got rejected by machines. In our implementation, a maximum of 1000 attempts are made to assign tasks to machines, after which request handler gives up and marks

Procedure: To calculate per task schedule

input : A job j consisting of n tasks, each of which requires cpu cores, $memory$ GB of memory to run. A task, t may specify a set of scheduling constraints, $constraints_t$, where $1 \leq t \leq n$

output : A map from tasks to cluster nodes, $schedule$

Initialize $schedule =$ an empty map

for each task t in job j **do**

$tries = 0$

$schedule.put(t, null)$

while $++tries < MAX_TRIES$ **do**

select a random node $node$, from cluster state

$node.acquire_lock()$

if $node.availableCPU \geq cpu$ $\&\&$

$node.availableMemory \geq memory$ $\&\&$ $node$ satisfies $constraints_t$ **then**

$node.availableCPU - = cpu$

$node.availableMemory - = memory$

$schedule.put(t, node)$

$node.release_lock()$

break

end

$node.release_lock()$

end

return $schedule$

Algorithm 1: Default Scheduling Algorithm

Table 4.1: Variation of failure rate with heartbeat interval.

Heartbeat Interval	% failed jobs	% failed tasks
100 ms	5.23	4.75
5 s	8.86	5.41
50 s	13.54	7.45
500 s	15.35	8.33

the request as failed.

4.2 Heartbeat Interval

As stated in the above section, nodes in the cluster send periodic heartbeat messages to scheduler which consists of resource (cpu and memory) availability at the node. The heartbeats are used to update the in-memory cluster state at the scheduler, which is used to make scheduling decisions. Longer heartbeat interval results in stale / inconsistent cluster state, which leads to bad scheduling decisions. On the other hand, shorter heartbeat intervals increase the network traffic and scheduler load. We experimented with different heartbeat intervals to study their trade-offs.

Table 4.1 shows the percentage of jobs and tasks which suffered bad scheduling decisions for different heartbeat intervals. As expected, higher heartbeat intervals resulted in higher percentage of failed jobs. Note that a job fails if at least one of its constituting task fails. Figure 4.1 shows the scheduler cpu load over the course of experiment for different heartbeat intervals. A heartbeat interval of 100ms exerts significantly more load than that of 5 and 50 seconds, which are almost equivalent in terms of scheduler cpu load. Figure 4.2 shows the effect of heartbeat interval on scheduling delay of jobs. Lower heartbeat interval of 100 ms suffers higher scheduling delay as compared to its counterparts due to the increase in scheduler cpu load. Cluster utilization (not shown here) remains approximately the same for all heartbeat intervals. We conclude that a heartbeat interval of 5 seconds exploits the trade-off between failure rate, scheduler cpu load and scheduling delay, very well for Google cluster workload.

In the rest of this chapter, we use a heartbeat interval of 5 seconds.

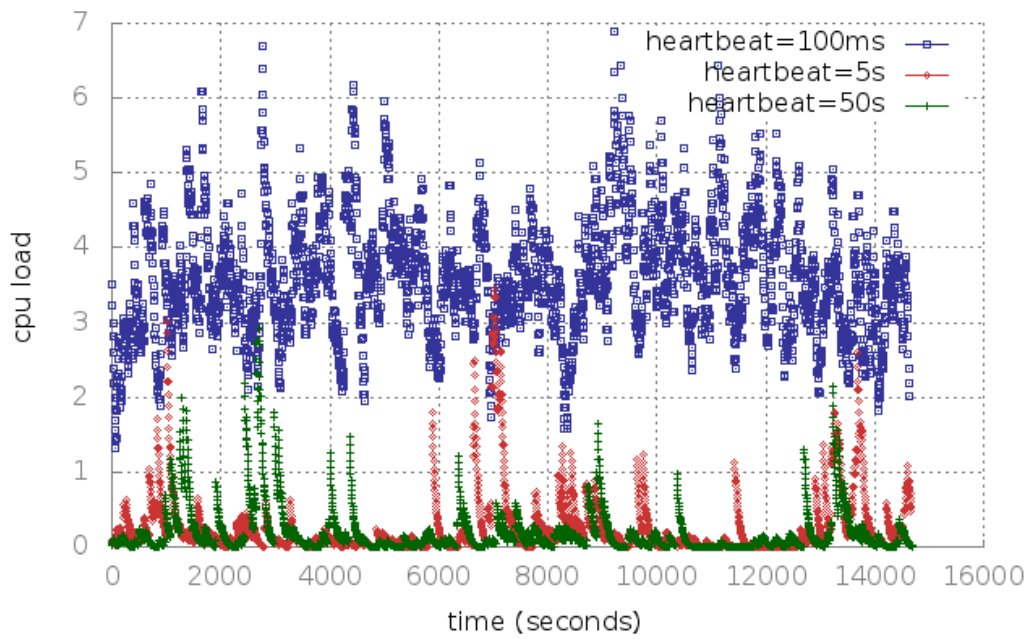


Figure 4.1: Scheduler cpu load for different heartbeat intervals.

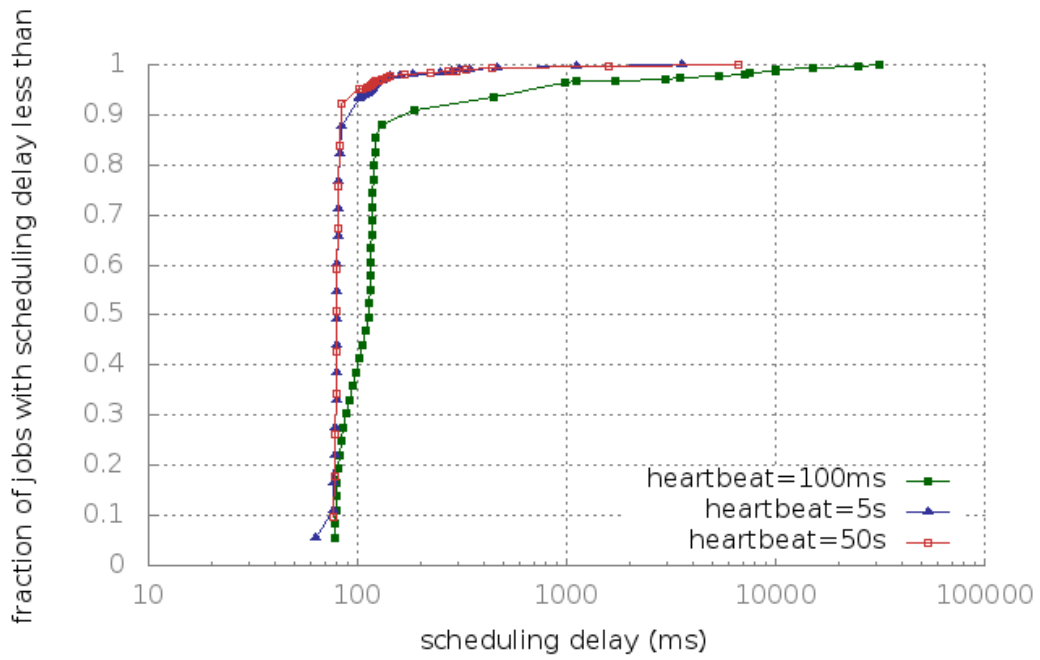


Figure 4.2: Cumulative distribution of job-wise scheduling delay for different heartbeat intervals.

4.3 Path Limit

The job server (Figure 3.1) receives job requests and calculates schedule according to the scheduling algorithm (Algorithm 1). On one extreme, it could serve requests in the order they arrive. On other hand, requests can be served concurrently. In the latter case, requests do not suffer from *head-of-the-line* blocking problem where a complex job request increases the delay for awaiting requests. The maximum number of job requests which can be concurrently served is referred to as *path limit*. We study the effect of path limit on scheduling delay and cluster utilization. Figure 4.3 compares scheduling delay for four different path limits. It shows that a path limit of three behaves poorly in terms of scheduling delay as compared to higher values. Results are particularly interesting for *single-path* scheduler with path limit of 1. About 45% of jobs were served with a very small delay. These jobs would have been the ones with very few number of tasks and have happened to arrive when scheduler was idle. However, rest of the jobs suffered head-of-the-line blocking problem resulting in high scheduling delay. Figure 4.4 shows that cluster utilization is low for single-path scheduler as compared to concurrent schedulers. Cluster utilization remains approximately the same as path limit goes from three to being unbounded. We conclude that a path limit of 100 is suitable of Google workload because it behaves almost like a job server with no upper bound on the number of concurrent job requests in terms of scheduler delay, while modestly increasing the cpu load (not shown here).

In the rest of this chapter, we configure our implementation of monolithic scheduler with a path limit of 100.

4.4 Scheduling Constraints

Apart from cpu and memory requirements, some tasks may specify additional scheduling constraints. For example, a task may need a machine with specific kernel version or a machine with GPU. These constraints may increase the complexity of scheduling algorithms. We studied the effect scheduling constraints on scheduler load (Figure 4.5). The scheduler load remains almost the same except for two spikes in case of scheduling constraints. Since only

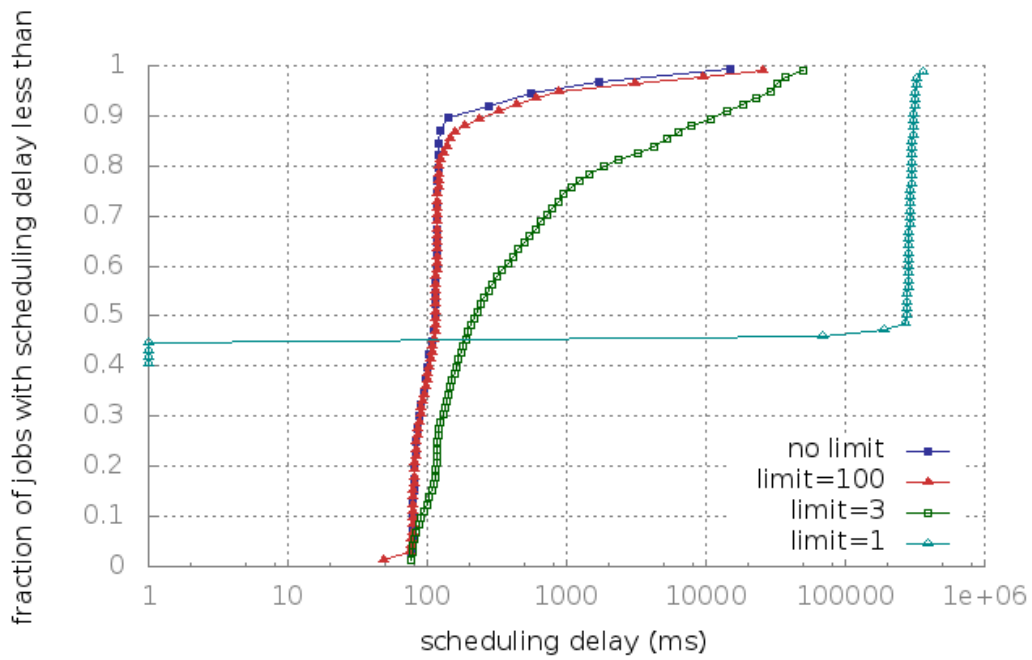


Figure 4.3: Cumulative distribution of job-wise scheduling delay for different path limits. *No limit* means that there is no maximum bound on the number of requests that can be served concurrently.

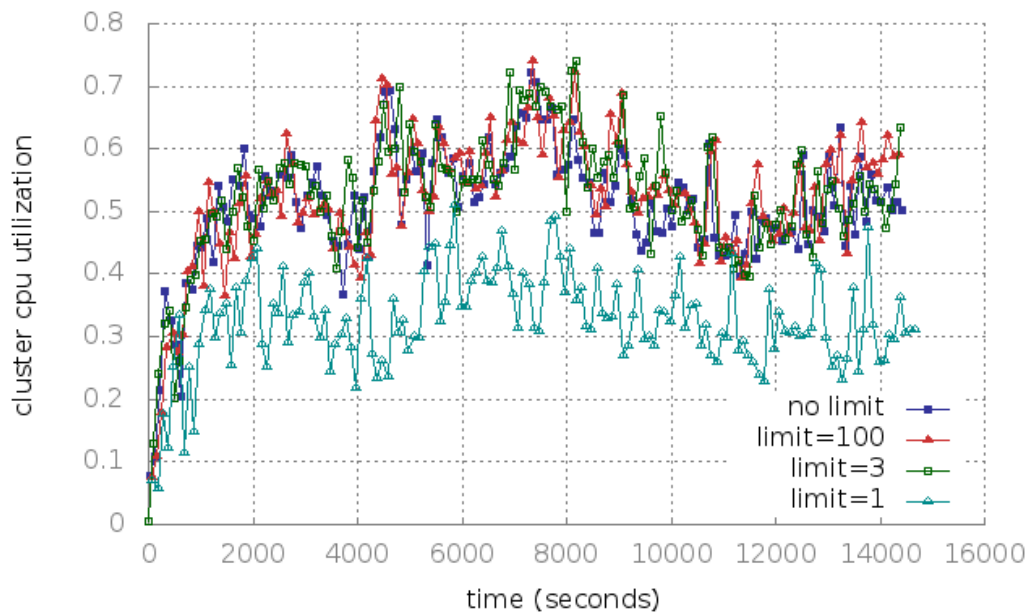


Figure 4.4: Cluster cpu utilization for different path limit values. *No limit* means that there is no maximum bound on the number of requests that can be served concurrently.

