Multi-agent Cluster Scheduling for Scalability and Flexibility

Andrew Konwinski

Multi-agent Cluster Scheduling for Scalability and Flexibility

by

Andrew D. Konwinski

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2012

Multi-agent Cluster Scheduling for Scalability and Flexibility

Abstract

Multi-agent Cluster Scheduling for Scalability and Flexibility

by

Andrew D. Konwinski

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

This dissertation presents a taxonomy and evaluation of three cluster scheduling architectures for scalability and flexibility using a common high level taxonomy of cluster scheduling, a Monte Carlo simulator, and a real system implementation. We begin with the popular Monolithic State Scheduling (MSS), then consider two new architectures: Dynamically Partitioned State Scheduling (DPS) and Replicated State Scheduling (RSS). We describe and evaluate DPS, which uses pessimistic concurrency control for cluster resource sharing. We then present the design, implementation, and evaluation of Mesos, a real-world DPS cluster scheduler that allows diverse cluster computing frameworks to efficiently share resources. Our evaluation shows Mesos achieve high utilization, respond quickly to workload changes, and flexibly cater to diverse frameworks while scaling to 50,000 nodes in simulation and remaining robust. We also show existing and new frameworks sharing cluster resources. Finally, we describe and evaluate RSS, a cluster scheduling architecture being explored by Google in Omega, their next generation cluster management system. RSS uses optimistic concurrency control for sharing cluster resources. We show the tradeoffs between optimistic concurrency in RSS and pessimistic concurrency in DPS and quantify the costs of the added flexibility of RSS in terms of job wait time and scheduling utilization.

Dedicated to my best friend Jocelyn Kirsch. Since my first year as a PhD student, Jocelyn has stood by me through tough and terrific times. I depend daily on her brilliant sense of humor and audacious personality for energy, inspiration, and picking good project names!

Contents

List of Figures

List of Tables

Acknowledgements

My path to this doctoral dissertation has been anything but solitary. As a social creature to my core, I owe much to my mentors, lovers, family, and friends.

Since arriving at Berkeley, Randy H. Katz's advice and friendship have motivated, shaped, and inspired my research. Beyond research, through Randy's mentorship I have grown as a person.

Professors Anthony D. Joseph and Ion Stoica provided feedback, guidance, strategy, and words of encouragement, without which this dissertation would not exist.

I owe much to Malte Schwarzkopf, whose hard work and thoughtful, determined attitude towards research inspired and contributed fundamentally to this dissertation. John Wilkes mentored me at Google; it was through his tactical advice and understanding of my unique personality that I found my stride as an engineer and researcher. Thanks to the rest of my collaborators at Google, particularly Michael Abd-El-Malek, Joseph Hellerstein, Rasekh Rifaat, and Victor Chudnovsky, for their feedback, discussions, and brainstorms.

For their support since before I can remember, I owe my family: my Mom and Dad, Tim and Debbie Konwinski, as well as my sisters Lindsey Gregory and Jamie Murray, and brother-in-law Jared Murray. My family has shown unending support, imbuing me with the confidence to tackle the hardest of problems.

Jocelyn Kirsch has been my closest friend and most intimate supporter since my first year of graduate school. Our relationship has taken many twists and turns, and along the way I have grown profoundly and found deep happiness. Her support and frequent infusions of energy carried me through what felt like impossible challenges. I am lucky to have her in my life.

Beth Trushkowsky has been more than simply a cube-mate to me, through class projects, daunting homework assignments, and countless visits to Euclid Avenue for coffee. She is the most perceptive person I've ever met and one of the best listeners. I look forward to our lifelong friendship, and revisiting our many memories over the years to come, such as when I embarrassed myself explaining what a DAG is to Richard Karp during our 270 class project meeting.

Matei Zaharia has been a personal friend, partner, and role model throughout graduate school. I have learned, grown, traveled, laughed, and worked feverishly with him through many deadlines, conferences, weekend brunches, talks, visits to industry, and more. I look forward to future years of friendship and collaboration with Matei.

To my cohort of fellow graduate students, scholars, and friends at UC Berkeley: Benjamin Hindman, Ali Ghodsi, Michael Armbrust, Patrick Wendell, George Porter, Rodrigo Fonseca, Ariel Rabkin, Isabelle Stanton, Brian Gawalt, Kurtis Heimerl, Kuang Chen, Ariel Kleiner, Tyson Condie, Lester Mackey, Robert Carroll, Reynold Xin, and Andrew Krioukov.

To my original academic mentor, professor Bill Bultman, who first explained to me the concept of a PhD and instilled in me the passion to pursue one. Also, to Evelyn Li for her support and epic feasts during my ramen noodle years.

Jessica Eads has been a role model as well as a source of encouragement, food, and much more; always at just the right moments. Also, to Bradley Sickenius, my personal advisor and source of council, for his stabilizing force in my life.

Also, thanks to my mentors at LANL John Bent, and Meghan Quist, and James Nunez; and also my team and collaborators at Yahoo! including Owen O'Malley, Arun C Murthy, Jerome Boulon, Runping Qi, Eric Yang, and Mac Yang.

Finally, thanks to Remzi Arpaci-Dusseau for his guidance during my formative years as a researcher and pushing me to never settle. Also thank to my high school history teacher Brian Stamper for helping me realize the power of questioning everything and core beliefs.

Curriculum Vitæ

Andrew D. Konwinski

Education

2007	University of Wisconsin, Madison B.S., Computer Science
2009	University of California, Berkeley M.S., Electrical Engineering and Computer Science
2012	University of California, Berkeley Ph.D., Electrical Engineering and Computer Science

Personal

Born	October 15, 1983, Pekin Illinois, United States of America

Chapter 1

Introduction

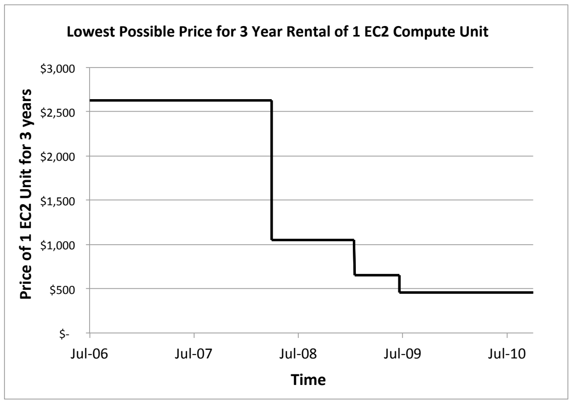
1.1 Motivation

In recent years, clusters of commodity servers have been widely adopted as the primary computing platform for large Internet services, data-intensive scientific applications and enterprise analytics. This has been driven by two trends: (1) the decreasing cost of computing resources (see Figure 1.1) and (2) the increasing accessibility of those resources via public and private provisioning "cloud" interfaces (see [19]). Driven by these trends, researchers and practitioners have been developing a diverse array of cluster computing frameworks to simplify programming the cluster. Prominent examples include MapReduce [24], Dryad [31], MapReduce Online [23] which supports streaming jobs, Pregel [35], a specialized framework for graph computations, and others [25, 34, 38].

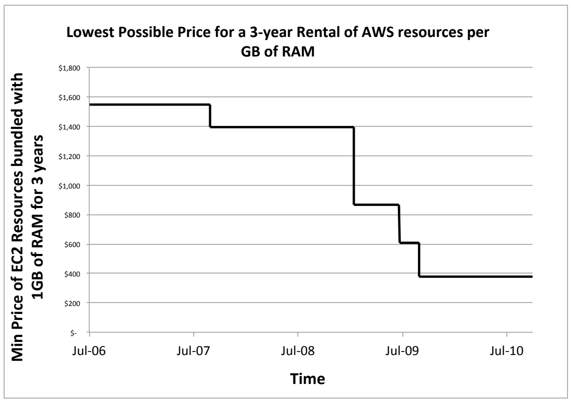
New cluster computing frameworks will likely continue to emerge, and no single framework will be optimal for all applications [1]. Furthermore, multiplexing a cluster between frameworks improves utilization and allows applications to share access to large datasets that may be too costly to replicate across clusters. Therefore, organizations will want to simultaneously run *multiple frameworks in the same cluster*, picking the best one for each application.

In addition to growing demand by organizations to run multiple diverse frameworks, the computations being performed by those frameworks are growing in size (i.e., jobs consisting of increasing numbers of tasks running in parallel across the cluster). This is due to the declining cost of storage resources (i.e., disks), enabling organizations to cost-effectively collect much more data than can be processed on a single machine by using open source distributed storage systems. To support large jobs to process this "big data", clusters are also growing in size (i.e., number of machines). For example, in 2011, Google published a trace from one of their private clusters composed of 12,500 machines [28]. In addition to

[1]By "framework," we mean a software system that manages and executes one or more jobs (i.e., collections of software executables called "tasks") on a cluster.



(a) *The price to rent resources associated with 1 EC2 compute unit for three years.*



(b) *The minimum price to rent a bundle of resources associated with 1GB of ram for three years. Note that this is less than a single EC2 Compute Unit.*

Figure 1.1: *Prices for renting Amazon EC2 resources over time between July 2006 and October 2010. The price is calculated by estimating the cheapest way to acquire 1 EC2 compute unit (left) or the minimum bundle of resources associated with 1GB of RAM (right). Drops in prices are due to price discounts for existing resource types, as well as new types of resources being offered by Amazon, e.g., High CPU instances. Graphs reproduced from [14].*

supporting larger jobs, large clusters allow organizations to achieve improved economies of scale by amortizing cluster management and operational expenses.

We refer to the process an organization uses to allocate the physical compute resources (e.g., CPUs, storage disks, memory) to its users as *cluster scheduling systems* or *resource management systems.* Such systems can be primarily human systems (i.e., operational staff allocating and configuring hardware resources for software developers) or software systems (i.e., a series of interacting daemons that automate allocation and configuration).

Problem Statement: *In the face of increasing demand for cluster resources by diverse cluster computing applications and the growing number of machines in typical clusters, it is a challenge to design cluster schedulers that provide flexible, scalable, and efficient resource allocations.* We can measure Flexibility in terms of the ease of support for multiple frameworks. We measure scalability as the job arrival rate and number of machines that can be managed while providing adequate scheduling response times. Finally, we measure resource utilization as the aggregate usage of the underlying physical resources being managed (e.g., average utilization of all CPUs in the cluster).

Designing a scalable and efficient cluster scheduling system that supports a wide array of both current and future frameworks is challenging for several reasons. First, each framework will have different scheduling needs, based on its programming model, communication pattern, task dependencies, and data placement. Second, the scheduling system must scale to clusters of tens of thousands of nodes running thousands of jobs with millions of tasks. Finally, because all the applications in the cluster depend on the scheduling system, the system must be fault-tolerant and highly available.

In this dissertation, we examine and classify the architectures of existing software cluster scheduling systems with respect to flexibility, scalability, and efficiency. Additionally, we propose and evaluate novel *multi-agent* cluster scheduling architectures, and implementations of them, that achieve high cluster utilization while providing a more flexible abstraction to framework developers and scaling to clusters with an order of magnitude more machines than existing *single-agent* scheduling architectures.

Historically, cluster scheduling systems are monolithic, which we use to mean architected with a single scheduling agent process that makes all scheduling decisions sequentially. The agent takes input about framework requirements, resource availability, and organizational policies, and computes a global schedule for all tasks. While this approach can optimize scheduling across frameworks, it faces several challenges. The first is complexity. The scheduler would need to provide a sufficiently expressive API to capture all frameworks' requirements, and must solve an online optimization problem for potentially millions of tasks. Even if such a scheduler was feasible, this complexity would have a negative impact on its scalability, resilience, and maintainability. Second, as new frameworks and new scheduling policies for current frameworks are continuously being developed [33, 47, 48, 51], it is not clear whether we are even at the point to have a full specification of framework requirements. Third, many existing frameworks implement their own sophisticated scheduling [32, 48], and moving this functionality to a global scheduler would require expensive refactoring.

1.2 Requirements

Here we outline a number of requirements that must be met by the design of resource schedulers for today's cluster workloads. In this dissertation, we will evaluate a number of cluster scheduling architectures based on their ability to satisfy the following requirements:

Scalability. The aggregate number of machines and job-arrival-rate that can be handled by a single cluster management system. Specifically, we expect scheduler response times to remain fixed as we simultaneously add more machines and increase job-arrival-rate or job size.

Flexibility. Support running a highly heterogeneous mix of jobs simultaneously on a common cluster (e.g., batch analytics and web frameworks).

Usability and Maintainability. Easily adapt new types of jobs and problems to run on the cluster. Simplify deploying, and upgrading software frameworks.

Fault isolation. Minimize dependencies between unrelated jobs to avoid cascading failures.

Utilization. Achieve high cluster resource utilization according to several metrics, e.g., cpu utilization, memory utilization, etc.

1.3 Approach and Contributions

In summary, we aim to understand existing cluster scheduling systems, and then apply that understanding to design and implement a better cluster scheduling system. After outlining our assumptions about the workload and hardware environments we target, we propose a general model and taxonomy that will be used throughout the dissertation to compare three cluster scheduling architectures and concrete software implementations of those.

At a high level, we strive to apply the following iterative research process:

1. Use our general model and taxonomy to define and explain a cluster scheduling architecture.
2. Analyze and compare the architecture to the others using Monte Carlo simulations with simplifying assumptions.
3. When possible, describe and evaluate a real-world implementation of the architecture.
4. Repeat for next architecture.

In our first iteration of this research methodology, we apply our general model and taxonomy to identify two existing cluster scheduling architectures: *Monolithic State Scheduling (MSS)* and *Statically Partitioned State Scheduling (SPS)*. We provide a simulation based evaluation of MSS to observe the architectural limitations along the key performance metrics of scheduler utilization and amount of time jobs spend in pending queues.

We next use our general model and taxonomy to propose a novel, more decentralized, cluster scheduling architecture: *Dynamically Partitioned State Scheduling (DPS).* We analyze the architecture based on the requirements described in Section 1.2, and then describe in detail our implementation and evaluation of Mesos, a cluster scheduler that we built with the DPS architecture.

Finally, we use our general model and taxonomy to identify an even more decentralized architecture, *Replicated State Scheduling (RSS)*, which we have explored in collaboration with Google. We analyze the architecture and compare it to MSS and DPS using a simulation framework we built, that plays synthetic cluster workload traces parameterized using measurements of real Google workload traces.

In summary, our contributions are:

- A taxonomy, general model, and simulation framework to compare the scalability, flexibility, and utilization of the MSS, DPS, and RSS architectures.

- The design, implementation, and evaluation of Mesos, a Dynamically Partitioned State scheduling system that scales to clusters of tens of thousands of machines, provides increased framework development flexibility, and increases resource utilization.

- An evaluation of the design of Omega, a Replicated State Scheduling system being explored by Google.

1.4 Dissertation Structure

The remainder of this dissertation begins by presenting two taxonomies in Chapter 2. First a taxonomy of cluster workloads, and then a general taxonomy of cluster schedulers that we will use throughout the dissertation to compare of a number of existing and new cluster scheduling architectures.

In Chapter 3, we introduce and describe our cluster scheduling Monte Carlo simulation framework and use it along with our scheduler taxonomy to define and evaluate the *Monolithic State Scheduling (MSS)* architecture.

Chapter 4 introduces the general concept of multi-agent scheduling, and focuses on a particular class of scheduling architectures, called Partitioned State Scheduling, in which cluster state is divided between multiple scheduling agents as non-overlapping scheduling domains. The discussion is further divided into independent discussions of two subclasses of PSS: *Statically Partitioned State Scheduling (SPS)* and *Dynamically Partitioned State Scheduling (DPS)*. We conclude the chapter with an analysis of the behavior of DPS using a simple performance model and a Monte Carlo simulator.

In Chapter 5, we present the design, implementation, and evaluation of Mesos, a real world DPS cluster management system.