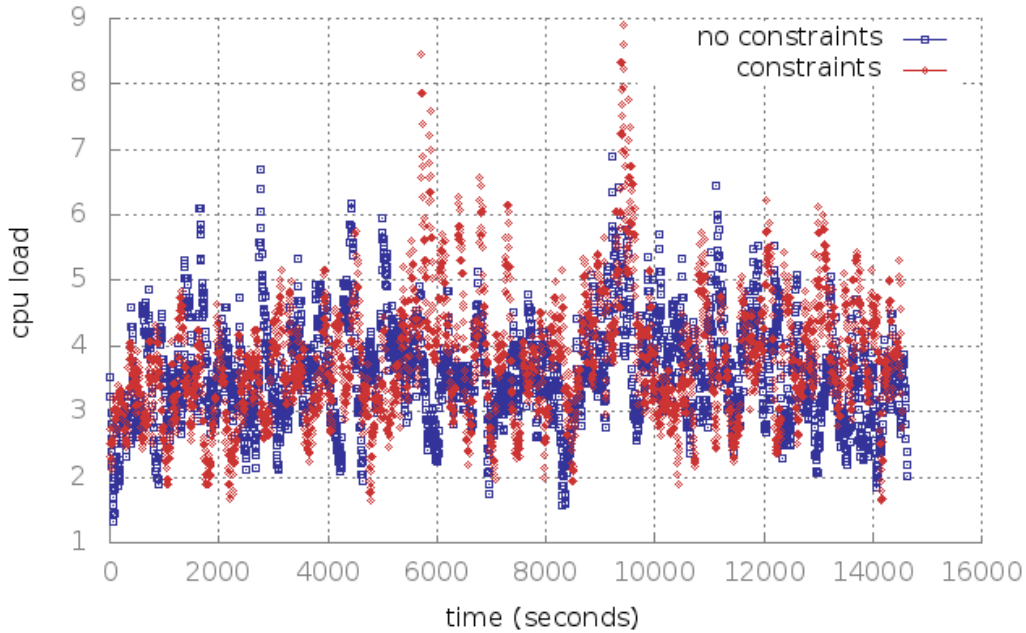


Figure 4.5: Effect of scheduling constraints on scheduler cpu load.

5% of tasks specify at least one constraint, their effect on scheduler load is not significant.

4.5 Scheduling Algorithm

Given a job, a scheduling algorithm calculates its schedule by assigning a node from cluster state to each task in the job. It may also give up and return *null* in case no such node is found. We experimented with three scheduling algorithms:

- *Random*: Scheduler selects a node at random from cluster state. If the node has enough resources available to run the given task, node is returned. Otherwise, it returns *null*.
- *Ten Tries*: This algorithm is like Random, except it tries ten random nodes before giving up. This algorithm is outlined in Algorithm 1

Table 4.2: Failure rate for different scheduling algorithms.

Algorithm	% failed jobs
Random	18.60
Ten Tries	5.23
Check All	11.14

- *Check All*: As a preprocessing step, a total ordering is specified on the cluster nodes. Given a task, scheduler iterates through the nodes starting at a random position. At every step of the iteration, it checks if enough resources are available on the current node to run the task, in which case the node is returned. If no such node is found, it gives up and returns *null*.

Table 4.2 shows the failure rate of each algorithm. As expected, Ten Tries performs significantly better than Random. However, it also performs better than Check All. This is because Ten Tries is more random in nature as compared to Check All and is able to beat the contention due to concurrent job requests. Figure 4.6 shows that Ten Tries and Check All have better cluster utilization than Random.

4.6 Components of Scheduling Delay

As stated before, a job contains one or more tasks. Since the distribution of number of tasks in jobs is highly skewed (Figure 1.1), the job and task delay distributions take up distinctly different shapes. Figure 4.7 compares the job and task delay distribution for our implementation of monolithic scheduler. Although 90% of jobs experienced a scheduling delay of less than 100 ms, only 20% tasks were able to run with a scheduling delay of 100 ms or less. Although start and end point of both the distributions is the same, they take up distinctly different shapes.

We divide the task-wise scheduling delay into two components: scheduler processing time and network delay. Figure 4.8 shows the distribution of these two components that make up the total task scheduling delay. Note that network delay includes the transmission delay as well time as the time spent in kernel network stack, making it the application level network delay.

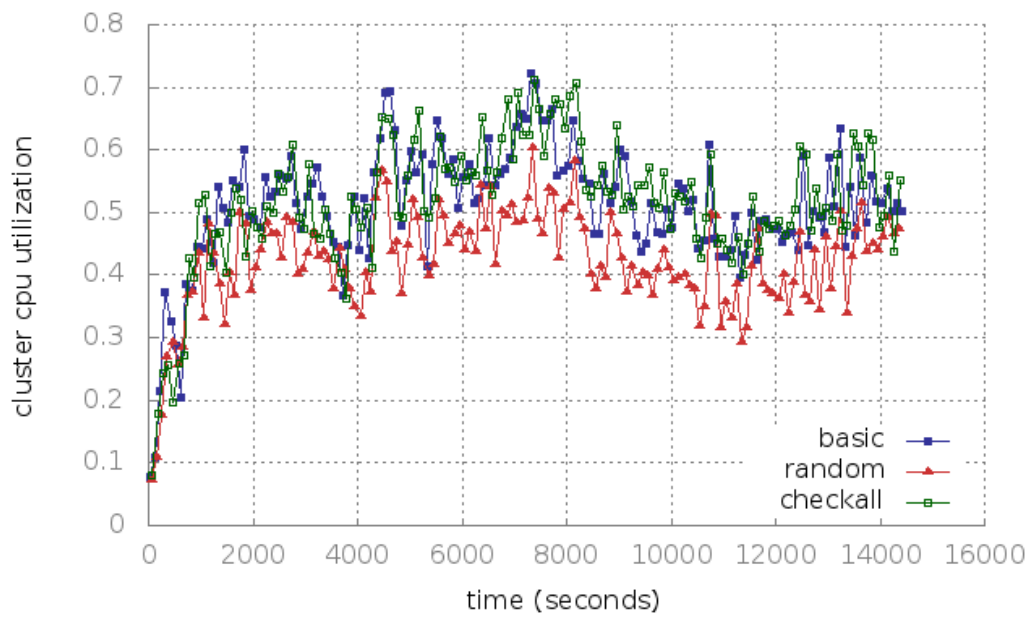


Figure 4.6: Cluster cpu utilization for different scheduling algorithms.

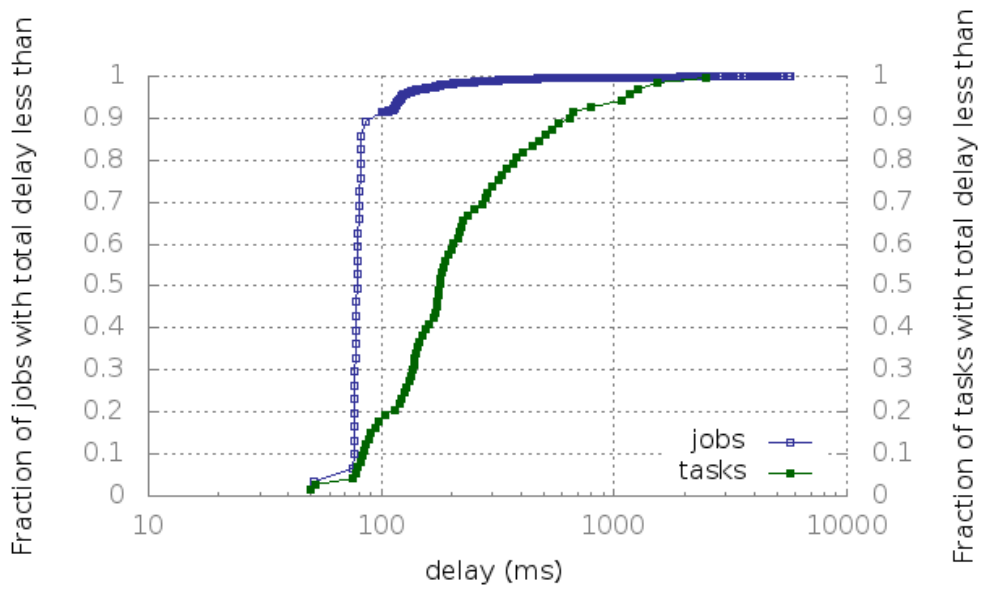


Figure 4.7: Relationship between job and task delay distribution for Google workload for our implementation of monolithic scheduler.

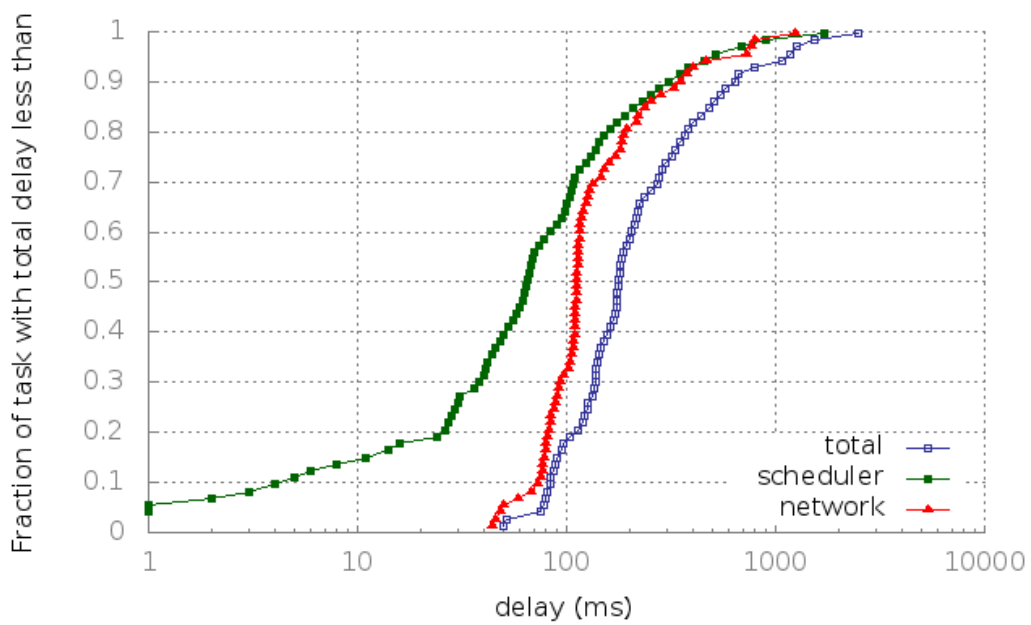


Figure 4.8: Distribution of network and scheduler delay that make up the total task delay.

Although the average RTT time between all machines involved in the experiment is less than 1 ms, observed application level network time is almost always greater than 10 ms. This shows that the majority of the network time is contributed by the kernel network stack. Moreover, the plot shows that network delay is almost always greater than scheduler processing time. This suggests that scheduling in monolithic architecture is a network I/O intensive task.

4.7 Verification of Cluster Emulation

In order to verify that the behaviour of scheduler in an emulated cluster is strongly correlated to the same in a real cluster, we conducted similar experiments as the experiments on emulated cluster described above, by running our implementation of monolithic scheduler in a real cluster of 38 nodes. Each of the 38 node in the real cluster consists of 2 quad-core Xeon E5620 2.4GHZ CPUs, which gives a total of 16 logical cores after enabling hyper-threading. Each node contains 64GB RAM, 512GB SSD, 4+.5 TB disk and are connected to each other with 1 gigabit per second Ethernet. All machines run Ubuntu 14.04.1 LTS with kernel version 3.13.0-34-generic.

We then compare all the metrics collected from experiments on real cluster against the metrics collected from experiments running on emulated cluster of size 38. Figure 4.9 and Figure 4.10 show the similarity of job and task delay distribution respectively for real and emulated clusters. This verifies that emulated clusters consisting of tens of nodes strongly represent the behaviour of real clusters of the same size.

We used the above result to verify emulation of clusters consisting of thousands of nodes. We emulated a 38 node cluster on each of the 38 node in the real cluster, thus, resulting in a cluster of size 1444 (38x38). We refer to it as a *hybrid* cluster, given that it is a real cluster of emulated clusters. Since 38 node emulation has been verified, we assume that this hybrid cluster approximately represents a real cluster of size 1444. We ran our implementation of monolithic scheduler on this hybrid cluster and compared the results to the results collected from an emulated cluster of size 1444. Figure 4.9 and Figure 4.10 also show the similarity of job and task delay distribution respectively for these hybrid and emulated clusters of size 1444. Although the

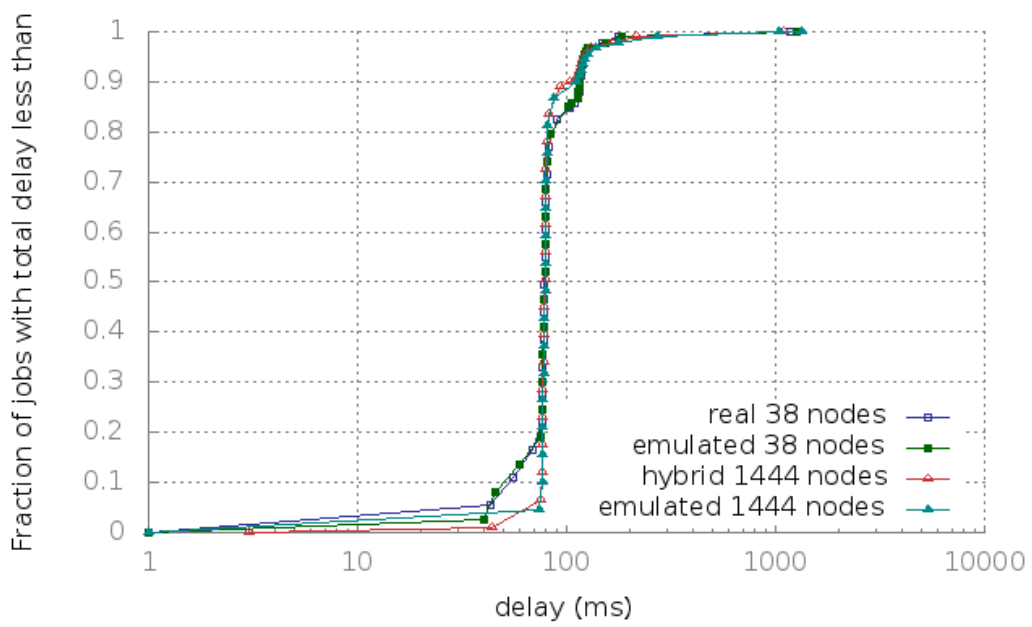


Figure 4.9: Job delay distribution for emulated and real clusters of small and large size.

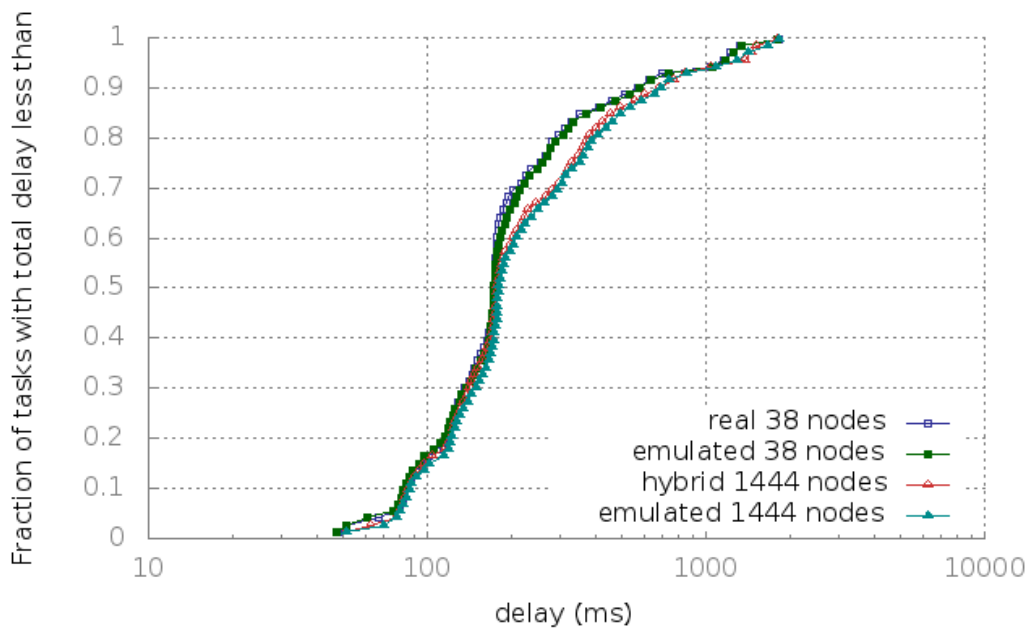


Figure 4.10: Task delay distribution for emulated and real clusters of small and large size.

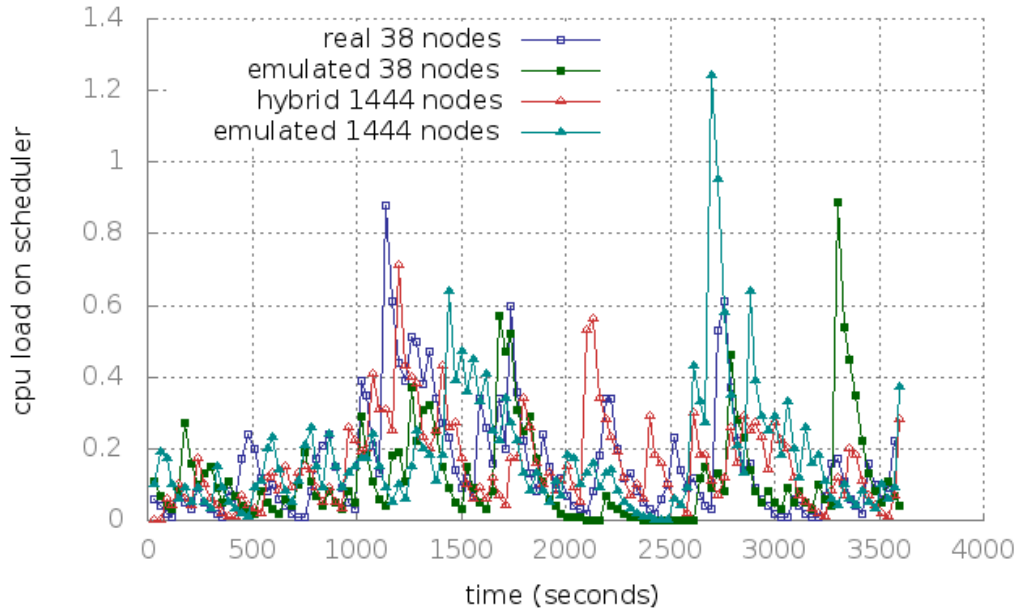


Figure 4.11: Variation of scheduler cpu load with time for real and emulated clusters of small and large size.

results from cluster of different sizes (38 and 1444 nodes) differ, the results from real and emulated clusters of the same size show striking similarities. This verifies that emulated clusters consisting of thousands of nodes strongly represent the behaviour of real clusters of the same size. Due to the lack of a real cluster of thousands of nodes, we took the recursive approach of 'hybrid' clusters to verify the emulation of bigger clusters.

Figure 4.11 compares the cpu load of scheduler for real and emulated clusters of small and large sizes, which remains approximately the same for all cases, except for a couple of spikes in case of emulated clusters. Figure 4.12 shows the cumulative number of failed requests for real and emulated clusters of different sizes. Although number of failures increases for larger clusters, the results remain approximately the same for real and emulated clusters of the same size.

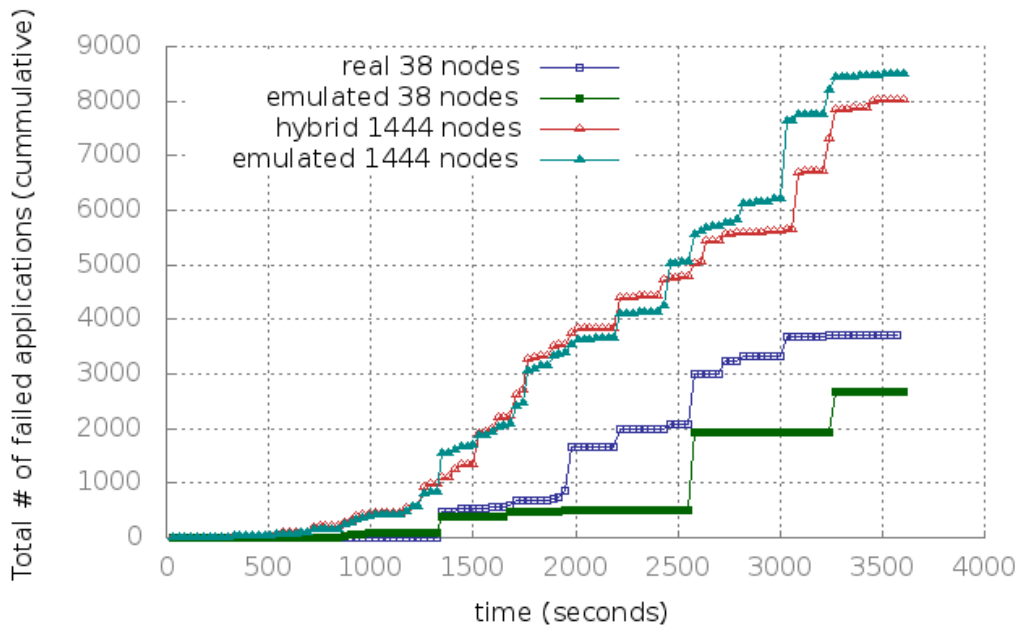


Figure 4.12: Variation of number of failed requests (cumulative) with time for emulated and real clusters of small and large size.

CHAPTER 5

EVALUATION OF HADOOP YARN SCHEDULERS

Hadoop YARN [11] is one of the most popular cluster management framework, which is available as an open source project under Apache License 2.0. Cloudera [18] and Hortonworks [19] are two of the major firms providing support and services for Hadoop YARN. Major open source distributed computation frameworks such as Spark [20] and Storm [3] provide support for running on clusters managed by Hadoop YARN. Yahoo! is one of the prominent user of Hadoop YARN [11].

Job scheduling is one of the major task in cluster management. The latest version of Hadoop YARN contains three pluggable schedulers namely, Capacity scheduler [21], Fair scheduler [22] and Fifo scheduler [23]. We evaluated all three schedulers under workload generated by replaying Google traces. We measured various components of scheduler delay and cluster resource usage. The rest of this chapter is organized as follows: In Section 5.1, we briefly describe the design and architecture of Hadoop YARN, followed by a discussion on the experimental set up in Section 5.2. In Section 5.3, we present and discuss the experimental results. For the rest of this chapter, we would use YARN and Hadoop YARN interchangeably.

5.1 Overview of Hadoop YARN Architecture

Figure 5.1 represents the interaction between different components in YARN. Hadoop YARN consists of a cluster-wide component called *Resource Manager* (RM), which runs as a daemon on a dedicated machine. RM tracks cluster resource usage and node liveness. Such a tracking is made possible with the help of per-node daemon, called *Node Manager* (NM). A NM daemon runs on each machine in the cluster and sends periodic heartbeats to RM, mainly consisting of resource usage and health status of the node.

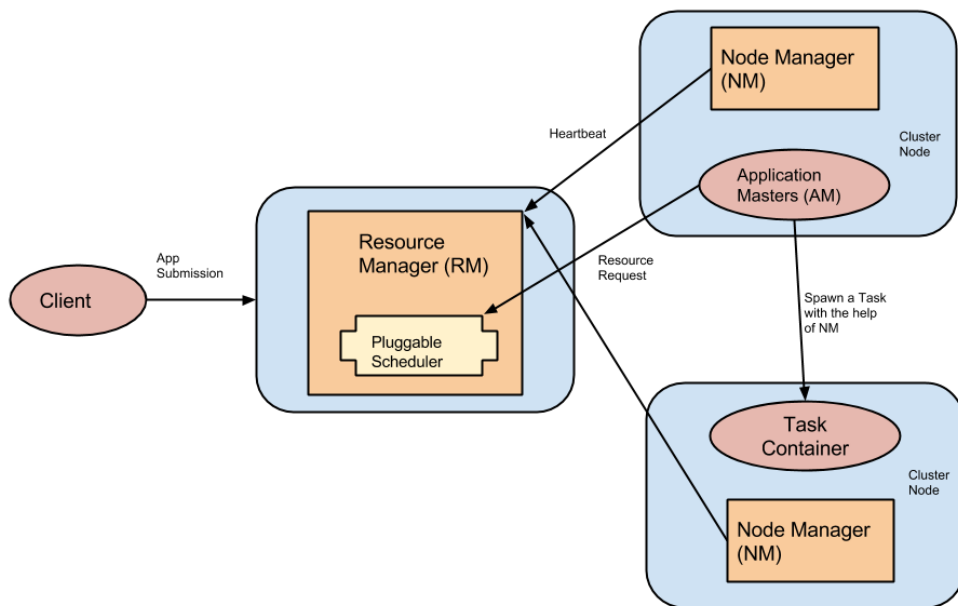


Figure 5.1: Hadoop YARN Architecture.

RM accepts applications via a public submission protocol. The submission contains required resources and commands to run a per-application process called *Application Master* (AM), which itself runs on one of the cluster node. AM sends periodic heartbeats to RM to ensure liveness, which consists of resource requests to run tasks. RM responds to a resource request by granting 'container' lease, which is a logical bundle of resources (cpu and memory) on a particular node. AM can use the container to run a task with the help of NM running on the node on which the container is granted.