Chapter 6 introduces a new scheduling architecture we are exploring in collaboration with Google, called *Replicated State Scheduling (RSS)*, in which scheduling domains may overlap and optimistic consistency control is used to resolve conflicting transactions. Then we evaluate the overhead of conflicts using a realistic simulator playing real google workload traces.

Finally, we conclude in Chaper 7.

Chapter 2

Related Work and Taxonomy of Cluster Scheduling

We begin this Chapter in Section 2.1 by laying a foundation documenting our assumptions about the cluster computing environment we use throughout the rest of this dissertation. In Section 2.2, we introduce a taxonomy of cluster workloads based on key dimensions such as job arrival rate and job pickiness. We then present a case study of real workload traces from production Facebook clusters. Section 2.3 discusses the job scheduling problem, and provides a general modeling framework to understand and compare cluster scheduling architectures. This framework consists of component definitions and a description of the purpose and process of cluster scheduling, i.e., how the involved components interact, and outlines the design tradeoff space for cluster scheduling architectures. Then, in Section 2.4 we propose a taxonomy of cluster scheduling architectures to use as a reference throughout this dissertation. Section 2.5 presents a survey of existing cluster scheduling systems.

2.1 Target Cluster Environment

In this section, we list the key assumptions about the cluster environment assumed by the three scheduling architectures we evaluate in this dissertation. These assumptions focus on the types and arrangement of resources in the cluster.

- *Use of commodity servers:* The clusters targeted by the systems we evaluate employ inexpensive but unreliable servers. Such commodity hardware is composed of cheaper, lower quality, components that are more likely to fail. Organizations build clusters of commodity servers to achieve cost benefits associated with moving fault tolerance from hardware to software [19, 20]. Therefore, the applications running over these systems must tolerate hardware failures. This assumption implies that all cluster scheduling architectures must be able to recover from the failure of any machine in the cluster.

- *Tens to hundreds of thousands of servers:* Large internet companies, such as Google and Facebook, maintain some of the world's largest clusters, with hundreds of thousands of machines [20]. As cluster computing continues to become more mainstream with the availability of an increasing number of open source cluster computing frameworks, non-tech organizations will continue to build larger and larger clusters. Cluster size has a direct impact on the design of the cluster resource scheduling systems for two main reasons. First, the scheduling decision time of many popular scheduling algorithms is a function of the number of machines being scheduled over. Second, as the size of the cluster increases, typically so does the number of users, and thus the average job arrival rate.

- *Heterogeneous resources:* Large clusters of commodity servers tend to be heterogeneous. We have observed this in clusters inside Google [28], Twitter, Facebook, and more. This includes heterogeneity in software (e.g., machines with different versions of the Linux kernel) and hardware (e.g., machines with varying amounts of memory, or a subset of machines containing flash drives). Much of this heterogeneity results from the natural growth of clusters spread out over years, typically tracking the growth of the organization. Each time that a batch of new servers is added, the newest generation of hardware is purchased. This leads to cross-generation heterogeneity, and directly affects the design of cluster schedulers since many jobs are picky about the hardware they run on, that is, they require or prefer to run on particular types of machines[1].

- *Use of commodity networks:* Typical data centers use Ethernet networks in a tree topology. In these networks, bisection bandwidth per node is much lower than that of the outlink from each node, which in turn is lower than the total bandwidth of the disks on the node. Therefore, to run efficiently, data and communication intensive applications must control the placement of their computations and the design of cluster schedulers must facilitate jobs expressing such placement preferences.

Note that many of the above assumptions about cluster environment differ from typical high performance computing environments (see Section 2.5.1).

2.2 Cluster Workload Taxonomy

We now discuss the kinds of workloads found in our target clusters *that affect the design of cluster scheduling architectures.* We make this concrete through a case study of Facebook's Hadoop workload. The workloads of interest can vary along the following dimensions:

Service Jobs vs. Terminating Jobs. There are two primary types of jobs used in cluster frameworks: service jobs and terminating jobs. Service Jobs consist of a set of

[1]We define and discuss "picky" jobs in detail in Sections 2.2 and 4.7

service tasks that conceptually are intended to run forever, and these tasks are interacted with by means of request-response interfaces, e.g., a set of web servers or relational database servers. Time and resources are required to perform startup activities for each new job and task, such as forking operating system processes or threads, loading data from disk into memory, and populating caches. Service Jobs are useful because they amortize the cost of such startup activities across many service requests. Terminating Jobs, on the other hand, are given a set of inputs, perform some work as a function of those inputs, and are intended terminate eventually. Scientific applications, such as weather simulations, are often written as terminating jobs. They take a set of input parameters, compute for hours or days, and return a set of output parameters. Most existing cluster management systems were not built to support both service and terminating jobs. In fact, traditional HPC cluster management systems support only terminating jobs, requiring users to specify an estimate of job run-time with each scheduling request, and terminating the job after that amount of time has elapsed. Part of our evaluation of the three scheduling architectures we propose in this dissertation is to gauge their support for both types of jobs. We call frameworks that manage service jobs *"service frameworks"* and those that manage terminating jobs *"terminating-job frameworks"*.

Task pickiness. Tasks can have requirements and preferences about which physical resources they are bound to. For example, a job's tasks may require machines that have a public IP address. Pickiness is a measurement of how many cluster resources could potentially satisfy a task's requirements or preferences; more picky implies fewer resources are satisfactory. This dimension of cluster workload has major implications for the design of the cluster scheduling architecture, since users must be able to describe the requirements or preferences for their jobs to the cluster scheduling system.

Job elasticity. Some jobs are able to use different amounts of resources at different times, based on load. Other jobs require a fixed amount of resources for the duration of their runtime. This dimension impacts the design of cluster schedulers at the level of the API that frameworks use to be acquire resources for their tasks. Cluster schedulers that do not provide an easy way for frameworks to dynamically grow, shrink, or swap out the resources they are using are less "flexible" and, in general, achieve lower utilization of resources.

Job and task duration. The length of jobs and tasks can range from very short to very long. We also refer to this dimension as granularity; we call short jobs or tasks "fine grained" and long ones "coarse grained". While service job are intended to run until they fail or are manually restarted, the duration of terminating jobs may vary widely. Job and task granularity are relevant to the design of cluster schedulers. Fine grained jobs are likely to be more negatively impacted by the time they spend waiting to be scheduled in the Job Queue of a Job Manager, which can happen when the scheduling agent becomes heavily utilized. Task granularity has a major effect on the number of tasks that must be handled per unit time in the system. The more fine-grained the tasks, the more tasks that can complete, and thus the more that must be scheduled per unit time.

Task scheduling-time sensitivity. Service frameworks have traditionally been used when requests require very quick response times, for example, web servers that must respond

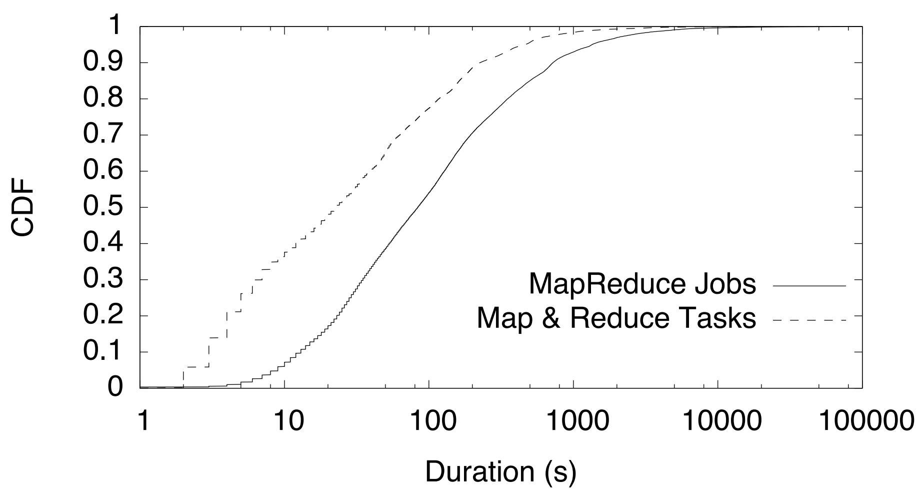


Figure 2.1: *CDF of job and task durations in Facebook's Hadoop data warehouse (data from [48]).*

to requests for web pages within hundreds of milliseconds. On the other hand, terminating-job frameworks have traditionally been used for batch applications. However, terminating-job frameworks are increasingly being used for latency sensitive workloads. Examples of such uses include interactive data analysis through SQL interfaces such as Hive[2] on top of Hadoop, and periodic report generating jobs, such as hourly analysis of ad click data. In jobs such as these, tasks have durations as short as a second or less. Thus scheduling delay can have a significant impact on user experience (see the discussion of "Job and task duration" above). This dimension impacts the design of any cluster schedulers that aim to support such latency sensitive workloads, either via service jobs or terminating jobs. Such schedulers must be able to provide some statistical guarantees to jobs about the maximum time tasks will have to wait to be scheduled.

In this thesis we will evaluate scheduling systems based on their ability to handle workloads consisting of mixes of jobs spread across the dimensions discussed above. We next use our taxonomy to classify two specific workload mixes: Facebook's data analytics MapReduce workload, and mixed workloads of terminating and service jobs.

As our first example of a popular workload that can be categorized according to the above taxonomy, many organizations today use the open source Hadoop MapReduce framework [2] as their primary cluster manager, and run only data analytics jobs in those clusters. We now present a case study of the Hadoop workload inside of one such organization, Facebook.

2.2.1 Case Study: Facebook's Hadoop and Hive Workload

As an example of a popular workload that all general cluster schedulers should seek to support, consider the workload of data analytics jobs run inside the Hadoop data warehouse at Facebook [6]. Using the language of our workload taxonomy in Section 2.2, this workload consists exclusively of elastic, terminating jobs, with a mix of fine and coarse grained tasks.

Facebook loads logs from its web services into a Hadoop cluster of 2000 machines, where they are used for applications such as business intelligence, spam detection, and advertisement optimization.

Figure 2.1 shows a CDF of task and job durations in the workload. We see a wide range of durations for both tasks and jobs. Most jobs are short, the median job being 84s long, but some jobs run for many hours. Also, the jobs are composed of fine-grained map and reduce tasks; the median task length is 23s. In addition to "production" jobs that run periodically, the cluster is used for many experimental jobs, ranging from multi-hour machine learning computations to 1-2 minute ad-hoc queries submitted interactively through Hive.

2.2.2 Mixed Workloads: Service and Terminating Jobs

In the case study of Facebook workload above, though we observed a wide range of different job and task durations, all jobs were of the terminating type. There is another even more challenging workload type gaining popularity, consisting of a mixture of terminating and service jobs. Such mixed workloads are attractive to organizations because front-end applications usually have variable load patterns, such as diurnal cycles and spikes. Thus there is an opportunity to scale them down during periods of low load and use free resources to speed up back-end workloads. Google runs a common cluster resource manager that is responsible for scheduling just such a workload consisting of service and terminating jobs [41]. In chapter 6, we present our exploration in collaboration with Google of an architecture aimed to support such mixed workloads.

Next, we shift our focus away from the environment and inputs to cluster scheduling systems, and onto to the design and purpose of the systems themselves. We begin in the next section by presenting a general model identifying the abstract components of cluster scheduling systems.

2.3 A General Model for Cluster Scheduling

In this section we describe the high level process of cluster scheduling and define necessary terminology. We seek a description that is abstract enough to serve as a modeling framework into which a variety of specific cluster scheduling architectures can fit. This will form the framework within which we will propose and evaluate cluster schedulers throughout this dissertation.

[2]Hive[3] is a system that offers a SQL-like programming model and compiles queries into Hadoop MapReduce jobs.

2.3.1 The Purpose and Process of Cluster Scheduling

At a high level, the fundamental objective of a cluster resource scheduler, or just "cluster scheduler", is to pair units of execution, i.e., *tasks*, with the resources required for those executions to run. Tasks must be assigned to a server containing available resources before they can execute. To do this, scheduling agents create *task-resource assignments* (i.e., assign resources such as CPUs and memory to tasks that will execute using those resources) by editing the row of cluster state that represents the server to which the resources belong. Each time a task-resource assignment is made, the scheduling agent modifies the row of cluster state that represents the server, editing each of the columns representing resources required by the task, ensuring that the values never become negative. Jobs are owned by users and may have relative levels of importance, priority, etc. Upon a task's completion, the task-resource assignment is destroyed and the resources that were assigned to the task are freed by again editing the row of cluster state representing the server and increasing the values of columns representing the resources that were used by the task.

2.3.2 Terminology

- **Resource:** A consumable abstraction of a physical device that provides a service that is necessary or useful to accomplish a computational goal. Examples include CPU, disk, and RAM.
- **Server:** A server machine that can hold a number of Resources.
- **Cluster State:** A logical data table, similar to a database table, with one row per server in the cluster. Columns represent different types of resources that a server can contain. Rows can be added, read, updated, or deleted. These operations correspond to the addition, reconfiguration, or removal of whole Servers to/from the cluster or resources to/from individual servers. Figure 2.2 contains a conceptual diagram representing cluster state and a zoomed-in view of a single row.
- **Scheduling Domain:** A subset of the rows in Cluster State.
- **Task:** An atomic unit of computation. A Task optionally "has a" set of resource requirements, and optionally "has a" set of placement constraints.
- **Job:** A unit of workload. A Job "has a" collection of tasks, "has a" list of job-level scheduling constraints, and "has a" User. Jobs can be "service jobs" or "terminating jobs" (see Section 2.2).
- **Task-Resource Assignment:** The pairing of a task with a non-empty set of available resources on a single machine. Conceptually this consists of the following information sufficient to create a transaction that can be atomically "applied" to cluster state: {machine ID, task ID, $\langle \mathit{resource\,name}_1$, $\mathit{resource\,amount}_1\rangle$, $\langle \mathit{resource\,name}_2$, $\mathit{resource\,amount}_2\rangle$, ...}. Specifically, when a task-resource assignment is applied to cluster state, the row of cluster state uniquely identified by *machine ID*, will have the value corresponding to the column associated with $\mathit{resource\,name}_n$ decreased by the corresponding $\mathit{resource\,amount}_n$. For example, a task-resource assignment might look like: {machine ID: 12, task ID: 20, ⟨"CPUs", 2.0⟩, ⟨"Memory (GB)", 1.0⟩}.

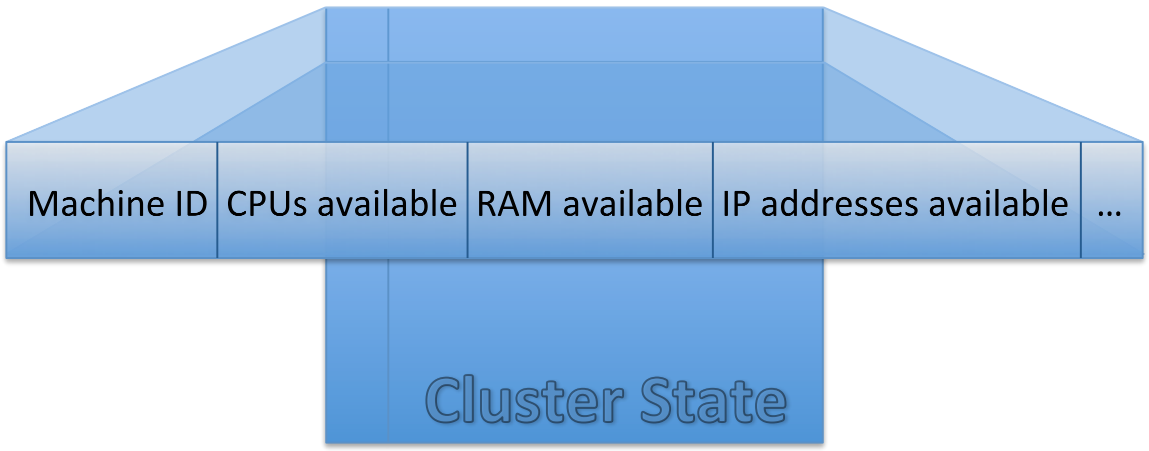


Figure 2.2: *A diagram of "cluster state", a table of data representing the state of available resources on machines in the cluster. The zoomed-in rectangle represents the beginning of what a single row in cluster state might look like.*

- **Scheduling Agent Policy:** The rules and algorithms used by a Scheduling Agent to make Task-Resource Assignments. Examples include: fair sharing, random, priority scheduling.
- **Scheduling Agent:** Creates Task-Resource Assignments based on a Scheduling Agent Policy. A Scheduling Agent "has a" Scheduling Agent Policy. Inputs are a Job and access to a Scheduling Domain. Output is a set of Task-Resource Assignments.
- **Job Transaction:** A set of Task-Resource Assignments for a single job submitted simultaneously to Cluster State.
- **Job Scheduling Decision Time:** The amount of time a Scheduling Agent spends building a job transaction, i.e., attempting to schedule all of the Tasks in a Job, matching them with Servers that have suitable resources available.
- **User:** submits Jobs to Job Managers. A Scheduling Agent Policy may use a Job's User to make decisions about and optionally has an expectation of maximum scheduling time acceptable for each job.
- **Job Queue:** A Queue of Jobs with tasks that have not run to completion.
- **Job Manager:** The entity responsible for managing Jobs and interfacing with Users. A Job Manager "has a" Scheduling Agent and "has a" Job Queue.