AM logic could be as simple as running a set of tasks by requesting containers from RM. However, AM could contain more complex logic to run a DAG of jobs where the execution of tasks depend on each other. Although RM provides task monitoring interfaces, the responsibility of tracking task execution and fault tolerance is delegated to AM.

5.1.1 YARN Schedulers

A global view of cluster state enables RM to maintain allocation invariants and arbitrate resource contention between jobs. RM allows for a pluggable scheduling policy for resource allocation. YARN official release comes with three default schedulers as follows:

- *Fifo scheduler*: It maintains a queue of allocation requests and serves them in the order of submission. It does not offer any allocation invariant and it's primary merit is simplicity.
- *Capacity scheduler*: It is suitable for multi-tenant clusters, where two or more organizations share the cluster. The scheduler allows for the creation of per organization 'queue' with specific fraction of cluster resources. The sum of fraction of all queues should be equal to one. It guarantees that a queue will be provided with its share of resources if not more. However, a queue can be provided with more resources than its capacity in case other queues are running low on demand.
- *Fair scheduler*: It aims to ensure that all running applications, on average, get an equal share of resources over time. It helps overcome head-of-the-line blocking problem where short jobs wait a for long job to be finished. Like Capacity scheduler, it also supports the notion

of queues to fairly divide resources between entities in a specified proportion. It also supports flexible scheduling policies within different queues.

5.2 Experimental Set-up

We thoroughly analyze the performance of YARN schedulers experimentally, with default settings. We ran experiments on Hadoop YARN version 2.6 [24], which is the latest stable release of Hadoop YARN during the writing of this report. We conducted all experiments on a cluster of 22 HP Proliant DL160 G6 nodes. Each node consists of 2 quad-core Xeon E5620 2.4GHZ CPUs, which gives 16 logical cores after enabling hyperthreading. Each node contains 64GB RAM, 512GB SSD, 4+.5 TB disk and are connected to each other with 1 gigabit per second Ethernet. All machines run Ubuntu 14.04.1 LTS with kernel version 3.13.0-34-generic. We used a dedicated Dell 320 machine to run RM. This machine is superior in configuration to cluster nodes. It contains four quad core processors, giving a total of 32 logical cores after enabling hyper-threading. It contains 128GB RAM, 64GB SSD and 512GB of storage. It runs Ubuntu distribution with kernel version 3.5.0-43-generic.

We run one YARN application for each job in Google traces (Section 3.1). For the rest of this chapter, we would use 'job' and 'application' interchangeably. We used Workload Generator from our testbed (Section 3.1) for submitting applications to RM. It runs on a machine with the same configuration as the one running RM. It replays Google traces and runs one client thread per job to submit corresponding application to RM. The Application Master is provided with the number of tasks in the job for which it requests resources from RM. Each task sleeps for a specified duration (according to the duration of task in the trace) and terminates. All tasks are run with the same priority and do not specify any locality / machine constraints. When all the tasks are complete, AM sends the measured delays to Workload Generator and terminates. To summarize, we use three components from the traces: job inter-arrival time, number of tasks in the jobs and duration of each task. Since the cpu and memory data in the trace is present in anonymized form, we run each task with 1 vcore and 100 MB of memory.

Since our goal is to analyze the performance of schedulers, we disable security by disabling Kerberos authentication. Besides, all executables are placed on all nodes before starting the experiment so as to eliminate delays due to copying files over network. Thus, the measured delay can be seen as the scheduling overhead. All schedulers are configured with default settings consisting of only one queue with 100% capacity. All applications are submitted under a single user name.

5.3 Experimental Results

For all experiments, we replayed workload from Google traces for a duration of 1 hour (3600 seconds), unless otherwise stated. To facilitate shorter experiment durations while covering a major portion of trace, we replayed the traces with a speed of 5x for all experiments, unless otherwise stated. We study the effect of this speed in Section 5.3.1.

With such a setting, each experiment generates 3,654 jobs consisting of 116,291 tasks. In order to run a workload of such magnitude over a relatively smaller cluster consisting of 22 nodes, we reduced the duration of all tasks by a factor of 100. This ensures that cluster contains enough resources to run the arriving tasks and enable us to study the performance of scheduler with smaller clusters. We study the effect of reduced durations of tasks in Section 5.3.2.

Each experiment was ran twice and as expected, the results from the two runs were highly correlated for all collected metrics, for all three schedulers. In this section, we report the results from the first run of each experiment. We measured various components of scheduling delay and cluster cpu usage, as follows.

AM delay refers to the time it took for scheduler to start the Application Master for a given job. It represents the difference between time at which AM started execution and the time at which application was submitted. Figure 5.2 shows the cumulative distribution of AM delay for the three schedulers. Fair and Fifo scheduler start AM within 1 second for 90% of the jobs, while Capacity scheduler takes more than 1000 seconds for 40% of the jobs. Due to the ability of avoiding head-of-the-line blocking problem, Fair scheduler performs significantly better than Capacity scheduler, resulting in

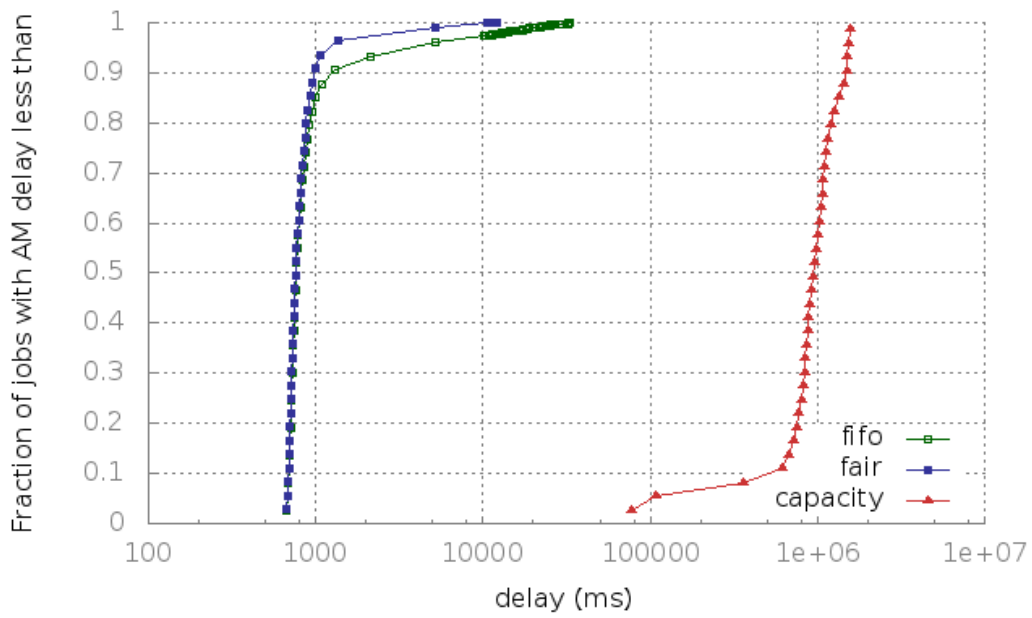


Figure 5.2: Cumulative distribution of AM delay for Capacity, Fair and Fifo Scheduler. This is a job-wise delay distribution.

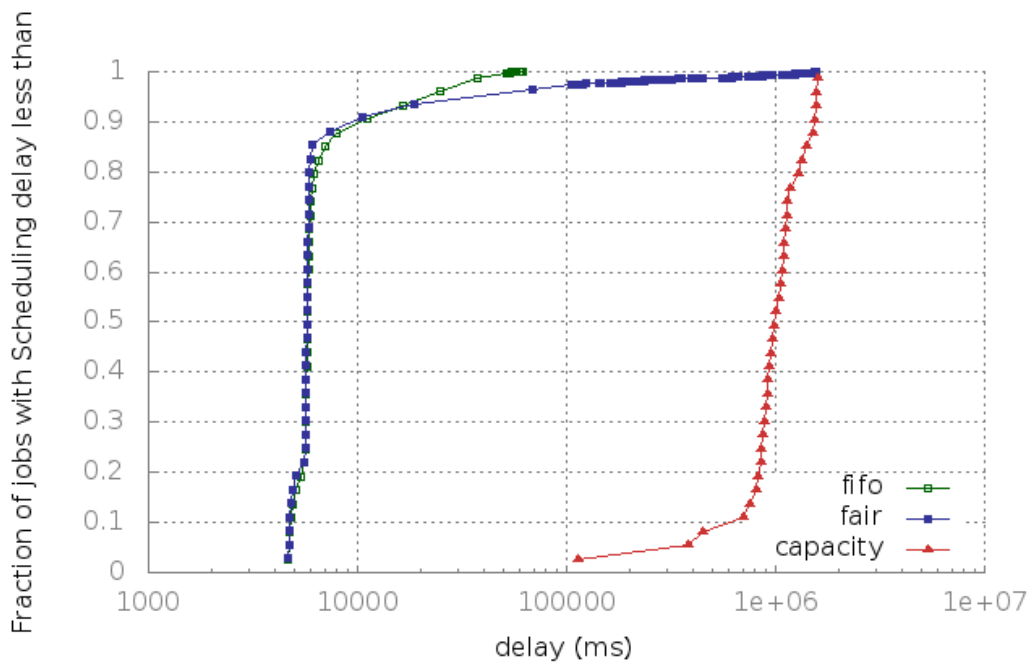


Figure 5.3: Cumulative distribution of total Scheduling delay for Capacity, Fair and Fifo Scheduler. This is a job-wise delay distribution.

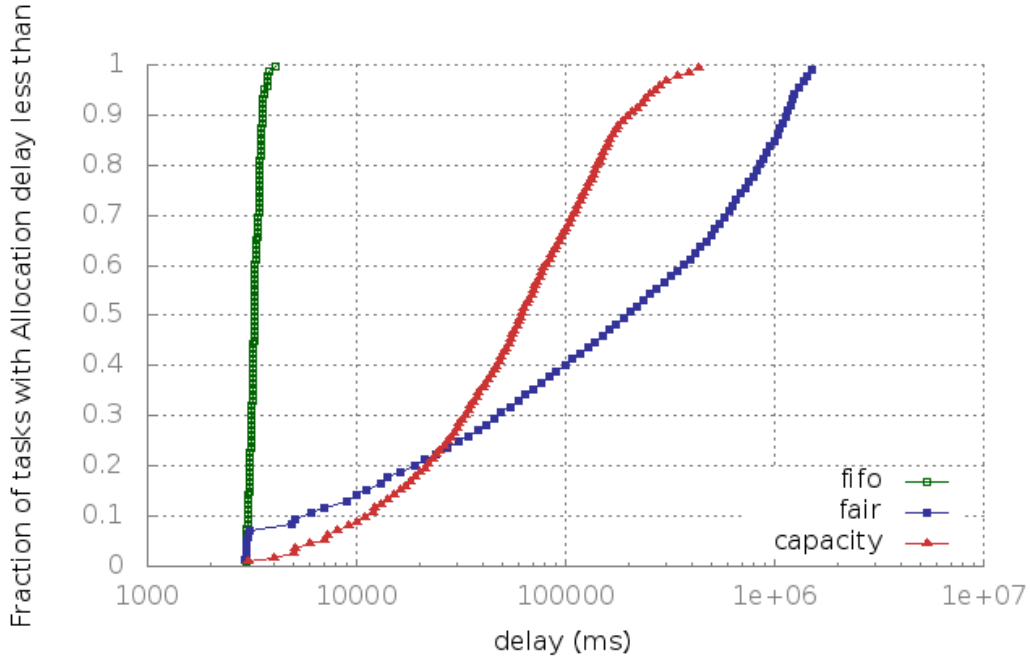


Figure 5.4: Cumulative distribution of Allocation delay for Capacity, Fair and Fifo Scheduler. This is a task-wise delay distribution.

a performance gap of over two order of magnitudes. Figure 5.3 shows the cumulative distribution of total *scheduling delay* of the jobs, which refers to the sum of AM delay and the delay resulting from interactions between AM and RM to run all the tasks. Please note that this delay does not include the running duration of tasks. For Fifo and Capacity scheduler, this distribution closely resembles the distribution of AM delay. This suggests that AM delay is the major contributor of scheduling delay for these two schedulers. However, in case of Fair scheduler, AM-RM interactions give rise to a long tail in scheduling delay distribution. We speculate that the fair sharing of resources results in longer running time for complex jobs (containing large number of tasks). Since a relatively smaller fraction of jobs are complex, such a scheduling policy gives rise to a longer tail in total scheduling delay.

As shown in Figure 1.1, the number of tasks in a job follows a skewed distribution i.e. a small number of jobs make up the majority of tasks. Due to such a relationship, task-wise delay distributions take up distinctly differ-

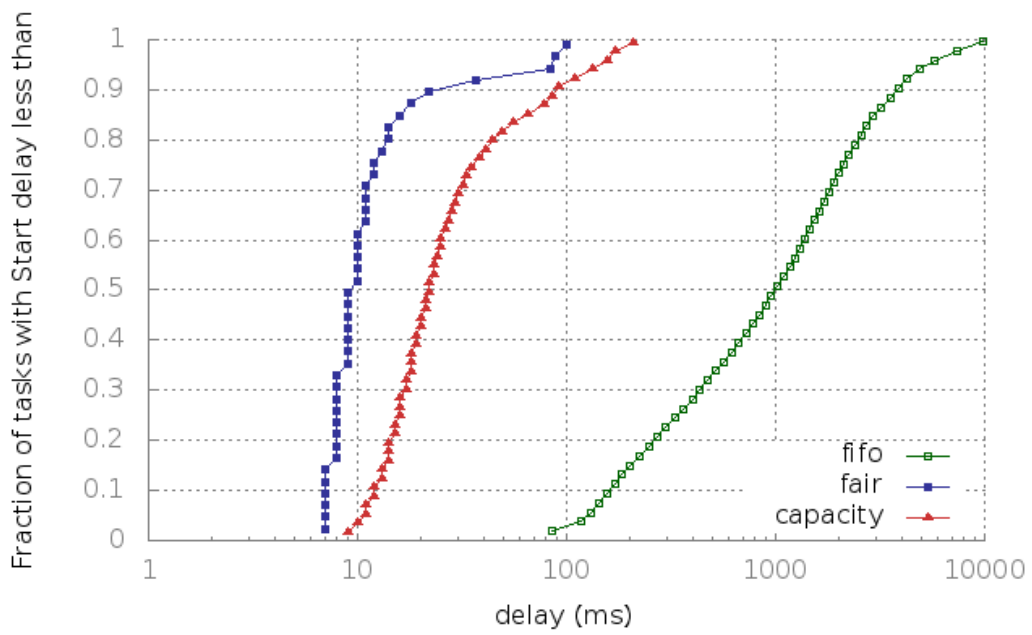


Figure 5.5: Cumulative distribution of Task-Start delay for Capacity, Fair and Fifo Scheduler. This is a task-wise delay distribution.

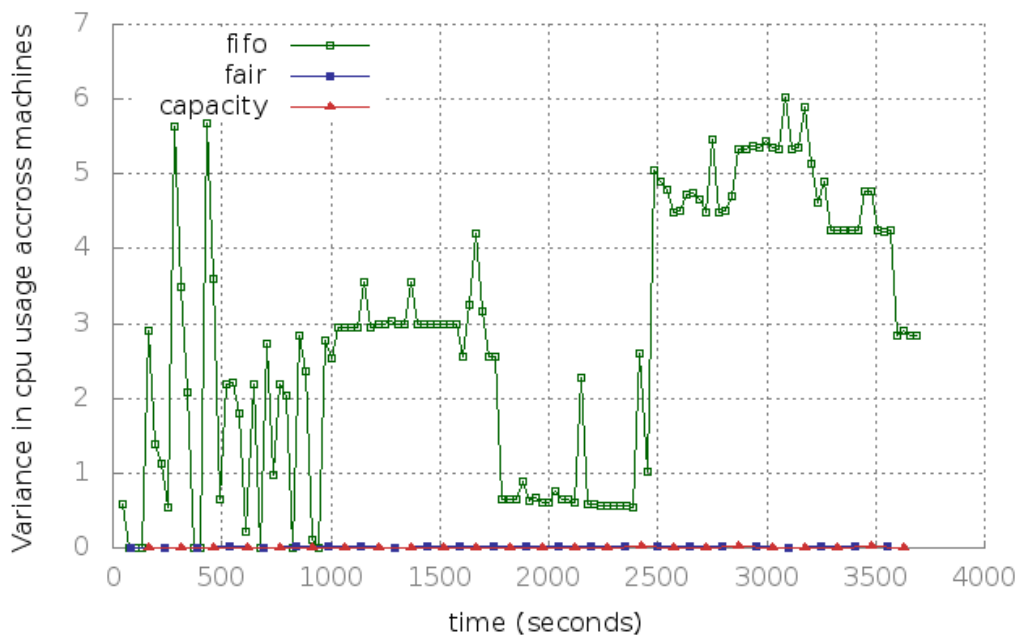


Figure 5.6: Variation of standard deviation of cpu usage across nodes with time, for YARN schedulers.

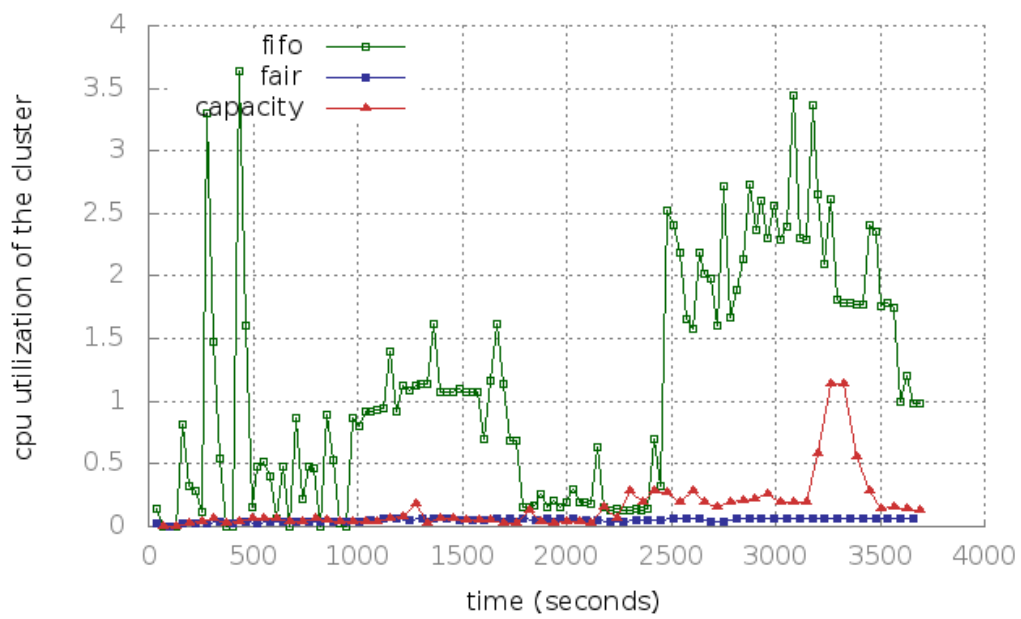


Figure 5.7: Variation of total cpu utilization of cluster with time, for YARN schedulers.

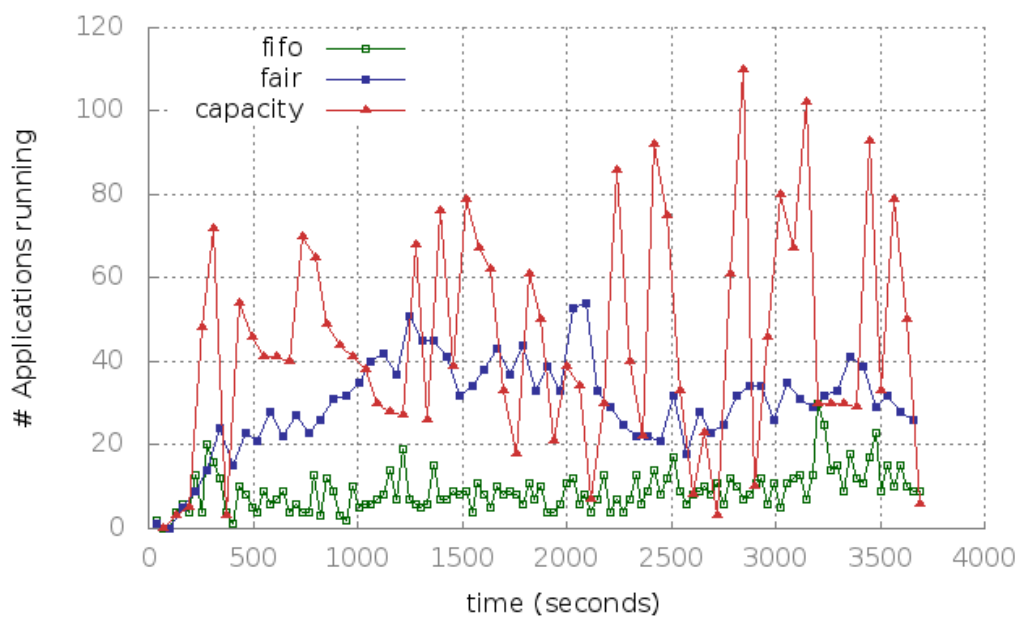


Figure 5.8: Variation of number of running applications with time, for YARN schedulers.

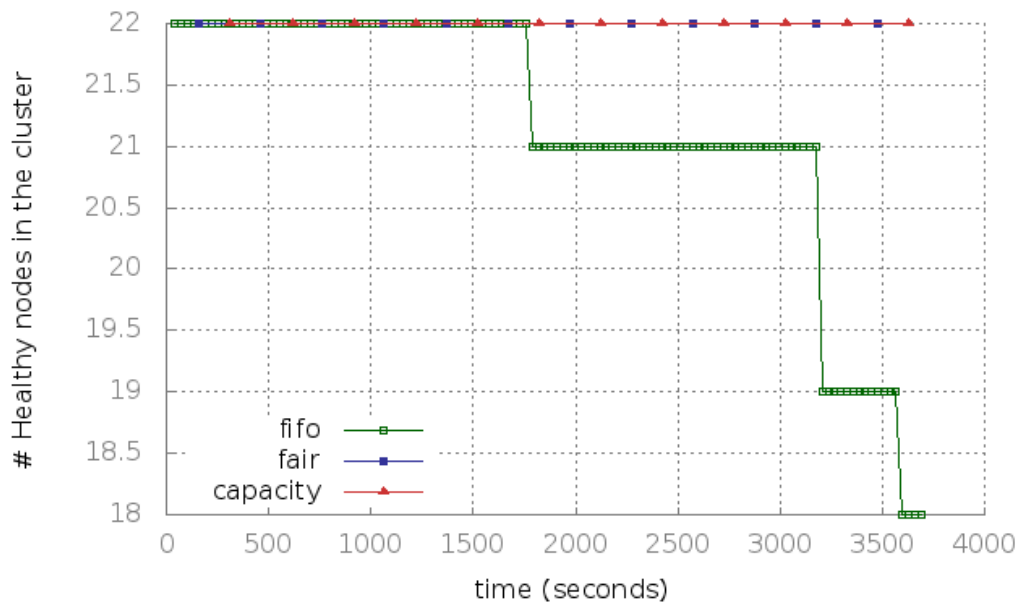


Figure 5.9: Variation of number of healthy nodes in the cluster with time, for YARN schedulers.

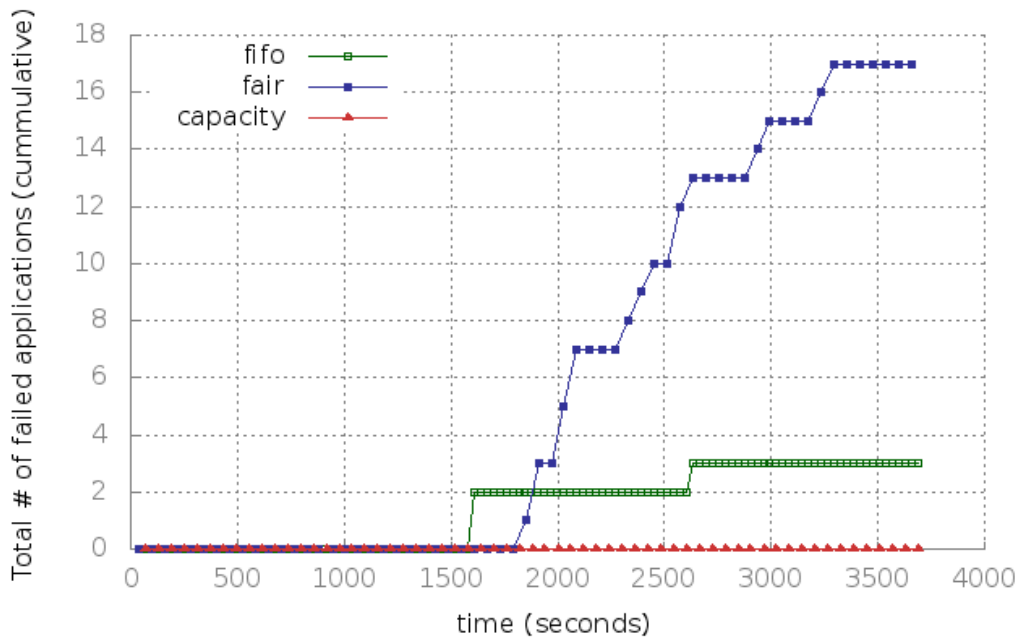


Figure 5.10: Variation of cumulative number of failed applications with time, for YARN schedulers.

ent shapes than job-wise delay distributions. We define *allocation delay* of a task as the time taken for allocating a container to the task on a cluster node after Application Master has been started. Figure 5.4 shows the distribution of allocation delay of all tasks. As expected, Fifo scheduler offers similar allocation delay to all the tasks due to its fifo nature of handling resource requests. However, Capacity and Fair scheduler show a wide variation of allocation delays because they are more concerned with job-wise allocation. Fair scheduler offers minimal delay to 10% of the tasks, which allows it to perform efficiently for 90% of the jobs. Apart from variations in allocation delay, we also note that Fifo scheduler performs marginally better than its peers. However, this performance gain comes at the cost of making poor allocation decisions. This can be seen in Figure 5.6 which shows the variance of cpu usage across cluster machines. The figure depicts that container allocation in Fifo scheduler is severely uneven, resulting in unbalanced load across cluster machines, while Capacity and Fair scheduler successfully balance the cpu load among cluster machines.