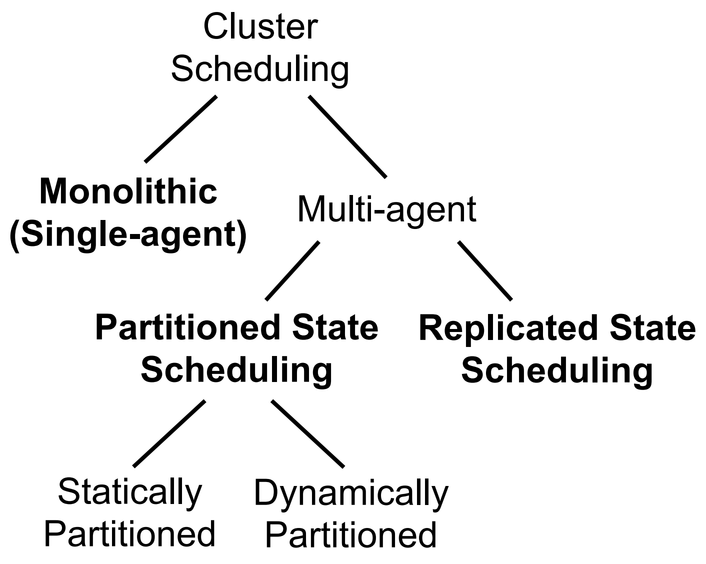

Cluster Scheduling
Monolithic (Single-agent)
Multi-agent
Partitioned State Scheduling
Replicated State Scheduling
Statically Partitioned
Dynamically Partitioned

Figure 2.3: *Overview of the entire cluster scheduler taxonomy.*

2.4 Taxonomy of Cluster Scheduling Architectures

Throughout the rest of this dissertation, we will describe and compare three general types of cluster scheduling architectures: Monolithic State Scheduling (MSS), Partitioned State Scheduling (PSS), and Replicated State Scheduling (RSS). To set the stage, we provide a high level overview of each of these here.

Monolithic State Scheduling. Single scheduling agent, with exclusive access to cluster state, performs all scheduling decisions serially i.e., in order; no job-level scheduling concurrency.

Partitioned State Scheduling. Multiple scheduling agents each perform independent scheduling decisions in parallel on non-overlapping partitions of cluster state. We focus primarily on a variant of PSS in which the partitions are dynamically resized by a centralized meta-scheduling agent called Dynamically Partitioned State Scheduling (DPS).

Replicated State Scheduling. Multiple scheduling agents each maintain full private copies of an authoritative common cluster state. Agents perform their scheduling decisions in parallel. Optimistic concurrency and conflict resolution policies are used to detect and resolve conflicts resulting from concurrent writes to the same row of cluster state.

Figure 2.3 presents the above scheduling architectures in a tree taxonomy. We will return to this figure as we cover each of the three architectures in detail throughout the rest of this dissertation. We cover Monolithic State Scheduling in Chapter 3; Partitioned State Scheduling in Chapters 4 and 5; and Replicated State Scheduling in Chapter 6.

Now that we have established a basic understanding of the abstract components and purpose of a cluster scheduling system and introduced a high level taxonomy of architectures we will consider, we present a survey of related work in the cluster scheduling space.

2.5 Survey of Related Work

In this section, we describe a number of cluster scheduling systems related to our work.

2.5.1 HPC and Grid Schedulers

The high performance computing (HPC) community has long been managing clusters [43, 52] to support scientific computing applications such as very large simulations of natural phenomena (e.g., weather, or nuclear explosions). Their target compute environment typically consists of specialized hardware, such as expensive high speed networking (e.g., infiniband) and storage devices. In their workloads, jobs do not need to be scheduled local to their data. Furthermore, all of the tasks within a typical job are tightly coupled via message passing with synchronization barriers. Also, a job tends not to change its resource demands during its lifetime. At the beginning of a typical HPC job, a number of long running processes (i.e., tasks) are started, these often represent n-dimensional partitions of a large space. These tasks exchange messages with their neighbors as they step through iterations representing forward movement through time, and they may run for weeks or months. Because of the typically long run-time of these tasks, HPC schedulers can easily keep up with the scheduling workload presented. The increasing popularity of computation frameworks with tasks with much shorter average running time has led to a new cluster environment (see Section 2.1) which we will target in this dissertation.

Additionally,HPC schedulers require users to declare the required resources at job submission time. The Job's tasks are then assigned to machines in the cluster, which they use for the life of the job. This does not allow jobs to locally access data distributed across the cluster, and this is not typically a problem because HPC workloads typically do very little reading from disk (i.e., they are "write-heavy"). Similarly, HPC jobs do not usually need to grow or shrink dynamically (though they may be able to benefit from doing so), and so popular HPC schedulers do not focus on features that enable this.

In summary, HPC environments generally have the following characteristics[3]:

- Specialized hardware resources
- Homogeneous resources
- Computationally intensive, write-heavy workloads

Grid computing has mostly focused on the problem of making diverse virtual organizations share geographically distributed and separately administered resources in a secure and interoperable way.

[3]We say "generally" because there are probably cases of HPC jobs or users that are exceptions to all observations.

2.5.2 Public and Private Clouds

Virtual machine clouds such as Amazon EC2 [1] and Eucalyptus [39] isolate applications while providing a low-level abstraction (VMs). The environments they were built to target have a few relevant characteristics that differ from to those that this dissertation is concerned with. These systems generally do not let applications specify placement preferences or requirements beyond the amount of resources their VMs require and some affinity or non-affinity preferences (i.e., whether a job's VMs can be placed on the same physical machine). In the environments we will focus on, applications may require more advanced ways to express their pickiness.

2.5.3 Condor

The Condor cluster manager uses the ClassAds language [40] to match nodes to jobs. Using a resource specification language has implications on scheduling flexibility for applications as we will further discuss in this dissertation, since not all requirements may be expressible, e.g. delay scheduling [48] is not expressible. Also, porting existing frameworks, which have their own schedulers, to Condor (an MSS) may be more difficult than porting them to run in scheduling architectures where existing schedulers fit naturally into a two-level scheduling model (PSS and RSS).

In the next chapter, we present an overview of the Monolithic State Scheduling architecture.

Chapter 3

Monolithic State Scheduling

In this Chapter we present *Monolithic State Scheduling (MSS)*, the first of three scheduling architectures that we evaluate in this dissertation, and our baseline for comparison. We describe the architecture and then evaluate its scalability and flexibility using Monte Carlo simulations.

3.1 MSS Architecture Overview

In MSS, a single scheduling agent is present, contained within a single job manager. Thus, we also refer to MSS as "single-agent scheduling". All task-resource assignments are made serially by this single scheduling agent that implements all policy choices in a single code base. Figure 3.1 shows where this architecture fits into the taxonomy we introduced in Chapter 2. As discussed in Section 2.5, most existing cluster management systems, such as Hadoop and existing HPC systems like Torque [43], use MSS.

In an MSS architecture, though scheduling policies reside within a single code base, different types of jobs may exercise different code paths in the scheduling logic. For example, using the language of our workload taxonomy in Section 2.2, one could imagine non-picky, terminating "batch" jobs being scheduled with a quick-and-simple scheduling algorithm. On the other hand, service jobs might be very picky about the resources they can use in order to meet stringent availability and performance targets and could require computationally expensive task-resource assignment logic to achieve high enough quality task placement.

We refer to scheduling in which task-resource assignments are made using a single scheduling assignment policy, i.e., a single code path, as *single-path* scheduling, and scheduling logic that chooses resources differently according job type, i.e., via multiple code paths, as *multi-path.*

Single-path Monolithic State Schedulers are straightforward to implement and use, but suffer inherent limitations in scale, flexibility, and extensibility. Due to their serial nature, they are intrinsically limited in scale, leading to job queuing backlog as scheduler utilization

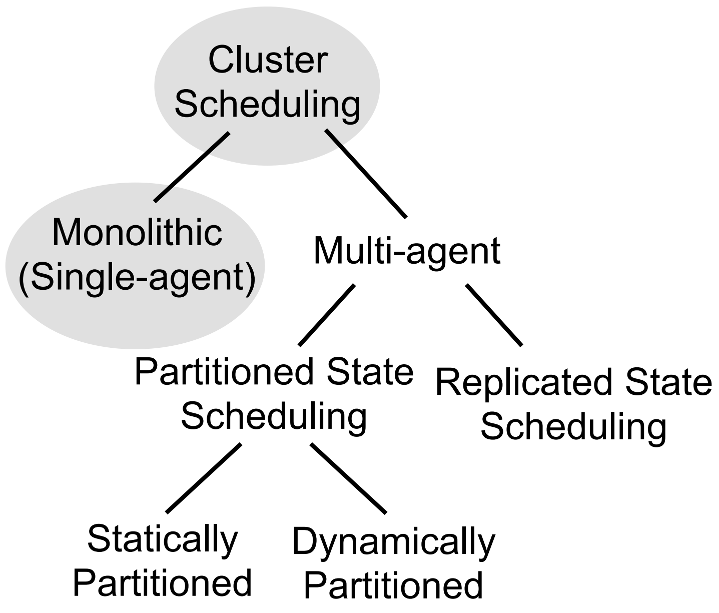

Cluster Scheduling
Monolithic (Single-agent)
Multi-agent
Partitioned State Scheduling
Replicated State Scheduling
Statically Partitioned
Dynamically Partitioned

Figure 3.1: *Our cluster scheduler taxonomy with Monolithic State Scheduling (MSS) highlighted.*

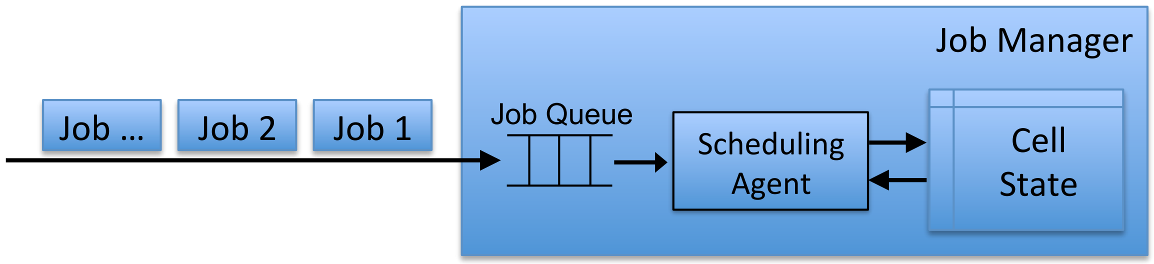


Figure 3.2: *A diagram of the Monolithic Resource Scheduling (RSS) architecture. Jobs are submitted to job manager, and enter a job queue, where they wait until they are handled by the Scheduling Agent, which creates task-resource assignments for each task of the job by editing rows of cluster state.*

approaches 100%. They are also prone to head of line (HOL) blocking, wherein a large and high-priority job with complex scheduling constraints may cause scheduling delays for all subsequent jobs, and hence suboptimal cluster utilization, or, in the worst case, starvation. Also, because all jobs must be handled by a single scheduling algorithm, the scheduling agent's logic must be kept general enough to handle a wide variety of jobs. This limits the flexibility of the scheduling agent, making it difficult to support specialized scheduling logic for each unique workload. This is especially of concern when handling mixed workloads as described in Section 2.2.2. For example, workloads consisting of mostly batch-style jobs may be best scheduled via a simple fair sharing algorithm without preemption, while a workload consisting of latency sensitive jobs may be best scheduled via a strict priority algorithm where tasks are preempted to meet goals for maximum or average task scheduling-time.

Multi-path MSS addresses some of the flexibility concerns with single-path MSS by allowing different scheduling algorithms to be used within a single scheduling agent. However, multi-path MSS does not address the scaling issues associated with a single scheduling agent making all task-resource assignments in serial. Nor does it fully address extensibility issues: using different code paths within a single code base makes it more difficult to independently evolve the separate scheduling algorithms than if they were cleanly separated into different components of the architecture. For example, if a change is fixed in one scheduling path, then service for all job types will be interrupted when the new code is pushed into production and the scheduling agent process is restarted by the organization's cluster operations team. Additionally, the presence of multiple scheduling code-paths introduces the new problems of fault and performance isolation. In Chapter 4, we will present multi-agent scheduling, a natural evolution of multi-path MSS allowing for more independence and parallelism that addresses many of these concerns.

3.2 Monte Carlo MSS Simulation

To explore the performance of the MSS architecture, we collaborated with researchers at Google to implement a simple monolithic scheduler in a Monte Carlo simulator fed by workloads that use exponential distributions for key input parameters such as job interarrival times. The exponential distributions are parameterized by measurements from real Google workloads. In addition to evaluating the MSS architecture, we will further use our Monte Carlo simulator in Chapters 4 and 6 to evaluate and compare PSS and RSS.

Parameters. Our Monte Carlo simulations support multiple schedulers, although the results reported in this dissertation consider just two: batch and service. Each scheduler is fed a workload that has an average $\textit{job arrival rate}_i$ per second (i.e., the inverse of the average inter-arrival time), and an average of $\textit{tasks per job}_i$; both are used as parameters for exponential distributions. Table 3.1 shows the values of $\textit{job arrival rate}_i$ and $\textit{tasks per job}_i$ we derived from measurements of three Google clusters.

We model the scheduler decision time as a linear function of the form $\textit{job time}_i + \textit{task time}_i \times \textit{tasks per job}_i$, where $\textit{job time}_i$ is a constant overhead for a job and $\textit{task time}_i$ represents the incremental cost to place each task. Since most tasks in a job in our real-life workloads have the same resource requirements and constraints, this turns out to be a reasonable approximation of Google's current cluster scheduling logic.

	job arrival rate		*tasks per job*		*task duration*	
cell	batch	service	batch	service	batch	service
A (medium)	0.212/s	0.00274/s	37.19	23.19	274 s	317 s
B (large)	0.377/s	0.00870/s	52.37	8.18	707 s	418 s
C (medium)	0.268/s	0.00114/s	37.33	18.86	289 s	18610 s

Table 3.1: *Mean values of key properties from production Google workloads[42].*

In our experiments, we either vary *decision time*$_i$, using the values of *job arrival rate*$_i$ and *tasks per job*$_i$ presented in Table 3.1, or we vary *job arrival rate*$_i$, in which case we conservatively approximate practical scheduler decision times by using *job time* $= 0.1$ s and *task time* $= 5$ ms.