One of the effects of container allocation decisions is reflected in *task-start delay*, which is defined as the time it takes for a task to start execution after the container has been allocated. Figure 5.5 shows the distribution of task-start delay for all tasks. Due to the unbalanced allocation in case of Fifo scheduler, the NM on overloaded machines takes significantly longer time to start the tasks, as compared to the same in Capacity and Fair scheduler. Moreover, Fifo scheduling resulted in the failure of four nodes during the course of experiment, which can be seen in Figure 5.9. Since there were no hardware failures, we speculate that these nodes became unresponsive due to the large number of containers allocated on them by Fifo scheduler. As expected, there is no node failure in case of Capacity and Fair scheduler.

Figure 5.7 shows the total cpu utilization of the cluster, which fluctuates substantially for Fifo scheduler, while remains stable for the other two, especially Fair scheduler. Moreover, Fifo scheduler overloads the cluster resources, driving cpu utilization to go higher than 1 for a significant periods of time. On the other side, the other two schedulers do not overload the cluster.

Figure 5.8 shows the number of running applications in the cluster with time, which fluctuates substantially for Capacity scheduler, while remains stable for the other two schedulers. Besides, Capacity and Fair scheduler

keeps the applications alive for a longer period of time by making them wait for resources. On the other hand, Fifo scheduler assign resources to applications as soon as they arrive without any cap limits.

From the above results, it seems that Fair Scheduler outperforms it's peers in terms of delay and allocation decisions. However, Capacity scheduler beats it's counterparts in terms of application failures, as shown in Figure 5.10. While Capacity scheduler doesn't drop any application, 17 out of 3,654 jobs fail in case of Fair scheduler, resulting in a loss of 29,885 tasks out of a total of 112,732 tasks.

5.3.1 Effect of Trace Speed

To facilitate shorter experiment durations while covering a major portion of trace, we replayed the traces with a speed of 5x for all experiments discussed above. We refer to this speed of replaying trace as *trace speed*. To study the effect of trace speed on schedulers, we ran three experiments (one for each scheduler) with a trace speed of 1x for a duration of five times the duration of above experiments (5 hours), so as to maintain the same workload.

In case of Capacity scheduler, we observe a huge improvement in scheduling delay. Figure 5.11 shows the sensitivity of Capacity scheduler towards trace speed, in terms of scheduling delay. Unlike Capacity scheduler, Fifo scheduler shows little improvement in scheduling delay for slower experiment (Figure 5.12). In case of Fair scheduler, the improvement in performance is reflected in terms of the number of failed applications. Figure 5.14 shows that no application failed in the experiment with slow trace speed for Fair scheduler. Interestingly, in order to be able to run all applications, Fair scheduler compromised a little on the scheduling delay, as shown in Figure 5.13.

Apart from job delays, number of running applications decreased substantially for all three schedulers, in case of experiments with slower trace speed.

5.3.2 Effect of Task Duration

Each experiment generates 3,654 jobs consisting of 116,291 tasks. In order to run a workload of such magnitude over a relatively smaller cluster consisting of 22 nodes, we reduced the duration of all tasks by a factor of 100 for the

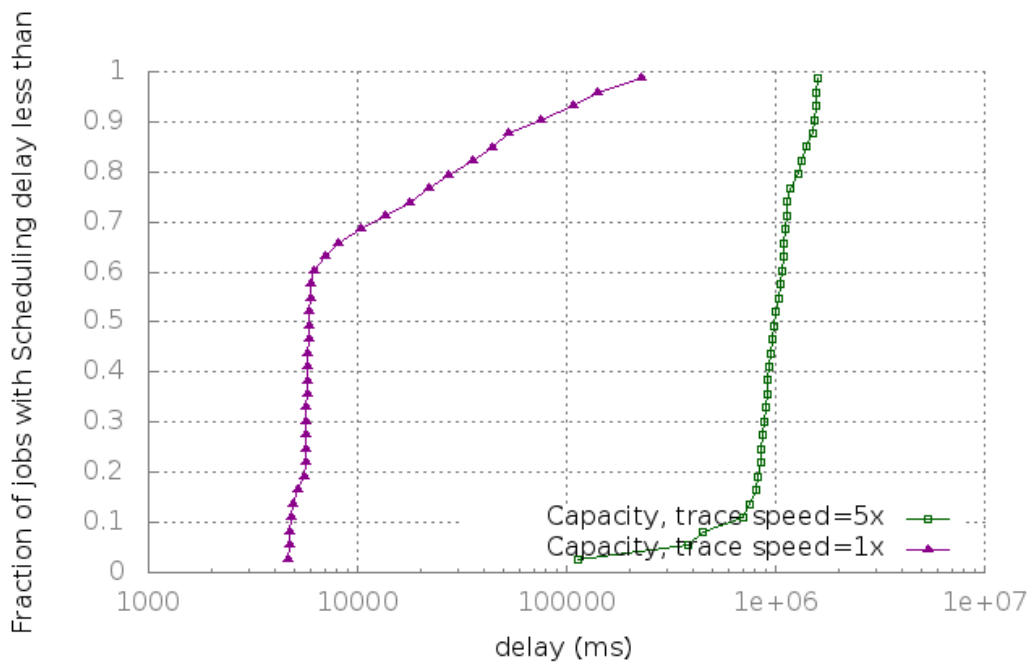


Figure 5.11: Variation of scheduling delay with trace speed for Capacity scheduler.

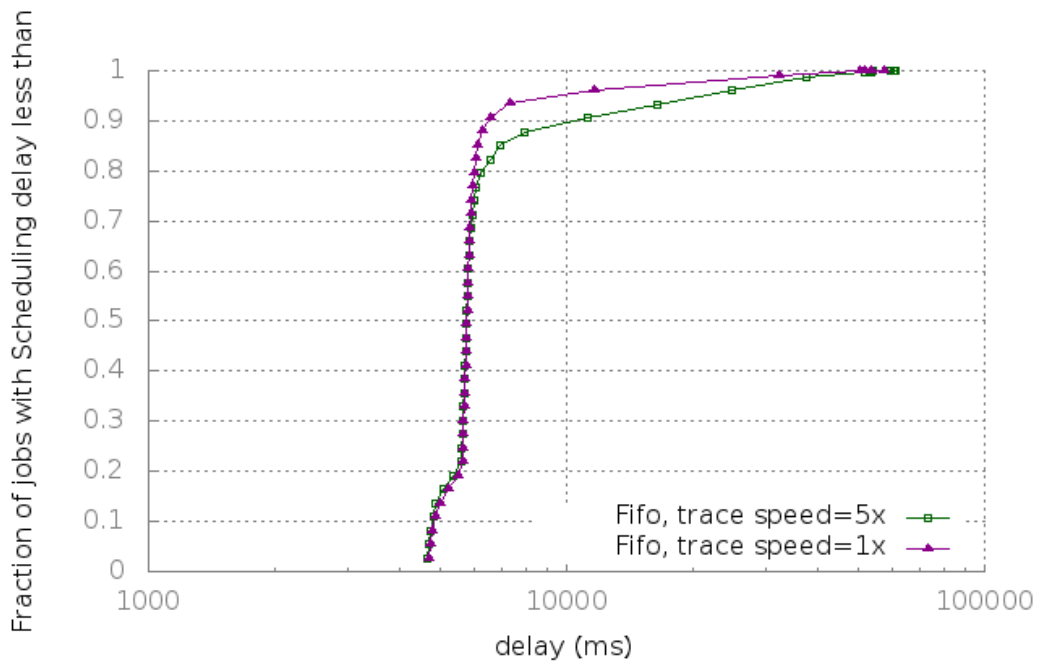


Figure 5.12: Variation of scheduling delay with trace speed for Fifo scheduler.

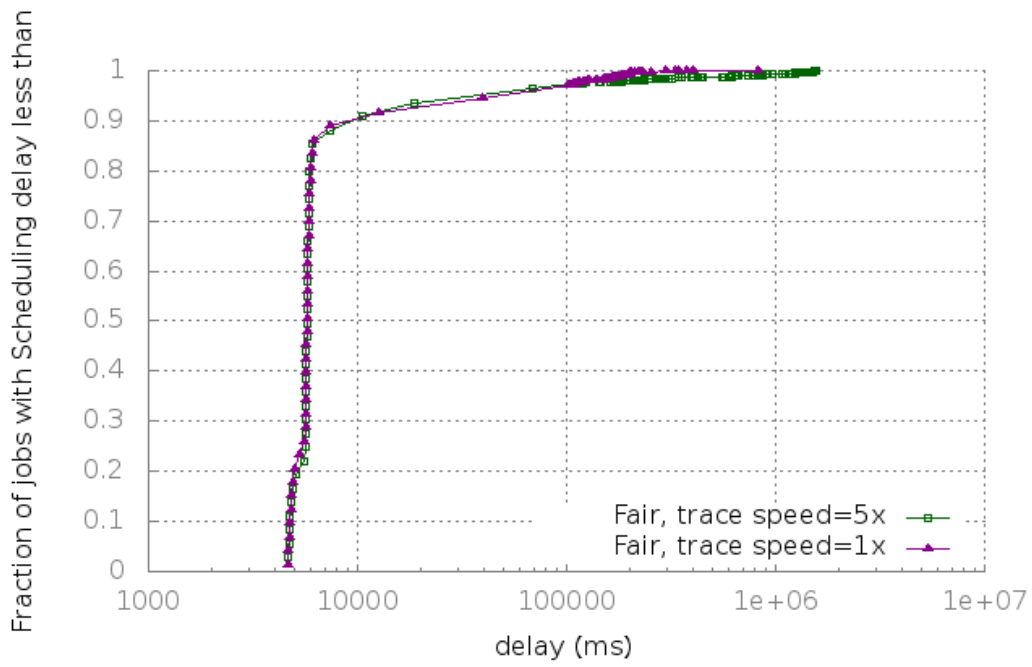


Figure 5.13: Effect of trace speed on scheduling delay for Fair scheduler.

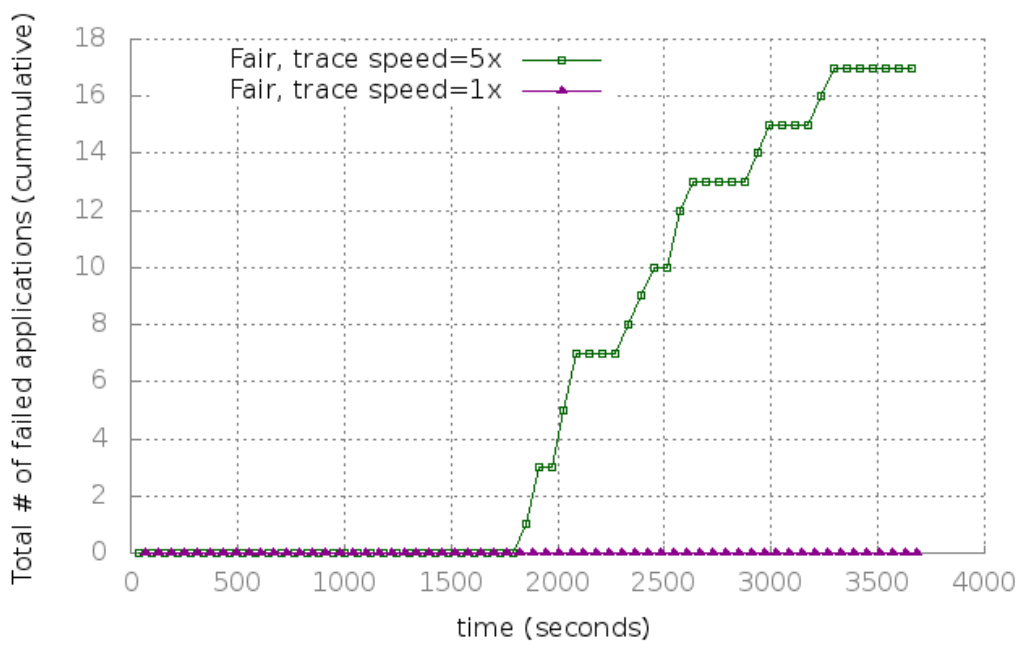


Figure 5.14: Effect of trace speed on application failures for Fair scheduler.

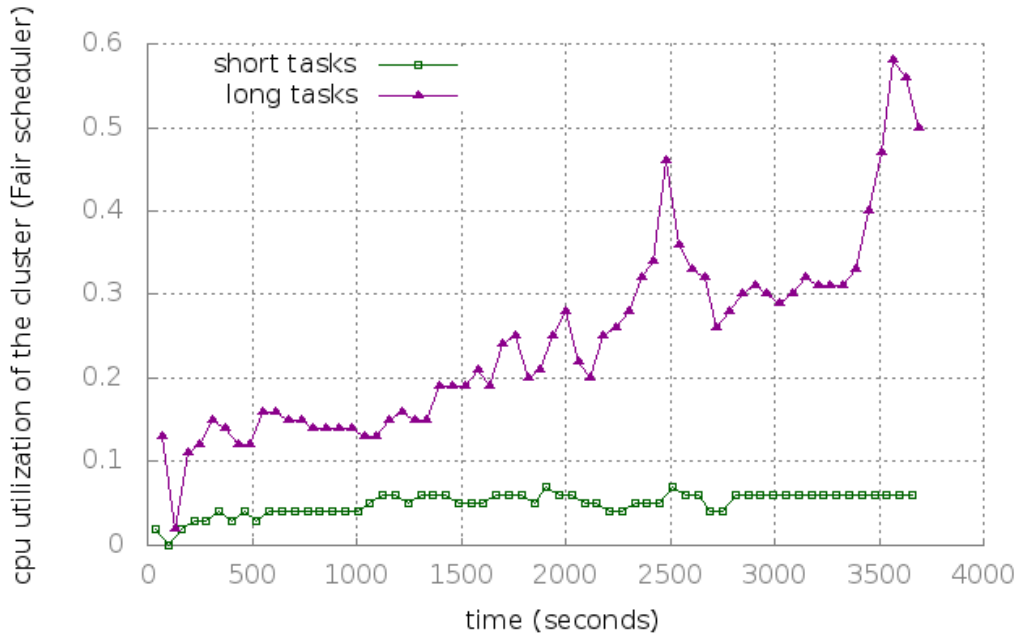


Figure 5.15: Effect of task duration on cluster cpu utilization for Fair scheduler.

above experiments. This ensures that the cluster contains enough resources to run the arriving tasks and enable us to study the performance of schedulers with smaller clusters.

We conducted experiments where the task durations were reduced by 10 instead of 100 to study the effect of task duration on scheduler performance. As expected, cpu utilization and running application count increased for all the three schedulers. Figure 5.15 and Figure 5.16 compare the cpu utilization and running application count respectively, for different durations of tasks for Fair scheduler. Figures for Capacity and Fifo schedulers are not shown since they depict similar trends.

However, in case of Fifo Scheduler, as shown in Figure 5.17, the unbalanced placement of tasks on cluster nodes resulted in 15 node failures, as compared to 4 in case of shorter tasks. This suggests that Fifo scheduler is unsuitable for enterprise clusters.

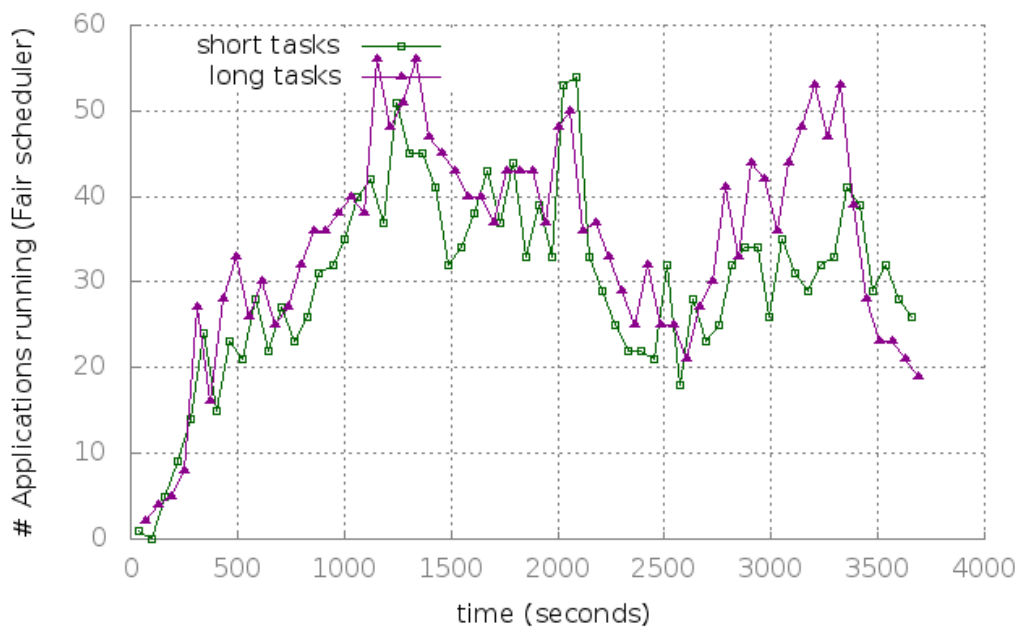


Figure 5.16: Effect of task duration on running application count for Fair scheduler.

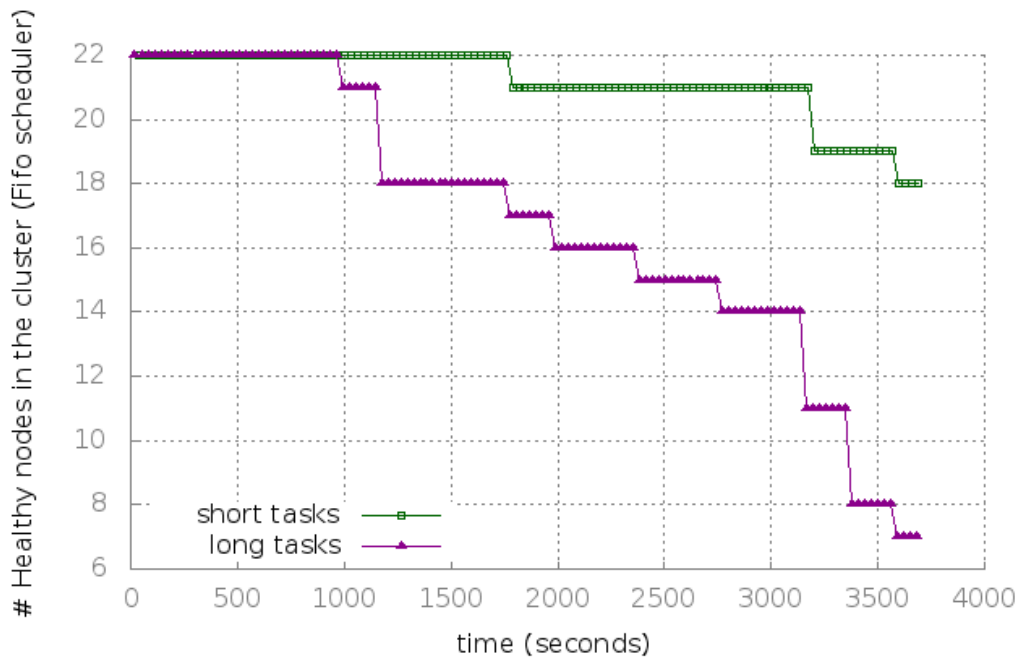


Figure 5.17: Effect of task duration on node failures for Fifo scheduler.

CHAPTER 6

RELATED WORK

In this chapter, we present a brief survey of the research projects on cluster schedulers. We also briefly discuss the past projects on analysis of cluster scheduling traces, including the traces from clusters at Google, Facebook and Cloudera.

6.1 Cluster Schedulers

In his PhD dissertation, Konwinski [15] provides a taxonomy of scheduler architectures. Broadly speaking, the author classifies scheduler architectures into single-agent (monolithic) and multi-agent (Figure 2.1). We provide an overview of the taxonomy in Section 2.2. We extend the classification to include the decentralized architecture as a type of multi-agent scheduler. In decentralized architecture, cluster machines maintain a queue of tasks waiting to be executed. Although scheduling-agents share the cluster machines, they do not synchronize with each other, but instead query the cluster machines for the length of the task queues to make intelligent scheduling decisions.

Hadoop YARN [11] can be classified as a monolithic scheduler, where a single scheduling-agent, called Resource Manager (RM), receives all the task scheduling requests. However, instead of receiving a job request with all the tasks, YARN supports more powerful and expressive semantics for receiving scheduling requests. A job request is issued by a long lived process called Application Master (AM), which can request and negotiate cluster resources with the RM. This allows for powerful scheduling semantics (such as resource hoarding, enforcing an order on task execution) at the expense of scheduling delay. The paper compares the performance of YARN with older Hadoop version and provides statistics from 2500 node cluster at Yahoo! Although YARN provides three pluggable scheduling policies: Capacity, Fair and Fifo

schedulers, all results in the paper are reported on the Capacity scheduler. We evaluate the three schedulers and provide a thorough comparison of their trade-offs in Chapter 5.

Ousterhout et al. [13] propose Sparrow, a decentralized multi-agent scheduling framework suitable for short-duration jobs, based on random sampling of worker nodes. When a job arrives to a Sparrow scheduler node, it selects twice the number of worker nodes as there are tasks in the job and queries them for the length of their task queues. The tasks are sent to the best nodes (lightly loaded) from this sample. Given the concurrent nature of job requests, we also observe the benefits of randomness in scheduling algorithms, as described in Section 4.5. Since Sparrow’s scheduling algorithm relies extensively on random sampling, it works well if there are more idle workers present in the cluster because the likelihood of querying an idle machine is high. The performance degrades as cluster load increases (and simulation results in the paper are consistent with this observation). The evaluation presented in the paper is restricted to short duration tasks and the comparison is restricted to only Spark [25] scheduler.

Mesos [10][15] can be classified as a partitioned multi-agent scheduler, where a centralized module dynamically partitions and distributes cluster resources between application specific scheduling agents. The distribution of resources is carried out in terms of resource offers from central module to scheduling agents, which may be accepted or rejected by the latter. However, experimental comparison is limited to static partitioning of the cluster. Schwarzkopf et al. [12] argues that such an architecture may lead to low cluster utilization due to hard partitioning of resources.

In contrast to partitioned multi-agent scheduler architecture, Schwarzkopf et al. [12] introduce shared state multi-agent scheduler architecture where scheduling agents have access to all cluster resources. In order to claim a resource, an agent needs to update a resilient central copy of cluster state in an atomic transaction. In case of conflicting transactions when two or more agents are trying to claim the same resource, only one of the transaction succeeds. Google Omega is an implementation of shared state architecture. Although, such an optimistic concurrency control provides the flexibility to run complex scheduling algorithms for picky tasks, it may result in scheduling delay, contention and starvation in case of high transaction failure rate. A transaction may contain more than one resource request. In such cases,

conflict rate depends on conflict resolution schemes. An atomic (all or nothing) transaction fails if any of the resource request cannot be satisfied. On the other hand, an incremental transaction allocates all the non-conflicting requests. Fortunately, conflict rate is low when evaluated on Google traces with incremental conflict resolution schemes. However, the performance also depends on how cell state is shared between framework schedulers, which is unclear from the paper.

Boutin et al. present Apollo [14], cloud-scale scheduler deployed at Microsoft. Analogous to AM, RM and NM in Hadoop YARN, Apollo contains per-job, per-cluster and per-node components called Job Manager (JM), Resource Monitor (RM) and Process Node (PN) respectively. However, unlike YARN, scheduling responsibilities are delegated to JM instead of RM, which makes it a multi-agent scheduler. RM collects heartbeats (advertised load values) from nodes and provide JM with cluster state. The scheduling algorithm, employed at JM, is a hybrid of previously discussed frameworks and involves communication between JM and PN. Unlike Omega, Apollo defers conflict resolution until after tasks are dispatched. Some of the characteristics of the workload on which the system is evaluated are strikingly similar to those of Google traces used in this report. For instance, task duration distribution of both workloads spans wide range of running times where duration of long running tasks are almost four order of magnitude larger than that of short ones. It suggests that Google traces are representative of workload from large cloud compute clusters across industry.

As already mentioned, the four schedulers discussed above: Mesos, Omega, Sparrow and Apollo; belong to multi-agent category of scheduling architectures. One of the reason mentioned for taking the multi-agent approach is the throughput and scheduling delay limitations of single-agent architectures, especially when scheduling short tasks over large cloud clusters. However, our implementation of monolithic scheduler, running on a 16 core (hyperthread enabled) machine with 128 GB of memory, is able to efficiently schedule workload from Google traces over a 6000 node emulated cluster, while offering a scheduling delay of less than 100 ms for 90% of the jobs. Given that the minimum task duration in Google workload from May 2011, is 10 seconds, a scheduling delay of 100 ms constitutes an overhead of 1%. Thus, our evaluation do not support the hypothesis that monolithic scheduler architecture suffers from performance limitation for large scale clusters. How-

ever, if scheduling workload consists of production latency-sensitive jobs, high availability becomes an important requirement, where multi-agent architectures have an advantage over centralized design. Besides, as pointed out by Schwarzkopf et al. [12], the primary reason for Google to shift away from monolithic architecture is *software maintainability*. Since scheduling requirements evolve over time, it becomes increasingly difficult to add new policies to a single monolithic scheduler due to the accumulation of code paths.