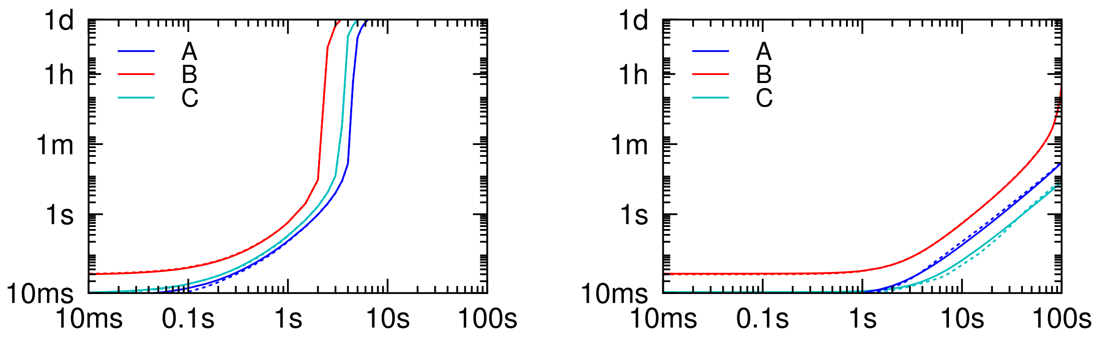
For all scheduling agents, we chose to use a randomized first-fit scheduling algorithm inspired by the logic used by Google's current scheduler. The algorithm used to schedule a single *job* is:

```
pool ← list of all machines in cluster
while pool has more machines and job has more tasks to schedule do
    candidate ← random machine in pool
    if candidate has enough available resources to fit at least one of job's tasks then
        create a task-resource assignment between candidate and job's next task to schedule
    else
        remove candidate from pool
    end if
end while
```

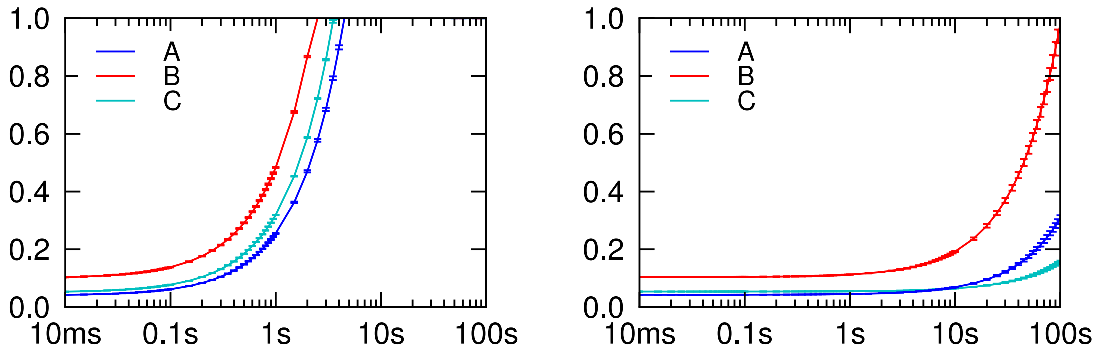
Pre-filling. The Monte Carlo simulator sets up the state of the cluster before each experiment by pre-filling it with jobs representing those that were present at the beginning of the simulation window. For this preloading phase, the simulator uses task-size data extracted from the same real clusters that our distribution parameters are derived from, up to the point where the cluster is about 60% full, which is comparable to utilization levels seen in practice at Google. We use this fill-to-a-point approach because the algorithm used in Google's real production system to over-commit resources is too complicated and specialized to use in our simple simulator. Once the cluster has been pre-filled, the synthetic workload is used to generate job inter-arrival rates and the number of tasks each job has by drawing from exponential distributions with the mean values shown in Table 3.1. For simplicity, each task is sized to 1.1 cores and 1.5GB of memory for all (i.e., pre-filled and non-prefilled) tasks, a common task size seen in practice in Google workloads.

We assume a serial monolithic scheduler with the same ("uniform") decision time for batch and service jobs, to reflect the need to run much of the same code for every job type. We also model a monolithic scheduler with a shorter job scheduling decision time for batch jobs than for service jobs; we refer to this as a *multi-path* monolithic scheduler, as it implies multiple possible code paths through the scheduling logic.

Figure 3.3 compares these approaches by simulating the system, starting with the pre-filled cluster state as described above, and running for 7 day of simulated time. In the



(a) *Job wait time.*



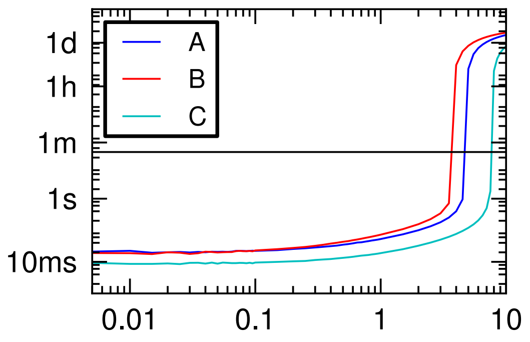
(b) *Scheduler utilization.*

Figure 3.3: *Monolithic schedulers with single-path (left) and multi-path (right): performance as a function of job time in the single-path case, or job time$_{service}$ for the multi-path case (both in seconds).*

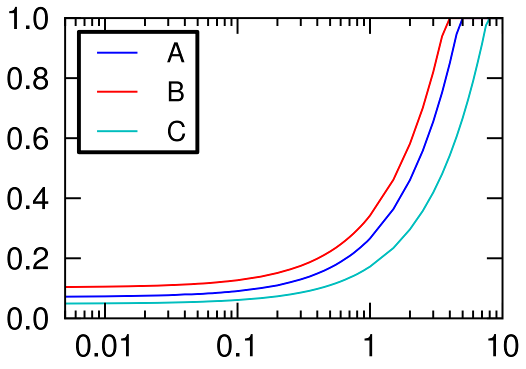
baseline case, we vary the scheduling decision time on the x-axis by changing *job time*. In the multi-path case, we split the workload into batch and service workloads and set *job time*$_{batch}$ = 0.1s and *task time*$_{batch}$ = 5ms, while we vary *job time*$_{service}$, keeping *task time*$_{service}$ = 5ms. FOr the purpose of our evaluation, we chose a baseline SLO for job wait time of 30s based on experience of Google cluster operating experience.

The results are not surprising: in the baseline case, the job wait time is short and the job wait time SLO is easily met until the decision time is large enough that the scheduler is saturated (Figure 3.3), or until the job arrival rate causes saturation of the scheduling agent (Figure 3.4).

With a "fast path" for batch jobs, both average job wait time and scheduler utilization decreased significantly, since the majority of jobs are batch ones. Head-of-line blocking effects still occur, in which "fast-path" batch jobs get stuck behind a service job, and the ultimate scalability is still limited by the processing capacity of a single scheduler. To do better, we need some form of parallel processing.



(a) *Job wait time; the SLO is 30s.*



(b) *Scheduler utilization.*

Figure 3.4: *Results of simulation of 7 days running a monolithic scheduler with uniform decision time: varying job arrival rate (in jobs per second). A, B, and C are Google compute cells as described in table 3.1.*

3.3 Hadoop, a Case Study of Monolithic State Scheduling

Hadoop is a popular cluster scheduling system that implements the Monolithic State Scheduling architecture. In Hadoop, the Job Manager is referred to as the *"JobTracker"*. The Hadoop JobTracker takes a global lock on the Java object representing Cluster State and makes scheduling decisions for each map or reduce task serially.

In addition to scaling limitations, Hadoop also suffers flexibility limitations associated with MSS. As we discussed in Section 2.2.1, Facebook runs large Hadoop clusters. To meet the performance requirements of their jobs, Facebook uses a fair scheduler for Hadoop that takes advantage of the fine-grained nature of the workload to allocate resources at the level of tasks and to optimize data locality [48]. Unfortunately, this means that the cluster can only run Hadoop jobs. If a user wishes to write an ad targeting algorithm using the well-known MPI message passing system (see Section 2.5 for more background on MPI) instead of MapReduce, perhaps because MPI is more efficient for this job's communication pattern, then the user must set up a separate MPI cluster and import terabytes of data into it. This problem is not hypothetical; our contacts at Yahoo! and Facebook report that users want to run MPI and MapReduce Online (a streaming MapReduce) [12, 13].

An ideal scheduling architecture would enable these usage scenarios by providing sharing between *multiple* fine-grained cluster computing frameworks with fundamentally different programming abstractions.

The scaling and flexibility limitations of the architecture of Hadoop discussed above has lead the Hadoop community to build Hadoop Next Generation (or Hadoop NextGen for short) which uses Partitioned State Scheduling.

3.4 Review of Monolithic State Scheduling

In this chapter, we first presented related work, our target cluster and workload environments, as well as a model and taxonomy of cluster scheduling.

Then, using an analytical model and a Monte Carlo simulation we observe scaling and flexibility limitations inherent to the Monolithic State Scheduling architecture. This conclusion is not surprising given the limitations that have been observed in existing real world monolithic cluster scheduling systems (e.g., Hadoop).

In the next chapter, we will introduce multi-agent scheduling architectures, which aim to address many of the scalability and flexibility limitations of Monolithic State Scheduling discussed in this chapter.

Chapter 4

Partitioned State Scheduling

In this chapter, we extend our general model of cluster scheduling by moving beyond monolithic state scheduling and introducing multi-agent scheduling, a technique for increasing the scalability and flexibility of cluster scheduling by allowing scheduling decisions to happen concurrently. We introduce one type of multi-agent scheduling called Partitioned State Scheduling (PSS) in which Cluster State is partitioned into non-overlapping scheduling domains, and discuss Mesos, a real-world system we built that implements PSS.

This chapter is organized as follows: we begin in Section 4.1 by expanding our taxonomy to include the concept of Multi-agent Scheduling, which allows multiple scheduling decisions to be made in parallel. Then, in Section 4.2, we look at two specific approaches to multi-agent scheduling: statically and dynamically partitioning Cluster State. In Section 4.5, we present definitions, metrics, and assumptions. In Section 4.6, we show that Partitioned State Scheduling performs very well when frameworks can scale up and down elastically, tasks durations are homogeneous, and frameworks prefer all nodes equally. In Section 4.7, we consider the case wherein different scheduling agents prefer different machines. We show that partitioned state scheduling can emulate a monolithic scheduler that performs fair sharing across frameworks in terms of scheduling wait times. In Section 4.8, we show that Mesos can handle heterogeneous task durations without impacting the performance of frameworks with short tasks. In Section 4.9, we discuss how frameworks are incentivized to improve their performance in PSS, and argue that these incentives also improve overall cluster utilization.

4.1 Multi-agent Scheduling

Multi-agent scheduling is similar to monolithic scheduling as described in Section 3 except that now multiple scheduling agents can be active at a time, each independently making task-resource assignments. In the rest of this chapter we introduce two types of multi-agent scheduling that both work by dividing Cluster State into non-overlapping par-

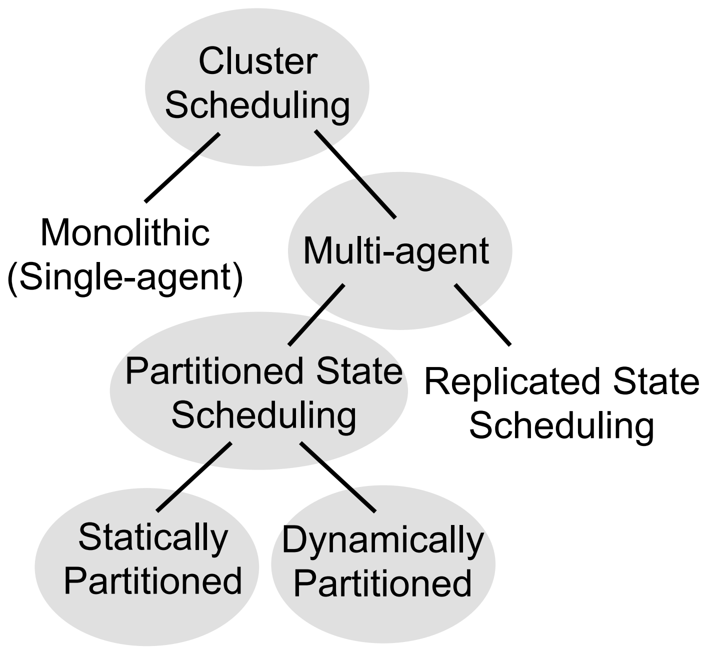

Cluster Scheduling
Monolithic (Single-agent)
Multi-agent
Partitioned State Scheduling
Replicated State Scheduling
Statically Partitioned
Dynamically Partitioned

Figure 4.1: *Our cluster scheduler taxonomy with Statically Partitioned State Scheduling (SPS) and Dynamically Partitioned State Scheduling (DPS) highlighted.*

titions. By dividing the resource scheduling work across multiple schedulers, partitioned state scheduling does not suffer the scaling limitations of monolithic scheduling.

4.2 Partitioned State Scheduling

One way to achieving multi-agent scheduling is to statically partition the rows of Cluster State into non-overlapping partitions and then, for each partition, allow at most a single scheduling agent to edit rows in that partition. This approach can further be broken into two sub-architectures in our taxonomy: static and dynamic partitioning of Cluster State. Figure 4.1 shows where these architectures fit into the taxonomy we introduced in Chapter 2.

4.2.1 Statically Partitioned State Scheduling (SPS)

The most straightforward way to partition Cluster State is to have all partitions be fixed sizes. We call this the *Statically Partitioned State Scheduling (SPS)* architecture. One implementation of this architecture used by most organizations managing clusters today is to partition the machines into "cells" either physically (e.g., via separate physical networks, sets of racks, or perhaps even separate buildings) or virtually (e.g., via Virtual Local Area Networks), each representing an independent scheduling domain. This partitioning may be done in a variety of ways, and also for a variety of reasons other than purely for scheduling scalability.

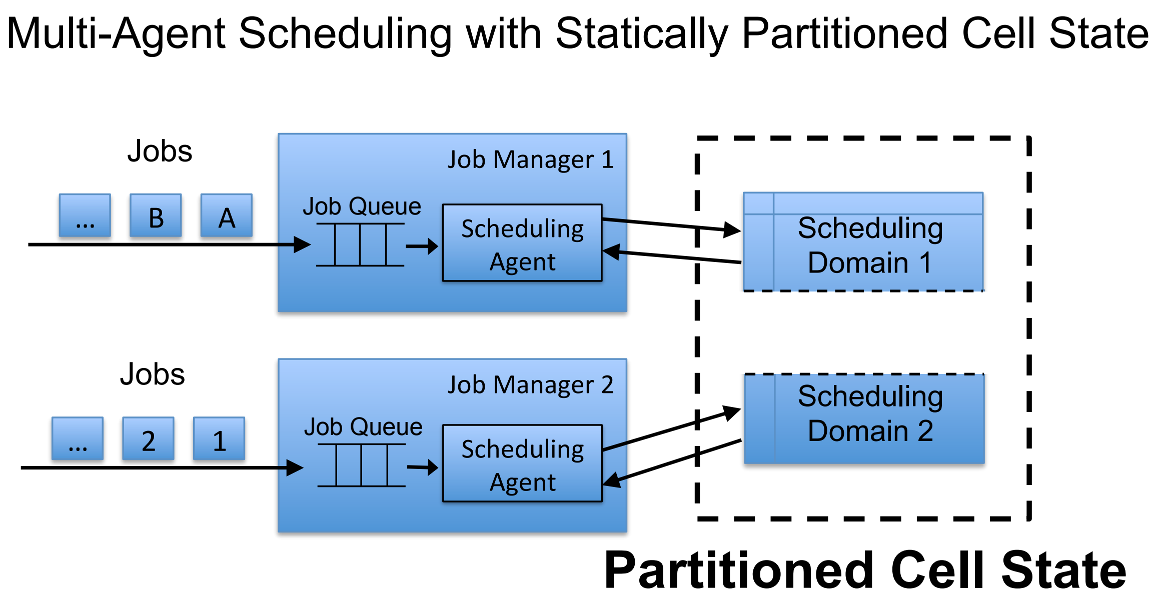


Figure 4.2: *A conceptual diagram showing Partitioned State scheduling with Statically Partitioned Cluster State. The dotted line in cluster state represents the division of rows into two static, non-overlapping scheduling domains, one for each scheduling agent.*

Identical sub-cells (for scale and isolation). Many organizations large enough to run jobs on thousands of machines run many small "cells" each with their own scheduler. This, for example, is how Yahoo runs Hadoop. Some of the advantages of statically partitioned cells are low scheduling decision time latency because of parallelism and improved fault and performance isolation. For example, there is no single point of failure and there is less head of line blocking.