Heartbeat mechanism is widely used in distributed systems for monitoring, ensuring fault tolerance and improving availability [26] [27] [28]. In cluster scheduling, it is used to keep track of cluster resource usage and enforcing liveness. Hadoop YARN allows users to configure the heartbeat interval, which is by default set to 1 second. Apollo also involve heartbeat messages from nodes to Resource Monitor. In Mesos, slaves periodically report resource availability to master node. Optimum heartbeat interval depends on the cluster size and nature of the workload. We used the testbed to study the variation of scheduling delay with different heartbeat intervals for Google workload and observed that it has a non-trivial impact on the performance. Schwarzkopf et al. [12] experimented with two versions of monolithic scheduler: single-path (no thread level parallelism) and multi-path. We experimented with a continuous variation of the size of the thread pool serving job requests in monolithic architecture.

Zaharia et al. [29] introduced *delay scheduling* in Hadoop Fair Scheduler. They avoided the approach of killing already running tasks in favor of waiting for resources to be released voluntarily by tasks, in order to achieve fairness. Due to the high rate of number of tasks finishing their execution per unit time in cloud computing workloads, such an approach achieves fairness while avoiding disadvantages of preemption. To improve data locality, the jobs are required to wait for a small extra time for a slot to be available on a machine closer to the data. Quincy [30] can be classified as a monolithic scheduler which maps the task scheduling problem to a graph data structure. In order to meet Service Level Agreements (SLA) associated with jobs, Cake [31] takes a two level scheduler approach where first level schedulers are attached to each individual resource in the cluster and maintains the associated task queue. These first level schedulers are maintained by a central second level scheduler according to job level SLAs. Apart from generic cluster schedulers, there is a plethora of projects which target application specific schedulers.

For instance, Aniello et al. [32] proposes on-line schedulers for Storm, which migrates tasks between machines to minimize the inter-node communication.

Ousterhout et al. [33] point out the benefits of small duration tasks (tiny tasks) in cloud computing environments. Although tiny tasks may be beneficial in terms of straggler mitigation and resource sharing, they would require major changes in existing infrastructures including distributed storage (file) systems, cluster schedulers, execution as well as programming models. Ghodsi et al. [34] study the meaning of fairness of multiple resource types, which is common in cloud clusters. The authors present and evaluate Dominant Resource Fairness (DRF) scheme which provides desirable properties such as strategy-proofness, envy-freeness, sharing incentive and Pareto efficiency in multi resource types environments.

6.2 Analysis of Scheduling Workload

For all experiments in this paper, we have used the publicly available traces from a 12K node multi-tenant Google cluster. A number of researchers have analysed this trace, highlighting the challenges involved in scheduling in cloud computing systems. Reiss et al. [9] have analyzed the trace and have observed significant heterogeneity in the workload in terms of execution duration, placement constraints, number of tasks, resource demands and usage. Apart from being scalable and efficient, the analysis shows the need for flexible resource management for multi-tenant clusters. Liu et al. [35] carried out similar analysis on Google traces.

Chen et al. [36] present a model for the scheduling workload from Google traces along various dimensions such as duration, resource requirement and number of tasks. From empirical observations, authors have characterized jobs into 9 clusters. However, the model is build on a 75 minute long trace. In this study, we worked on workload traces from 29-day period. Mishra et al. [37] also identify workload dimensions in Google traces and qualitatively break down each identified dimension into small, medium and large category. However, it is unclear if such a coarse break-up could be used effectively in sensitivity analysis.

Sharma et al. [38] modeled the task placement constraints in Google clusters and observed that such constraints may increase the scheduling delays

by 2 to 6 times. Abad et al. [39] proposed a model based on delayed renewal processes to generate object access workloads, where an object can be a file, media sessions etc.

Chen et al. [40] analyze MapReduce workload from six separate business-critical deployments inside Facebook and at Cloudera customers in e-commerce, telecommunications, media and retail. The authors observed the MapReduce workload to be highly bursty, unpredictable and heterogeneous. This is consistent with our analysis of Google workload.

CHAPTER 7

CONCLUSION

We developed an experimental testbed to facilitate performance testing of different scheduler architectures over large emulated clusters with diverse workloads from industrial traces. In order to evaluate scheduler architectures on large scale clusters with tens of thousands of machines, we developed the notion of *cluster emulation*. We verified the emulation by showing strong correlation between scheduler performance in emulated and real clusters containing thousands of nodes. We hope that such a testbed would allow research community to study scheduling in large cloud computing systems using relatively modest compute resources.

We show the usefulness of the testbed by thoroughly evaluating the performance of the monolithic scheduler architecture along various design parameters, over Google cluster traces. Our implementation of the monolithic architecture, running on a 16 core (hyperthread enabled) machine with 128 GB of memory, is able to efficiently schedule the workload from Google traces over a 6000 node emulated cluster, while offering a scheduling delay of less than 100 ms for 90% of the jobs. Given that the minimum task duration in Google workload from May 2011, is 10 seconds, a scheduling delay of 100 ms constitutes an overhead of 1%. Thus, we conclude that monolithic scheduler architecture could efficiently handle Google workload.

From our experiments, we conclude that scheduling in large cloud computing environment is a network I/O intensive process. The majority of the scheduling delay in monolithic architecture is contributed by kernel network stack. We found out that a heartbeat interval of five seconds is suitable for Google workload because it exploits the trade-off between failure rate and scheduler cpu load very well. We conclude that a path limit of 100 is enough for handling concurrent job requests without increasing the scheduling delay due to head-of-the-line blocking problem. We discovered that the presence of scheduling constraints in Google workload does not have significant effect on

scheduler cpu load. We conclude that randomness in scheduling algorithm is beneficial to beat the contention due to concurrent job requests.

We thoroughly evaluated the three default schedulers in Hadoop YARN: Capacity, Fair and Fifo, over workload generated by replaying Google traces. Based on our experiments, we conclude that the Fifo scheduler is not suitable for enterprise clusters. It's naive container placement decisions result in unbalanced load across cluster nodes which may result in overloaded nodes to become unresponsive. On the other hand, both Capacity and Fair scheduler are much more suitable for production clusters and keep the load balanced across cluster nodes. The two schedulers exploit different trade-offs. While the Fair scheduler offers less scheduling delay by avoiding head-of-the-line blocking problem, it may drop applications in case the load increases. On the other hand, the Capacity scheduler does not drop any application but errs on the side of higher scheduling delay. Among these two, Fair scheduler performs better for Google workload at its original rate: it provides a scheduling delay of less than 10 seconds for 90% of the jobs as compared to 70% in case of Capacity scheduler. However, this performance gain comes at the cost of longer tail in delay distribution for complex jobs.

7.1 Future Work

We plan to extend this study along the following dimensions.

- We plan to address a couple of simplifying assumptions we made in the trace replays (Section 3.1). Firstly, we are ignoring the re-scheduling events for the failed tasks. Given that the re-scheduling events significantly increase the scheduler load (Figure 3.2), we plan to address this assumption in the next version of testbed. Secondly, the cluster machines in the testbed are currently static. We plan to include the machine events from Google traces to add, remove and update cluster machines during the experiment.
- Apart from monolithic architecture, we plan to implement and evaluate other scheduler architectures using the testbed. We plan to compare the performance and trade-offs of different architectures.

- Hadoop YARN allows for pluggable scheduling policies and provides neat interfaces for writing custom schedulers. Using our scheduler testbed, we are designing and developing a scheduler for YARN optimized for minimizing scheduling delay. Being a popular open source project, YARN is a great way for research community to materialize their research.
- We are also working on characterization and modeling of scheduling workload. We are trying to fit the run-time duration of tasks in a job to well known distributions, so as to represent them with fewer parameters. We plan to use K-means clustering algorithm to group jobs into clusters according to their characteristics (cpu, memory, number of tasks etc). For each cluster of jobs, we would fit the arrival time of constituting jobs into a Poisson distribution. The Workload Generator could modify the mean inter-arrival times of different Poisson distributions (corresponding to different job clusters) to obtain different mixtures of workload, as per the configuration specified by the user. Such a synthetic workload would be useful for 'what-if' analysis.

REFERENCES

- [1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251254.1251264> pp. 10–10.
- [2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.
- [3] “Apache storm project,” <http://storm.incubator.apache.org/>. [Online]. Available: <https://storm.incubator.apache.org/>
- [4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild et al., “Spanner: Googles globally distributed database,” *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.
- [5] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” in *Proc. of the 36th Int’l Conf on Very Large Data Bases*, 2010. [Online]. Available: <http://www.vldb2010.org/accept.htm> pp. 330–339.
- [6] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
- [7] “Apache hbase project,” <http://hbase.apache.org/>. [Online]. Available: <http://hbase.apache.org/>
- [8] C. Reiss, J. Wilkes, and J. L. Hellerstein, “Google cluster-usage traces: format + schema,” Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20. Posted at URL <http://code.google.com/p/googleclusterdata/wiki/TraceVersion2>.

- [9] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC ’12. New York, NY, USA: ACM, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2391229.2391236> pp. 7:1–7:13.
- [10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1972457.1972488> pp. 295–308.
- [11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2523616.2523633> pp. 5:1–5:16.
- [12] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013. [Online]. Available: <http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf> pp. 351–364.
- [13] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow: Distributed, low latency scheduling,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2517349.2522716> pp. 69–84.
- [14] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, “Apollo: Scalable and coordinated scheduling for cloud-scale computing,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin> pp. 285–300.

- [15] A. Konwinski, “Multi-agent cluster scheduling for scalability and flexibility,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2012. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-273.html>
- [16] “Nebula: Workload generator based on google trace,” <https://github.com/gkhaneja/nebula>. [Online]. Available: <https://github.com/gkhaneja/nebula>
- [17] “Experimental testbed for schedulers,” <https://github.com/uiuc-srg/scheduler>. [Online]. Available: <https://github.com/uiuc-srg/scheduler>
- [18] “Hadoop yarn on cloudera hadoop distribution (cdh),” <http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-operators/>. [Online]. Available: <http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-operators/>
- [19] “Hortonworks focus on hadoop yarn,” <http://hortonworks.com/hadoop/yarn/>. [Online]. Available: <http://hortonworks.com/hadoop/yarn/>
- [20] “Apache spark project,” <https://spark.apache.org/>. [Online]. Available: <https://spark.apache.org/>
- [21] “Capacity scheduler for hadoop yarn,” <https://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>. [Online]. Available: <https://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>
- [22] “Fair scheduler for hadoop yarn,” <https://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>. [Online]. Available: <https://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>
- [23] “Scheduling in hadoop,” <http://www.ibm.com/developerworks/library/os-hadoop-scheduling/>. [Online]. Available: <http://www.ibm.com/developerworks/library/os-hadoop-scheduling/>
- [24] “Hadoop yarn releases,” <http://hadoop.apache.org/releases.html>. [Online]. Available: <http://hadoop.apache.org/releases.html>

- [25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.
- [26] F.-f. Li, X.-z. Yu, and G. Wu, “Design and implementation of high availability distributed system based on multi-level heartbeat protocol,” in *Control, Automation and Systems Engineering, 2009. CASE 2009. IITA International Conference on*. IEEE, 2009, pp. 83–87.
- [27] M. Treaster, “A survey of fault-tolerance and fault-recovery techniques in parallel systems,” *arXiv preprint cs/0501002*, 2005.
- [28] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.
- [29] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: ACM, 2010. [Online]. Available: <http://doi.acm.org/10.1145/1755913.1755940> pp. 265–278.
- [30] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 261–276.
- [31] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, “Cake: Enabling high-level slos on shared storage systems,” in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC ’12. New York, NY, USA: ACM, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2391229.2391243> pp. 14:1–14:14.
- [32] L. Aniello, R. Baldoni, and L. Querzoni, “Adaptive online scheduling in storm,” in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.
- [33] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, “The case for tiny tasks in compute clusters,” in *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*. USENIX Association, 2013, pp. 14–14.
- [34] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types.” in *NSDI*, vol. 11, 2011, pp. 24–24.

- [35] Z. Liu and S. Cho, “Characterizing machines and workloads on a google cluster,” in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 397–403.
- [36] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, “Analysis and lessons from a publicly available google cluster trace,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, 2010.
- [37] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, “Towards characterizing cloud backend workloads: insights from google compute clusters,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.
- [38] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, “Modeling and synthesizing task placement constraints in google compute clusters,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 3.
- [39] C. L. Abad, M. Yuan, C. X. Cai, Y. Lu, N. Roberts, and R. H. Campbell, “Generating request streams on big data using clustered renewal processes,” *Performance Evaluation*, vol. 70, no. 10, pp. 704–719, 2013.
- [40] Y. Chen, S. Alspaugh, and R. Katz, “Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads,” *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012. [Online]. Available: <http://dx.doi.org/10.14778/2367502.2367519>