However, there are disadvantages. Jobs that are too big to fit into a single partition simply cannot run, even if there are enough resources globally. Additionally, it is difficult for small organizational sub-units to get a partition assigned to them. It may take days or weeks for resources to get assigned through manual processes. Finally, this approach leads to fragmentation internal to the static partitions and low resource utilization.

4.2.2 Dynamically Partitioned State Scheduling (DPS)

To overcome many of the disadvantages of statically partitioning cluster state that we discussed above, we can allow scheduling domains to be resized dynamically. For example, if one scheduling domain partition is allowed to grow to be the size of the entire cluster, which would imply all other partitions can shrink down to be the null subset of cluster state, this can allow a cluster to support job whose tasks require the all of the resources in the cluster.

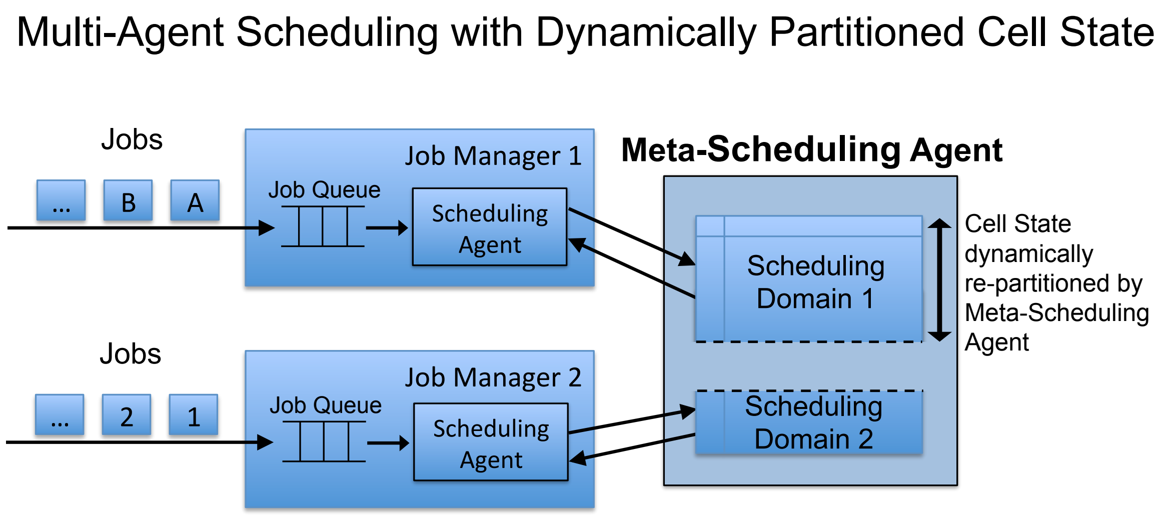


Figure 4.3: *A conceptual diagram showing Partitioned State scheduling with Dynamically Partitioned Cluster State. The meta-scheduling agent is responsible for resizing the non-overlapping scheduling domains according to a meta-scheduling policy.*

Meta-scheduling

However, dynamically resizing scheduling state partitions requires a second, or meta, level of scheduling to make the decision about the size of each partition over time. We call this concept, which is not new to this research, *meta-scheduling*. In our model, we call the entity that contains and enforces the meta-scheduling policies the *meta-scheduling agent*.

Compared to database concurrency control. Since cluster state can be seen as a database table, one way of reasoning about cluster state partitioning is to view the interactions between scheduling agents and cluster state as acquiring and releasing read and write locks on. For example, partitioning cluster state can be seen as each scheduling agent taking a *write lock* on the rows of cluster state that represent the machines in its scheduling domain partition. This would prevent other scheduling agents from reading or writing that row. However, unlike in typical applications for which databases are used, in which write locks are acquired for relatively short periods of time to update the data contained in the row, write locks on rows in cluster state represent control of physical resources which can then be used to run jobs. This distinction explains why little research has been done in the database community on the topic of policies for controlling the size or duration of write locks taken by different database clients, e.g., to enforce fairness or priority levels, whereas in the case of sharing cluster resources these policies are very important. The existence of these policies are the difference between “meta scheduling” in the context of clusters scheduling and classical concurrency control research.

DPS with Resource Offers

While one could envision the Job Managers attempting to take pessimistic locks on partitions of cluster state in a "pull" fashion, in this work we focus on an implementation of DPS in which the meta-scheduling agent actively decides how to partition cluster state according to some central sharing policy and then actively "pushes" updates about the partitions out to scheduling agents. We call this *Mesos-style DPS*, as this is the implementation we chose to use in the Mesos scheduling system, which we will discuss shortly. In Mesos, these updates are called *resource offers*. See Section 5.2.2 for a full description and discussion about resource offers.

4.3 Monte Carlo DPS Simulation

We modified our Monte Carlo simulator to evalute Mesos-style DPS cluster scheduling. Our setup here is parallel to our setup of Monte Carlo simulations of MSS in Section 3.2. We are still simulating one scheduler handling service jobs and one handling batch jobs.

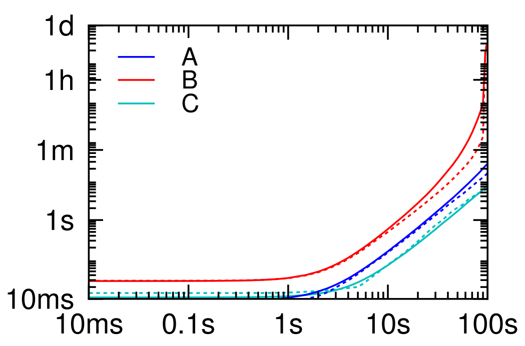
To simulate DPS, we added a meta-scheduling agent (or *resource allocator*) to the simulation framework. The resource allocator can make offers to frameworks as soon as resources free up, even if the scheduler is in the middle of making a scheduling decision, but to keep things simple, we assumed that a scheduler would only look at the set of resources that were available to it when it begins a scheduling attempt for a job (i.e., offers that arrive during the attempt are ignored). Resources that are not used at the end of scheduling a job are returned to the allocator; they may be re-offered back to the framework if it is the one furthest below its fair share.

Since we are now using two schedulers, we keep the decision time for the batch scheduler constant, and vary the decision time for the service scheduler by adjusting $job\ time_{service}$. (We chose to vary the service scheduler decision times because we were particularly interested in the effects of long service scheduling decisions.) As expected, the utilization graphs for the schedulers (Figure 4.4b) look very similar to those of the monolithic multi-path case, because the centralized resource allocator's DRF algorithm is quite fast (we assumed it took 1ms to make a resource offer).

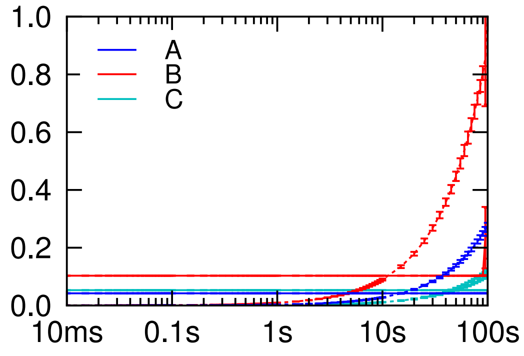
4.4 Analyzing Mesos DPS Behavior

In this section, we further study the behavior of Mesos's implementation of the DPS architecture under a variety of cluster workloads varying in characteristics such as task duration distribution, job elasticity, and job pickiness.

Our goal is not to develop an exact model of the system, but to provide a coarse understanding of its behavior, to further characterize the environments in which Mesos style DPS works well.



(a) *Job wait time.*



(b) *Scheduler utilization.*

Figure 4.4: *Two-level scheduling (Mesos): performance as a function of job time$_{service}$ for clusters A, B, and C (see Table 3.1 for description of each cluster).*

4.5 Definitions, Metrics and Assumptions

In the following discussion, we consider three metrics:

- *Framework ramp-up time:* time it takes a new framework to achieve its allocation (e.g., fair share);
- *Job completion time:* time it takes a job to complete, assuming one job per framework;
- *System utilization:* total cluster utilization.

We characterize workloads along two dimensions: elasticity and task duration distribution. An *elastic* framework, such as Hadoop and Dryad, can scale its resources up and down, i.e., it can start making progress as soon as its first task is assigned to a node, and as its tasks finish, the nodes they occupy can be released. In contrast, a *rigid* framework, such as MPI, can start running its jobs only after it has acquired a fixed quantity of resources, and cannot scale up dynamically to take advantage of new resources or scale down without a large impact on performance. For task durations, we consider both homogeneous and heterogeneous distributions.

We also differentiate between two types of resources: mandatory and preferred. A resource is *mandatory* if a framework must acquire it to run. For example, a graphical processing unit (GPU) is mandatory if a framework cannot run without access to GPU. In contrast, a resource is *preferred* if a framework performs "better" using it, but can also run using another equivalent resource. For example, a framework may prefer running on a node that locally stores its data, but may also be able to read the data remotely if it must.

We assume the amount of mandatory resources requested by a framework never exceeds its guaranteed share. This ensures that frameworks will not deadlock waiting for the

	Elastic Framework		Rigid Framework	
	Constant dist.	Exponential dist.	Constant dist.	Exponential dist.
Ramp-up time	T	$T \ln k$	T	$T \ln k$
Completion time	$(1/2 + \beta)T$	$(1 + \beta)T$	$(1 + \beta)T$	$(\ln k + \beta)T$
Utilization	1	1	$\beta/(1/2 + \beta)$	$\beta/(\ln k - 1 + \beta)$

Table 4.1: *Ramp-up time, job completion time and utilization for both elastic and rigid frameworks, and for both constant and exponential task duration distributions. The framework starts with no slots. k is the number of slots the framework is entitled to under the scheduling policy, and βT represents the time it takes a job to complete assuming the framework gets all k slots at once.*

mandatory resources to become free.[1] For simplicity, we also assume that all tasks have the same resource demands and run on identical slices of machines called *slots*, and that each framework runs a single job.

4.6 Homogeneous Tasks

We consider a cluster with n slots and a framework, f, that is entitled to k slots. For the purpose of this analysis, we consider two distributions of the task durations: constant (i.e., all tasks have the same length) and exponential. Let the mean task duration be T, and assume that framework f runs a job which requires βkT total computation time. That is, when the framework has k slots, it takes its job βT time to finish.

Table 4.1 summarizes the job completion times and system utilization for the two types of frameworks and the two types of task length distributions. As expected, elastic frameworks with constant task durations perform the best, while rigid frameworks with exponential task duration perform the worst.

4.6.1 Elastic Frameworks

An elastic framework can opportunistically use any slot offered by Mesos, and can relinquish slots without significantly impacting the performance of its jobs. We assume there are exactly k slots in the system that framework f prefers, and that f waits for these slots to become available to reach its allocation.

[1]In workloads where the sum of the mandatory resource demands of the active frameworks can exceed the capacity of the cluster, the allocation module needs to implement admission control.

Framework ramp-up time. If task durations are constant, it will take framework f at most T_s time to acquire k slots. This is simply because during a T_s interval, every slot will become available, which will enable Mesos to offer the framework all its k preferred slots.

If the duration distribution is exponential, the expected ramp-up time is $T_s \ln k$. The framework needs to wait on average T_s/k to acquire the first slot from the set of its k preferred slots, $T_s/(k-1)$ to acquire the second slot from the remaining $k-1$ slots in the set, and T_s to acquire the last slot. Thus, the ramp-up time of f is

$$T_s \times (1 + 1/2.. + 1/k) \simeq T_s \ln k. \tag{4.1}$$

Job completion time. Recall that βT_s is the completion time of the job in an ideal scenario in which the frameworks acquires all its k slots instantaneously. If task durations are constant, the completion time is on average $(1/2 + \beta)T_s$. To show this, assume the starting and the ending times of the tasks are uniformly distributed, i.e., during the ramp-up phase, f acquires one slot every T_s/k on average. Thus, the framework's job can use roughly $T_s k/2$ computation time during the first T_s interval. Once the framework acquires its k slots, it will take the job $(\beta k T_s - T_s k/2)/k = (\beta - 1/2)T_s$ time to complete. As a result the job completion time is $T_s + (\beta - 1/2)T_s = (1/2 + \beta)T_s$ (see Table 4.1).

In the case of the exponential distribution, the expected completion time of the job is $T_s(1 + \beta)$ (see Table 4.1). Consider the ideal scenario in which the framework acquires all k slots instantaneously. Next, we compute how much computation time does the job "lose" during the ramp up phase compared to this ideal scenario. While the framework waits T_s/k to acquire the first slot, in the ideal scenario the job would have been already used each of the k slots for a total of $k \times T_s/k = T_s$ time. Similarly, while the framework waits $T_s/(k-1)$ to acquire the second slot, in the ideal scenario the job would have been used the $k-1$ slots (still to be allocated) for another $(k-1) \times T_s/(k-1) = T_s$ time. In general, the framework loses T_s computation time while waiting to acquire each slot, and a total of kT_s computation time during the entire ramp-up phase. To account for this loss, the framework needs to use all k slots for an additional T_s time, which increases the expected job completion time by T_s to $(1 + \beta)T_s$.

System utilization. As long as frameworks can scale up and down and there is enough demand in the system, the cluster will be fully utilized.

4.6.2 Rigid Frameworks

Some frameworks may not be able to start running jobs unless they reach a minimum allocation. One example is MPI, where all tasks must start a computation in sync. In this section we consider the worst case where the minimum allocation constraint equals the framework's full allocation, i.e., k slots.

Job completion time. While in this case the ramp-up time remains unchanged, the job completion time will change because the framework cannot use any slot before reaching its full allocation. If the task duration distribution is constant the completion time is simply

$T(1+\beta)$, as the framework doesn't use any slot during the first T interval, i.e., until it acquires all k slots. If the distribution is exponential, the completion time becomes $T(\ln k + \beta)$ as it takes the framework $T \ln k$ to ramp up (see Eq. 4.1).

System utilization. Wasting allocated slots has also a negative impact on the utilization. If the task duration is constant, and the framework acquires a slot every T/k on average, the framework will waste roughly $Tk/2$ computation time during the ramp-up phase. Once it acquires all slots, the framework will use βkT to complete its job. The utilization achieved by the framework in this case is then $\beta kT/(kT/2 + \beta kT) \simeq \beta/(1/2+\beta)$. If the distribution is exponential, the utilization is $\beta/(\ln k - 1 + \beta)$ (see [30] for full derivation).[e]

If the task distribution is exponential, the expected computation time wasted by the framework is $T(k\ln(k-1)-(k-1))$. The framework acquires the first slot after waiting T/k, the second slot after waiting $T/(k-1)$, and the last slot after waiting T time. Since the framework does not use a slot before acquiring all of them, it follows that the first acquired slot is idle for $\sum_{i=1}^{k-1} T/i$, the second slot is idle for $\sum_{i=1}^{k-2} T/i$, and the next to last slot is idle for T time. As a result, the expected computation time wasted by the framework during the ramp-up phase is

$$T \times \sum_{i=1}^{k-1} \frac{i}{k-i} =$$
$$T \times \sum_{i=1}^{k-1} \left(\frac{k}{k-i} - 1\right) =$$
$$T \times k \times \sum_{i=1}^{k-1} \frac{1}{k-i} - T \times (k-1) \simeq$$
$$T \times k \times \ln(k-1) - T \times (k-1) \simeq$$
$$T \times (k\ln(k-1) - (k-1))$$

Assuming $k \gg 1$, the utilization achieved by the framework is $\beta kT/((k\ln(k-1)-(k-1))T + \beta kT) \simeq \beta/(\ln k - 1 + \beta)$ (see Table 4.1).

4.7 Placement Preferences

So far, we have assumed that frameworks have no slot preferences. In practice, different frameworks prefer different nodes and their preferences may change over time. In this section, we consider the case where frameworks have different preferred slots.

The natural question is how well Mesos will work compared to a monolithic scheduler whose scheduling agent has full information about framework preferences. We consider two cases: (a) there exists a system configuration in which each framework gets all its preferred slots and achieves its full allocation, and (b) there is no such configuration, i.e., the demand for some preferred slots exceeds the supply.

In the first case, it is easy to see that, irrespective of the initial configuration, the system will converge to the state where each framework allocates its preferred slots after at most one T interval. This is simple because during a T interval all slots become available, and as a result each framework will be offered its preferred slots.

In the second case, there is no configuration in which all frameworks can satisfy their preferences. The key question in this case is how should one allocate the preferred slots across the frameworks demanding them. In particular, assume there are p slots preferred by m frameworks, where framework i requests r_i such slots, and $\sum_{i=1}^{m} r_i > x$. While many allocation policies are possible, here we consider a weighted fair allocation policy where the weight associated with framework i is its intended total allocation, s_i. In other words, assuming that each framework has enough demand, we aim to allocate $p \cdot s_i / (\sum_{i=1}^{m} s_i)$ preferred slots to framework i.

The challenge in Mesos is that the scheduler does not know the preferences of each framework. Fortunately, it turns out that there is an easy way to achieve the weighted allocation of the preferred slots described above: simply perform lottery scheduling [46], offering slots to frameworks with probabilities proportional to their intended allocations. In particular, when a slot becomes available, Mesos can offer that slot to framework i with probability $s_i / (\sum_{i=1}^{n} s_i)$, where n is the total number of frameworks in the system. Furthermore, because each framework i receives on average s_i slots every T time units, the results for ramp-up times and completion times in Section 4.6 still hold.

4.8 Heterogeneous Tasks

So far we have assumed that frameworks have homogeneous task duration distributions, i.e., that all frameworks have the same task duration distribution. In this section, we discuss frameworks with heterogeneous task duration distributions. In particular, we consider a workload where tasks are either short or long, where the mean duration of the long tasks is significantly longer than the mean of the short tasks. Such heterogeneous workloads can hurt frameworks with short tasks. In the worst case, all nodes required by a short job might be filled with long tasks, so the job may need to wait a long time (relative to its execution time) to acquire resources.

We note first that random task assignment can work well if the fraction ϕ of long tasks is not very close to 1 and if each node supports multiple slots. For example, in a cluster with S slots per node, the probability that a node is filled with long tasks will be ϕ^S. When S is large (e.g., in the case of multicore machines), this probability is small even with $\phi > 0.5$. If $S = 8$ and $\phi = 0.5$, for example, the probability that a node is filled with long tasks is 0.4%. Thus, a framework with short tasks can still acquire some preferred slots in a short period of time. In addition, the more slots a framework is able to use, the likelier it is that at least k of them are running short tasks.

To further alleviate the impact of long tasks, Mesos can be extended slightly to allow allocation policies to reserve some resources on each node for short tasks. In particular, we can associate a maximum task duration with some of the resources on each node, after which tasks running on those resources are killed. These time limits can be exposed to the frameworks in resource offers, allowing them to choose whether to use these resources. This scheme is similar to the common policy of having a separate queue for short jobs in HPC clusters.

4.9 Framework Incentives

Mesos implements a multi-agent scheduling model, where each framework decides which offers to accept. In such a system, it is important to understand the incentives of entities in the system. In this section, we discuss the incentives of frameworks (and their users) to improve the response times of their jobs.

Short tasks: A framework is incentivized to use short tasks for two reasons. First, it will be able to allocate any resources reserved for short slots. Second, using small tasks minimizes the wasted work if the framework loses a task, either due to revocation or simply due to failures.

Scale elastically: The ability of a framework to use resources as soon as it acquires them—instead of waiting to reach a given minimum allocation—would allow the framework to start (and complete) its jobs earlier. In addition, the ability to scale up and down allows a framework to grab unused resources opportunistically, as it can later release them with little negative impact.

Do not accept unknown resources: Frameworks are incentivized not to accept resources that they cannot use because most allocation policies will count all the resources that a framework owns when making offers.

We note that these incentives align well with our goal of improving utilization. If frameworks use short tasks, Mesos can reallocate resources quickly between them, reducing latency for new jobs and wasted work for revocation. If frameworks are elastic, they will opportunistically utilize all the resources they can obtain. Finally, if frameworks do not accept resources that they do not understand, they will leave them for frameworks that do.

We also note that these properties are met by many current cluster computing frameworks, such as MapReduce and Dryad, simply because using short independent tasks simplifies load balancing and fault recovery.

4.10 Limitations of Partitioned State Scheduling

Although we have shown that partitioned state scheduling works well in a range of workloads relevant to current cluster environments, because scheduling agents often operate over a strict subset of cluster state, their scheduling decisions may be less optimal than that of a scheduling agent in monolithic state scheduling. We have identified three limitations of the partitioned state scheduling architecture:

Fragmentation: When tasks have heterogeneous resource demands, a distributed collection of frameworks may not be able to optimize bin packing as well as a centralized scheduler. However, note that the wasted space due to suboptimal bin packing is bounded by the ratio between the largest task size and the node size. Therefore, clusters running "larger" nodes (e.g., multicore nodes) and "smaller" tasks within those nodes will achieve high utilization even with distributed scheduling.

There is another possible bad outcome if allocation modules reallocate resources in a naïve manner: when a cluster is filled by tasks with small resource requirements, a framework f with large resource requirements may starve, because whenever a small task finishes, f cannot accept the resources freed by it, but other frameworks can. To accommodate frameworks with large per-task resource requirements, allocation modules can support a *minimum offer size* on each slave, and abstain from offering resources on the slave until this amount is free.

Interdependent framework constraints: It is possible to construct scenarios where, because of esoteric interdependencies between frameworks (e.g., certain tasks from two frameworks cannot be colocated), only a single global allocation of the cluster performs well. We argue such scenarios are rare in practice. In the model discussed in this section, where frameworks only have preferences over which nodes they use, we showed that allocations approximate those of optimal schedulers.

Framework complexity: Using resource offers may make framework scheduling more complex. We argue, however, that this difficulty is not onerous. First, whether using Mesos or a centralized scheduler, frameworks need to know their preferences; in a centralized scheduler, the framework needs to express them to the scheduler, whereas in Mesos, it must use them to decide which offers to accept. Second, many scheduling policies for existing frameworks are online algorithms, because frameworks cannot predict task times and must be able to handle failures and stragglers [24, 48, 51]. These policies are easy to implement over resource offers.

4.11 Partitioned State Scheduling Chapter Summary

In this chapter we have presented Partitioned State Scheduling, an approach to multi-agent cluster scheduling that overcomes the scaling and flexibility limitations inherent to Monolithic State Scheduling. We began by discussing the widely adopted technique of statically partitioning cluster state across scheduling agents (SPS), and then introduced Dynamically Partitioned State Scheduling (DPS), in which a centralized scheduling agent adjusts the size of the scheduling domain partitions on the fly. Finally, we analyzed the expected performance of DPS for our target workloads using both a Monte Carlo simulation and statistical models based on simplifying assumptions about workload parameters.

Chapter 5

Mesos, a Dynamically Partitioned State Scheduler

In this chapter, we further present and evaluate Mesos, a real-world implementation of DPS. Mesos scales to clusters of tens of thousands of machines, provides increased framework development flexibility, as well as increased resource utilization.

We begin in Section 5.1 by introducing Mesos and mapping the components of Mesos to the general model for cluster scheduling introduced in Section 2.3, and summarizing the design goals of Mesos. In Sections 5.2, and 5.3, we present details of the design philosophy, architecture, and implementation of Mesos. Then, in Section 5.4, we describe our implementation of several actual Job Managers (i.e., Mesos "Frameworks") that run on Mesos. Finally, in Section 5.5, we describe the experiments we ran to evaluate Mesos in action and present the results.

5.1 Mesos Background and Goals

Mesos was originally conceived after observing Hadoop's use of the Monolithic State Scheduling architecture. Specifically, Hadoop suffered scalability and reliability limitations associated with its Monolithic State Scheduling architecture. For example, Hadoop users frequently complained that a buggy MapReduce job could crash the Hadoop master causing all of the jobs being managed by the master to be lost.

In response to these observations, we originally designed Mesos to provide a number of benefits to practitioners who are used to running Hadoop clusters.

First, even organizations that only use one framework, such as Hadoop, can use Mesos to run multiple instances or multiple versions of that framework in the same cluster. Engineers at Yahoo! and Facebook indicated that this would be a compelling way to isolate production and experimental Hadoop workloads and to roll out new versions of Hadoop [12, 13].

Second, Mesos makes it easier to develop and immediately experiment with new frameworks. The ability to share resources across multiple frameworks frees the developers to build and run *specialized* frameworks targeted at particular problem domains rather than one-size-fits-all abstractions. Frameworks can therefore evolve faster and provide better support for each problem domain.

5.1.1 Dynamically Partitioned Scheduling in Mesos

Mesos implements the DPS architecture. In Mesos we call job managers ***frameworks***, and the functionality of the meta-scheduling agent is performed by the Mesos master.

5.1.2 Goals of Mesos

We can summarize the design goals for the initial version of Mesos as follows:

- Support and demonstrate multi-agent scheduling
- Support fair-sharing meta-scheduling policy
- Increase overall cluster utilization
- Scale to tens of thousands of machines and hundreds of jobs

5.2 Mesos Architecture

In this section, we begin by discussing our design philosophy for Mesos. We then describe the components of Mesos, our resource allocation mechanisms, and how Mesos achieves isolation, scalability, and fault tolerance.

5.2.1 Design Philosophy

Mesos aims to provide a scalable and resilient core for enabling various frameworks *to efficiently share* clusters. Because cluster frameworks are both highly diverse and rapidly evolving, our overriding design philosophy has been to define a minimal interface that enables efficient resource sharing across frameworks, and otherwise push control of task scheduling and execution to the frameworks. Pushing control to the frameworks has two benefits. First, it allows frameworks to implement diverse approaches to various scheduling concerns in the cluster, such as achieving data locality and dealing with faults. It also allows frameworks to evolve these solutions independently. Second, it keeps Mesos simple and minimizes the rate of change required of the system, which makes it easier to keep Mesos scalable and robust.

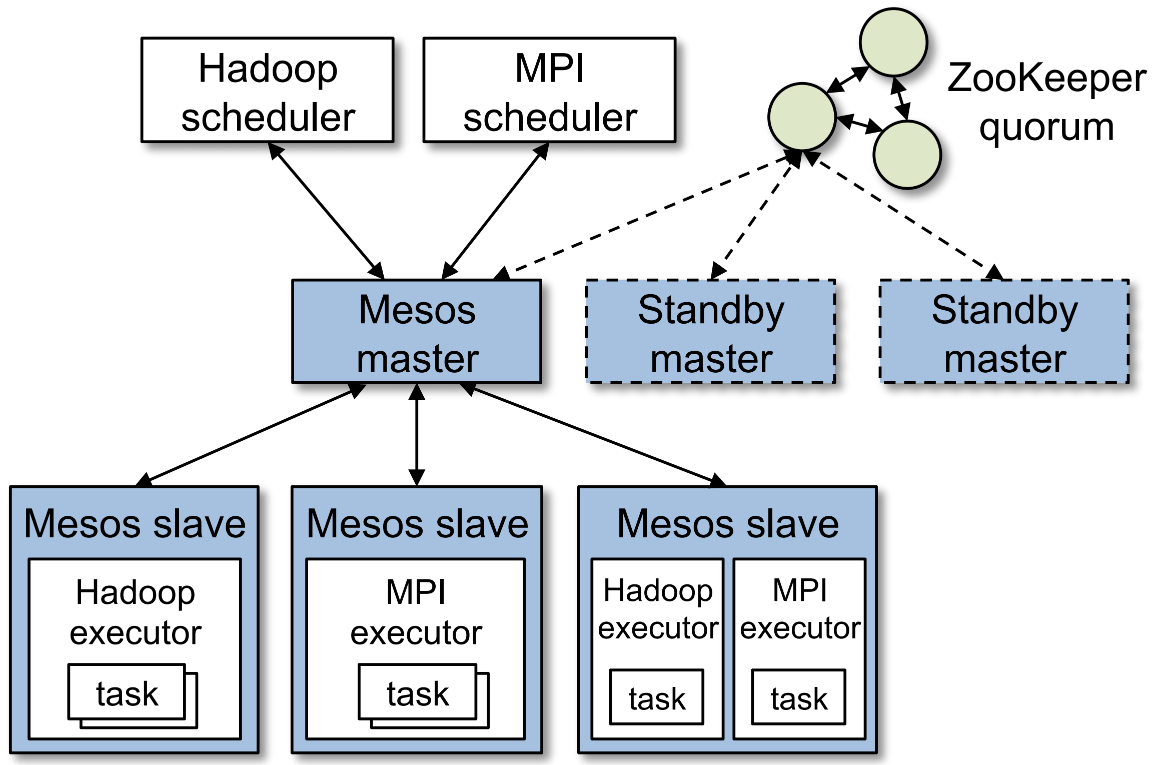


Figure 5.1: *Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).*

Although Mesos provides a low-level interface, we expect higher-level libraries implementing common functionality, such as fault tolerance, to be built on top of it. These libraries would be analogous to library OSes in the exokernel [26]. Putting this functionality in libraries rather than in Mesos allows Mesos to remain small and flexible, and lets the libraries evolve independently.

5.2.2 Architecture Overview

Figure 5.1 shows the main components of Mesos. Mesos consists of a *master* process that manages *slave* daemons running on each cluster node, and *frameworks* that run *tasks* on these slaves.

The master implements fine-grained sharing across frameworks using *resource offers*. Each resource offer is a list of free resources on multiple slaves. The master decides *how many* resources to offer to each framework according to an organizational policy, such as fair sharing or priority. To support a diverse set of inter-framework allocation policies, Mesos lets organizations define their own policies via a pluggable allocation module.

Each framework running on Mesos consists of two components: a *scheduler* that registers with the master to be offered resources, and an *executor* process that is launched on slave

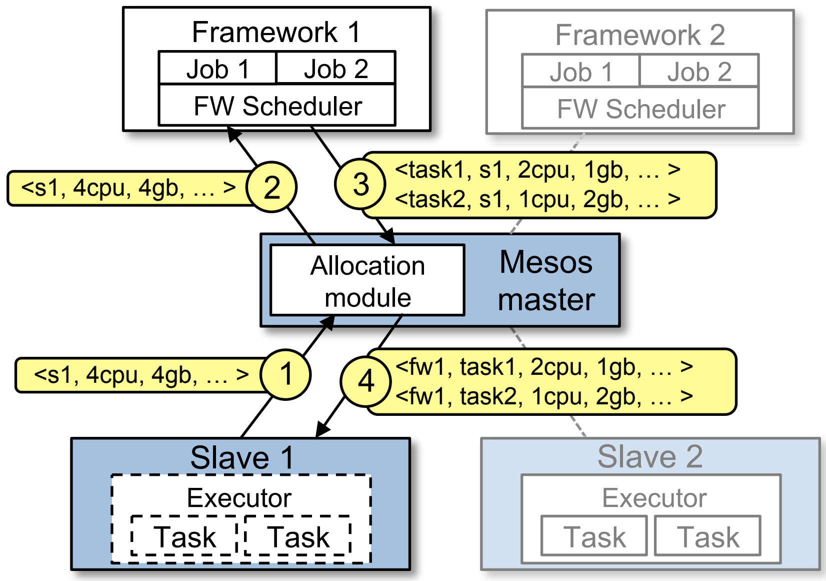


Figure 5.2: *Resource offer example.*

nodes to run the framework's tasks. While the master determines how many resources to offer to each framework, the frameworks' schedulers select *which* of the offered resources to use. When a framework accepts offered resources, it passes Mesos a description of the tasks it wants to launch on them.

Figure 5.2 shows an example of how a framework gets scheduled to run tasks. In Step (1), Slave 1 reports to the Master that it has 4 CPUs and 4 GB of memory free. The Master then invokes the allocation module, which tells it that Framework 1 should be offered all available resources. In Step (2), the Master sends a resource offer describing these resources to Framework 1. In Step (3), Framework 1's scheduler replies to the Master with information about two tasks to run on the Slave, using $\langle 2 \text{ CPUs}, 1 \text{ GB RAM}\rangle$ for the first task, and $\langle 1 \text{ CPUs}, 2 \text{ GB RAM}\rangle$ for the second task. Finally, in Step (4), the Master sends the tasks to the Slave, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted borders). Because 1 CPU and 1 GB of RAM are still free, the allocation module may now offer them to framework 2. In addition, this resource offer process repeats when tasks finish and new resources become free.

To maintain a narrow interface and enable frameworks to evolve independently, Mesos does not *require* frameworks to specify their resource requirements or constraints. Instead, Mesos gives frameworks the ability to *reject* offers. A framework can reject resources that do not satisfy its constraints to wait for ones that do. Thus, the rejection mechanism enables frameworks to support arbitrarily complex resource constraints while keeping Mesos simple and scalable.

One potential challenge with solely using the rejection mechanism to satisfy all framework constraints is efficiency: a framework may have to wait a long time before it receives an offer satisfying its constraints, and Mesos may have to send an offer to many frameworks before one of them accepts it. To avoid this, Mesos also allows frameworks to set *filters*, which are Boolean predicates specifying that a framework will always reject certain resources. For example, a framework might specify a blacklist of nodes it can not run on.

There are two points worth noting. First, filters represent just a performance optimization for the resource offer model, as the frameworks still have the ultimate control to reject any resources that they cannot express filters for and to choose which tasks to run on each node. Second, as we will show in this paper, when the workload consists of fine-grained tasks, such as those in typical MapReduce and Dryad jobs, the resource offer model performs surprisingly well even in the absence of filters. In particular, we have found that a simple policy called delay scheduling [48], in which frameworks wait for a limited time to acquire nodes storing their data, yields nearly optimal data locality with a wait time of 1-5s.

In the rest of this section, we describe how Mesos performs two key functions: resource allocation (§5.2.3) and resource isolation (§5.2.4). We then describe filters and several other mechanisms that make resource offers scalable and robust (§5.2.5). Finally, we discuss fault tolerance in Mesos (§5.2.6) and summarize the Mesos API (§5.2.8).

5.2.3 Resource Allocation

Mesos delegates allocation decisions to a pluggable allocation module, so that organizations can tailor allocation to their needs. So far, we have implemented two allocation modules: one that performs fair sharing based on a generalization of max-min fairness for multiple resources [27] and one that implements strict priorities. Similar policies are used in Hadoop and Dryad [32, 48].

In normal operation, Mesos takes advantage of the fact that most tasks are short, and only reallocates resources when tasks finish. This usually happens frequently enough so that new frameworks acquire their share quickly. For example, if a framework's share is 10% of the cluster, it needs to wait approximately 10% of the mean task length to receive its share.

We leave it up to the allocation module to select the policy for revoking tasks, but describe two related mechanisms here. First, while killing a task has a low impact on many frameworks, such as MapReduce, it is harmful for frameworks with interdependent tasks, such as MPI. We allow these frameworks to avoid being killed by letting allocation modules expose a *guaranteed allocation* to each framework—a quantity of resources that the framework may hold without losing tasks. Frameworks read their guaranteed allocations through an API call. Allocation modules are responsible for ensuring that the guaranteed allocations they provide can all be met concurrently. For now, we have kept the semantics of guaranteed allocations simple: if a framework is below its guaranteed allocation, none of its tasks should be killed, and if it is above, any of its tasks may be killed.

Second, to decide when to trigger revocation, Mesos must know which of the connected frameworks would use more resources if they were offered them. Frameworks indicate their interest in offers through an API call.

5.2.4 Isolation

Mesos provides performance isolation between framework executors running on the same slave by leveraging existing OS isolation mechanisms. Since these mechanisms are platform-dependent, we support multiple isolation mechanisms through pluggable *isolation modules*.

We currently isolate resources using OS container technologies, specifically Linux Containers [10] and Solaris Projects [16]. These technologies can limit the CPU, memory, network bandwidth, and—in new Linux kernels—I/O usage of a process tree. These isolation technologies are not perfect, but using containers is already an advantage over frameworks like Hadoop, where tasks from different jobs simply run in separate processes.

5.2.5 Making Resource Offers Scalable and Robust

Because task scheduling in Mesos is a distributed process, it needs to be efficient and robust to failures. Mesos includes three mechanisms to help with this goal.

First, because some frameworks will always reject certain resources, Mesos lets them short-circuit the rejection process and avoid communication by providing *filters* to the master. We currently support two types of filters: "only offer nodes from list L" and "only offer nodes with at least R resources free". However, other types of predicates could also be supported. Note that unlike generic constraint languages, filters are Boolean predicates that specify whether a framework will reject one bundle of resources on one node, so they can be evaluated quickly on the master. Any resource that does not pass a framework's filter is treated exactly like a rejected resource.

Second, because a framework may take time to respond to an offer, Mesos counts resources offered to a framework towards its allocation of the cluster. This is a strong incentive for frameworks to respond to offers quickly and to filter resources that they cannot use.

Third, if a framework has not responded to an offer for a sufficiently long time, Mesos *rescinds* the offer and re-offers the resources to other frameworks.

5.2.6 Fault Tolerance

Since all the frameworks depend on the Mesos master, it is critical to make the master fault-tolerant. To achieve this, we have designed the master to be *soft state*, so that a new master can completely reconstruct its internal state from information held by the slaves and the framework schedulers. In particular, the master's only state is the list of active slaves, active frameworks, and running tasks. This information is sufficient to compute how many resources each framework is using and run the allocation policy. We run multiple masters in a hot-standby configuration using ZooKeeper [4] for leader election. When the active master fails, the slaves and schedulers connect to the next elected master and repopulate its state.

Aside from handling master failures, Mesos reports node failures and executor crashes to frameworks' schedulers. Frameworks can then react to these failures using *the policies of their choice.*

Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers)	replyToOffer(offerId, tasks)
offerRescinded(offerId)	setNeedsOffers(bool)
statusUpdate(taskId, status)	setFilters(filters)
slaveLost(slaveId)	getGuaranteedShare()
	killTask(taskId)

Executor Callbacks	Executor Actions
launchTask(taskDescriptor)	sendStatus(taskId, status)
killTask(taskId)	

Table 5.1: *Mesos API functions for schedulers and executors.*

Finally, to deal with scheduler failures, Mesos allows a framework to register multiple schedulers such that when one fails, another one is notified by the Mesos master to take over. Frameworks must use their own mechanisms to share state between their schedulers.

5.2.7 Communication and Storage

The Mesos architecture does not impose any storage or communication abstractions on frameworks. Following our minimalist design philosophy, we wish to let frameworks choose their own abstractions, and to let these abstractions evolve independently of Mesos. In a typical installation, we expect that data will be shared through a cluster file system such as HDFS [2].

5.2.8 API Summary

Table 5.1 summarizes the Mesos API. The "callback" columns list functions that frameworks must implement, while "actions" are operations that they can invoke.

5.3 Mesos Implementation

We have implemented Mesos in about 10,000 lines of C++. The system runs on Linux, Solaris and OS X, and supports frameworks written in C++, Java, and Python.

To reduce the complexity of our implementation, we use a C++ library called `libprocess` [8] that provides an actor-based programming model using efficient asynchronous I/O mechanisms (`epoll`, `kqueue`, etc). We also use ZooKeeper [4] to perform leader election.

Mesos can use Linux containers [10] or Solaris projects [16] to isolate tasks. We currently isolate CPU cores and memory. We plan to leverage recently added support for network and I/O isolation in Linux [9] in the future.

We have implemented four frameworks on top of Mesos. First, we have ported three existing cluster computing systems: Hadoop [2], the Torque resource scheduler [43], and the MPICH2 implementation of MPI [21]. None of these ports required changing these frameworks' APIs, so all of them can run unmodified user programs. In addition, a team at UC Berkeley wrote a specialized framework for iterative jobs on top of Mesos called Spark[49, 50], which we use in our evaluation of Mesos and provide an overview of in Section 5.4.4.

A benefit of using C/C++ to implement Mesos is easily interoperability with other languages. In particular, we originally used the Simplified Wrapper and Interface Generator (SWIG) to easily generate interfaces and bindings in Python, and Java. We have since migrated to using the Java and Python native language interfaces to provide the same functionality.

5.3.1 Executor Isolation

Recall from Section 5.2.4, that a good isolation mechanism for Mesos should (a) have low overheads for executor startup and task execution and (b) have the ability to let Mesos change resource allocations dynamically.

Given those constraints, we present possible mechanisms below:

Processes and ulimit Using processes as the "container" for isolation is appealing because processes are a lightweight and portable mechanism. However, `ulimit` and `setrlimit` alone are insufficient for providing aggregate resource limits across process trees (e.g., a process and all of its descendants).

Virtual Machines Virtual machines are an appealing container, however, virtualization imposes I/O overheads [22] that may not be acceptable for data-intensive applications like MapReduce. In addition, VMs take a fairly long time to start up, increasing latency for short lived executors.

Cpusets, Containers, Zones, etc. Modern operating systems are beginning to provide mechanisms to isolate entire process trees. For example, Linux supports cpusets and cgroups for CPU isolation [17], and Linux containers [10] are aimed to provide more comprehensive isolation. These mechanisms tend to be very lightweight and are dynamically configurable while a process is running (similar to ulimit and setrlimit).

For our current implementation, we support Linux containers, Linux processes, or Solaris resource management mechanisms [16]. Solaris provides a relatively advanced set of mechanisms for resource isolation, which allows, for example, one to set cumulative limits on CPU share, resident set size, and OS objects such as threads, on a process tree. A nice property of the Solaris mechanisms is that one can configure, at least for some resources, the ability to let idle resources get used by processes that have reached their limits.

As operating system isolation mechanisms improve, they should only strengthen the performance isolation guarantees that Mesos can provide. We explore how well our current isolation mechanisms work in Section 5.5.

5.4 Mesos Frameworks

5.4.1 Hadoop Port

Porting Hadoop to run on Mesos required relatively few modifications, because Hadoop's fine-grained map and reduce tasks map cleanly to Mesos tasks. In addition, the Hadoop master, known as the JobTracker, and Hadoop slaves, known as TaskTrackers, fit naturally into the Mesos model as a framework scheduler and executor.

To add support for running Hadoop on Mesos, we took advantage of the fact that Hadoop already has a pluggable API for writing job schedulers. We wrote a Hadoop scheduler that connects to Mesos, launches TaskTrackers as its executors, and maps each Hadoop task to a Mesos task. When there are unlaunched tasks in Hadoop, our scheduler first starts Mesos tasks on the nodes of the cluster that it wants to use, and then sends the Hadoop tasks to them using Hadoop's existing internal interfaces. When tasks finish, our executor notifies Mesos by listening for task finish events using an API in the TaskTracker.

We used delay scheduling [48] to achieve data locality by waiting for slots on the nodes that contain task input data. In addition, our approach allowed us to reuse Hadoop's existing logic for re-scheduling of failed tasks and for speculative execution (straggler mitigation).

We also needed to change how map output data is served to reduce tasks. Hadoop normally writes map output files to the local filesystem, then serves these to reduce tasks using an HTTP server included in the TaskTracker. However, the TaskTracker within Mesos runs as an executor, which may be terminated if it is not running tasks. This would make map output files unavailable to reduce tasks. We solved this problem by providing a shared file server on each node in the cluster to serve local files. Such a service is useful beyond Hadoop, to other frameworks that write data locally on each node.

In total, our Hadoop port is 1500 lines of code.

5.4.2 MPI Port

MPI is a language-independent message-passing API used to implement parallel programs on supercomputers and clusters. We ported the popular MPICH2 [21] implementation of MPI to run on Mesos.

MPICH2 is normally implemented as a ring of intercommunicating daemons that run on the cluster nodes. Users invoke a utility called `mpiexec` to submit a program to the daemons, specifying the number of nodes required. Rather than make invasive changes to MPICH2, we created a Mesos framework, written in 200 lines of Python code. that acts as a "wrapper" around `mpiexec` that registers with the Mesos master, dynamically launches an MPICH2 daemon ring of the appropriate size, and runs the standard `mpiexec` to submit the job to these daemons. Our application accepts resource offers until it has enough of them to set up a daemon ring, then executes MPICH2 daemons in its tasks. These daemons in turn execute the user's application.

Note that because MPI uses the MPI daemons to launch more processes, nothing extra needs to be done to ensure subsequently launched processes on the slaves will be properly isolated. The simplicity of this approach is a direct consequence of how the majority of MPI jobs are executed – a job is given a static number of nodes for its entire duration. Since few MPI jobs fork and launch more parallel processes dynamically, we choose not to provide any mechanism for doing so, and we implemented a very simple scheduling policy – accept any offered resources that are large enough to launch the program.

Currently, the wrapper launches MPI jobs conservatively. That is, it never attempts to use more than its guaranteed allocation of the resources to avoid resource revocation.

5.4.3 Torque Port

We have ported the Torque cluster resource manager [43] to run as a framework on Mesos. The framework consists of a Mesos scheduler and executor, written in 360 lines of Python code, that launch and manage different components of Torque. In addition, we modified three lines of Torque source code to allow it *to elastically scale up and down* on Mesos depending on the jobs in its queue. The amount of code to change was very small and we had easy access to the source code due to Torque being open source. Even being unfamiliar with the complicated Torque code base, it took little effort (approximately five hours) to identify the location of the relevant code to change and test the effect of the changes.

After registering with the Mesos master, the framework scheduler configures and launches a Torque server and then periodically monitors the server's job queue. While the queue is empty, the scheduler releases all tasks (down to an optional minimum, which we set to 0) and refuses all resource offers it receives from Mesos. Once a job gets added to Torque's queue (using the standard `qsub` command), the scheduler begins accepting new resource offers. As long as there are jobs in Torque's queue, the scheduler accepts offers as necessary to satisfy the constraints of as many jobs in the queue as possible. On each node where offers are accepted, Mesos launches our executor, which in turn starts a Torque backend daemon and registers it with the Torque server. When enough Torque backend daemons have registered, the torque server will launch the next job in its queue.

Because jobs that run on Torque (e.g., MPI) may not be fault tolerant, Torque avoids having its tasks revoked by not accepting resources beyond its guaranteed allocation.

5.4.4 Spark Framework

Mesos enables the creation of specialized frameworks optimized for workloads for which more general execution layers may not be optimal. To test the hypothesis that simple specialized frameworks provide value, we identified one class of jobs that were found to perform poorly on Hadoop by machine learning researchers at our lab: *iterative jobs*, where a dataset is reused across a number of iterations. In response to this observation, a team of researchers in our lab built a specialized framework called Spark [49, 50] optimized for these workloads.

One example of an iterative algorithm used in machine learning is logistic regression [29]. This algorithm seeks to find a line that separates two sets of labeled data points. The

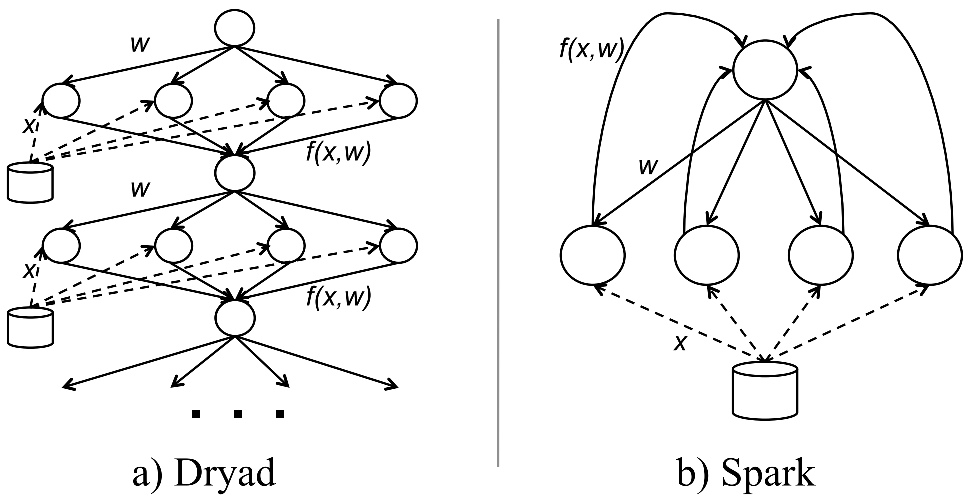


Figure 5.3: *Data flow of a logistic regression job in Dryad vs. Spark. Solid lines show data flow within the framework. Dashed lines show reads from a distributed file system. Spark reuses in-memory data across iterations to improve efficiency.*

algorithm starts with a random line w. Then, on each iteration, it computes the gradient of an objective function that measures how well the line separates the points, and shifts w along this gradient. This gradient computation amounts to evaluating a function $f(x, w)$ over each data point x and summing the results. An implementation of logistic regression in Hadoop must run each iteration as a separate MapReduce job, because each iteration depends on the w computed at the previous one. This imposes overhead because every iteration must re-read the input file into memory. In Dryad, the whole job can be expressed as a data flow DAG as shown in Figure 5.3a, but the data must still must be reloaded from disk at each iteration. Reusing the data in memory between iterations in Dryad would require cyclic data flow.

Spark's execution is shown in Figure 5.3b. Spark uses the long-lived nature of Mesos executors to cache a slice of the dataset in memory at each executor, and then run multiple iterations on this cached data. This caching is achieved in a fault-tolerant manner: if a node is lost, Spark remembers how to recompute its slice of the data.

By building Spark on top of Mesos, the Spark team was able to keep Spark's implementation small (about 1300 lines of code), yet still capable of outperforming Hadoop by 10× for iterative jobs. In particular, using Mesos's API saved the time to write a master daemon, slave daemon, and communication protocols between them for Spark. The main pieces they had to write were a framework scheduler (which uses delay scheduling for locality) and user APIs.

We refer the reader to [49, 50] for more details on Spark.

5.4.5 Elastic Web Server Farm

We built an elastic web server farm framework that takes advantage of Mesos to scale up and down based on external load.

Similar to the Torque framework, the web server farm framework uses a scheduler "wrapper" and executor "wrapper". The scheduler wrapper launches an haproxy [5] load balancer and periodically monitors its web request statistics to decide when to launch or teardown servers. Its only scheduling constraint is that it will launch at most one Apache instance per machine, and then set a filter to stop receiving further offers for that machine. The wrappers are implemented in 250 lines of Python.

5.5 Mesos Evaluation

We evaluated Mesos through a series of experiments on the Amazon Elastic Compute Cloud (EC2). We begin with a macrobenchmark that evaluates how the system shares resources between four workloads, and go on to present a series of smaller experiments designed to evaluate overhead, decentralized scheduling, our specialized framework (Spark), scalability, and failure recovery.

5.5.1 Macrobenchmark

To evaluate the primary goal of Mesos, which is enabling diverse frameworks to efficiently share a cluster, we ran a macrobenchmark consisting of a mix of four workloads:

- A Hadoop instance running a mix of small and large jobs based on the workload at Facebook.
- A Hadoop instance running a set of large batch jobs.
- Spark running a series of machine learning jobs.
- Torque running a series of MPI jobs.

We compared a scenario where the workloads ran as four frameworks on a 96-node Mesos cluster using fair sharing to a scenario where they were each given a static partition of the cluster (24 nodes), and measured job response times and resource utilization in both cases. We used EC2 nodes with four CPU cores and 15 GB of RAM.

We begin by describing the four workloads in more detail, and then present our results.

Macrobenchmark Workloads

Facebook Hadoop Mix Our Hadoop job mix was based on the distribution of job sizes and inter-arrival times at Facebook, reported in [48]. The workload consists of 100 jobs

Bin	Job Type	Map Tasks	Reduce Tasks	# Jobs Run
1	selection	1	NA	38
2	text search	2	NA	18
3	aggregation	10	2	14
4	selection	50	NA	12
5	aggregation	100	10	6
6	selection	200	NA	6
7	text search	400	NA	4
8	join	400	30	2

Table 5.2: *Job types for each bin in our Facebook Hadoop mix.*

submitted at fixed times over a 25-minute period, with a mean inter-arrival time of 14s. Most of the jobs are small (1-12 tasks), but there are also large jobs of up to 400 tasks.[1] The jobs themselves were from the Hive benchmark [7], which contains four types of queries: text search, a simple selection, an aggregation, and a join that gets translated into multiple MapReduce steps. We grouped the jobs into eight bins of job type and size (listed in Table 5.2) so that we could compare performance in each bin. We also set the framework scheduler to perform fair sharing between its jobs, as this policy is used at Facebook.

Large Hadoop Mix To emulate batch workloads that need to run continuously, such as web crawling, we had a second instance of Hadoop run a series of IO-intensive 2400-task text search jobs. A script launched ten of these jobs, submitting each one after the previous one finished.

Spark We ran five instances of an iterative machine learning job on Spark. These were launched by a script that waited two minutes after each job ended to submit the next. The job we used was alternating least squares (ALS), a collaborative filtering algorithm [53]. This job is CPU-intensive but also benefits from caching its input data on each node, and needs to broadcast updated parameters to all nodes running its tasks on each iteration.

Torque / MPI Our Torque framework ran eight instances of the `tachyon` raytracing job [45] that is part of the SPEC MPI2007 benchmark. Six of the jobs ran small problem sizes and two ran large ones. Both types used 24 parallel tasks. We submitted these jobs at fixed times to both clusters. The `tachyon` job is CPU-intensive.

Macrobenchmark Results

A successful result for Mesos would show two things: that Mesos achieves higher utilization than static partitioning, and that jobs finish at least as fast in the shared cluster

[1]We scaled down the largest jobs in [48] to have the workload fit a quarter of our cluster size.

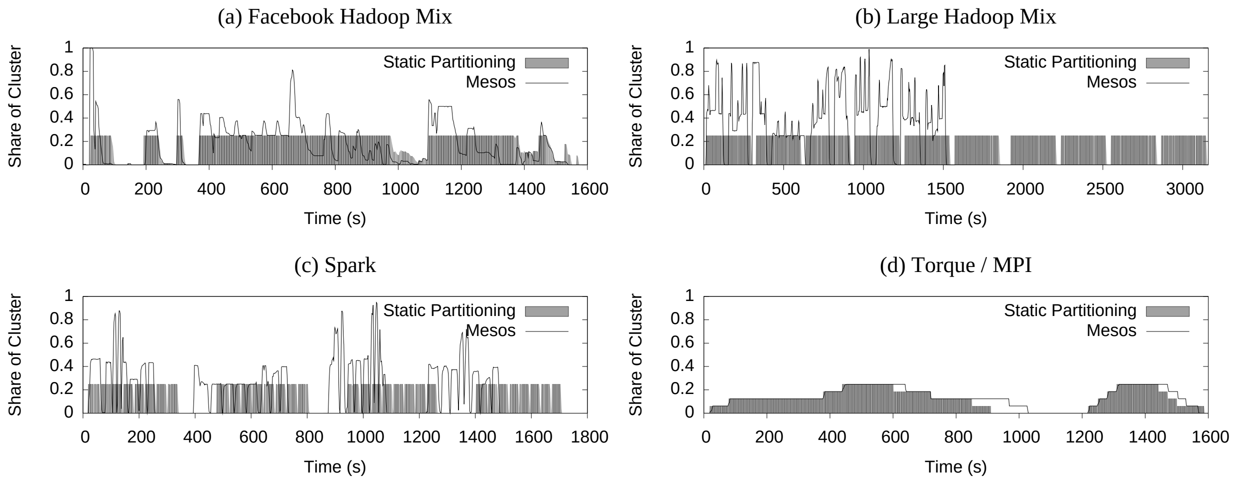


Figure 5.4: *Comparison of cluster shares (fraction of CPUs) over time for each of the frameworks in the Mesos and static partitioning macrobenchmark scenarios. On Mesos, frameworks can scale up when their demand is high and that of other frameworks is low, and thus finish jobs faster. Note that the plots' time axes are different (e.g., the large Hadoop mix takes 3200s with static partitioning).*

as they do in their static partition, and possibly faster due to gaps in the demand of other frameworks. Our results show both effects, as detailed below.

We show the fraction of CPU cores allocated to each framework by Mesos over time in Figure 5.5. We see that Mesos enables each framework to scale up during periods when other frameworks have low demands, and thus keeps cluster nodes busier. For example, at time 350, when both Spark and the Facebook Hadoop framework have no running jobs and Torque is using 1/8 of the cluster, the large-job Hadoop framework scales up to 7/8 of the cluster. In addition, we see that resources are reallocated rapidly (e.g., when a Facebook Hadoop job starts around time 360) due to the fine-grained nature of tasks. Finally, higher allocation of nodes also translates into increased CPU and memory utilization (by 10% for CPU and 17% for memory), as shown in Figure 5.6.

A second question is how much better jobs perform under Mesos than when using a statically partitioned cluster. We present this data in two ways. First, Figure 5.4 compares the resource allocation over time of each framework in the shared and statically partitioned clusters. Shaded areas show the allocation in the statically partitioned cluster, while solid lines show the share on Mesos. We see that the fine-grained frameworks (Hadoop and Spark) take advantage of Mesos to scale up beyond 1/4 of the cluster when global demand allows this, and consequently finish bursts of submitted jobs faster in Mesos. At the same time, Torque achieves roughly similar allocations and job durations under Mesos (with some differences explained later).

Second, Tables 5.3 and 5.4 show a breakdown of job performance for each framework. In Table 5.3, we compare the aggregate performance of each framework, defined as the sum

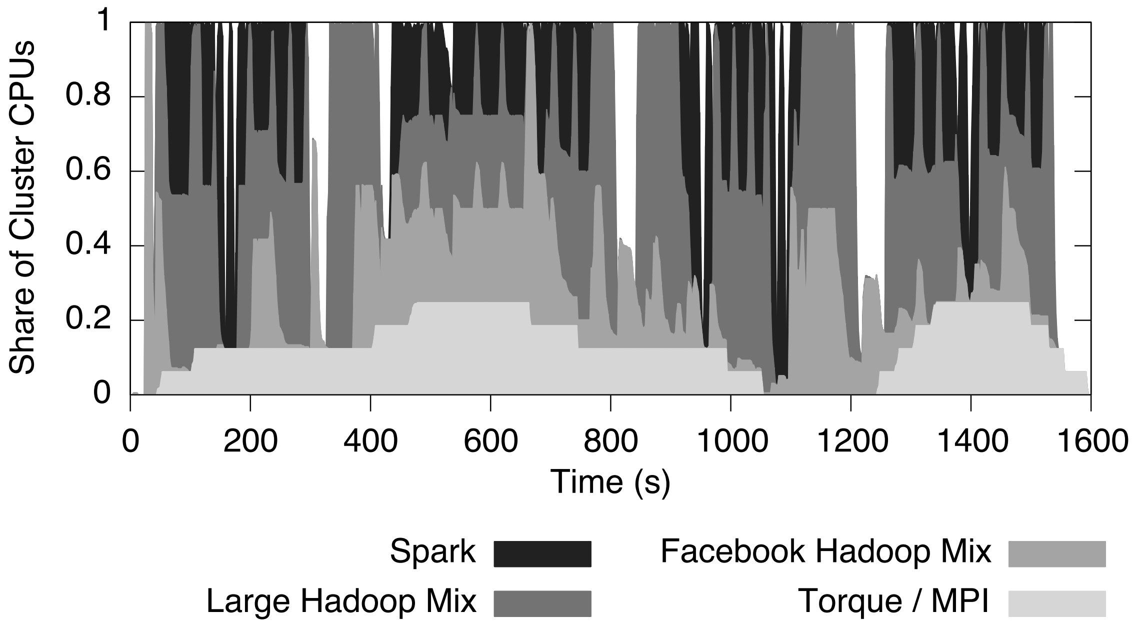


Figure 5.5: *Framework shares on Mesos during the macrobenchmark. By pooling resources, Mesos lets each workload scale up to fill gaps in the demand of others. In addition, fine-grained sharing allows resources to be reallocated in tens of seconds.*

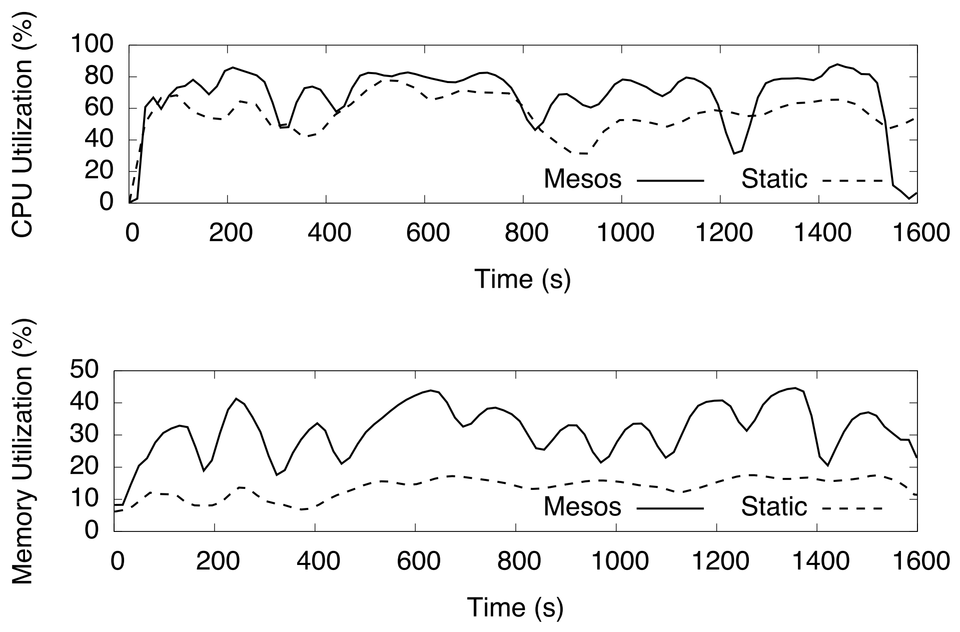


Figure 5.6: *Average CPU and memory utilization over time across all nodes in the Mesos cluster vs. static partitioning.*

Framework	Sum of Exec Times w/ Static Partitioning (s)	Sum of Exec Times with Mesos (s)	Speedup
Facebook Hadoop Mix	7235	6319	**1.14**
Large Hadoop Mix	3143	1494	**2.10**
Spark	1684	1338	**1.26**
Torque / MPI	3210	3352	**0.96**

Table 5.3: *Aggregate performance of each framework in the macrobenchmark (sum of running times of all the jobs in the framework). The speedup column shows the relative gain on Mesos.*

of job running times, in the static partitioning and Mesos scenarios. We see the Hadoop and Spark jobs as a whole are finishing faster on Mesos, while Torque is slightly slower. The framework that gains the most is the large-job Hadoop mix, which almost always has tasks to run and fills in the gaps in demand of the other frameworks; this framework performs 2x better on Mesos.

Table 5.4 breaks down the results further by job type. We observe two notable trends. First, in the Facebook Hadoop mix, the smaller jobs perform worse on Mesos. This is due to an interaction between the fair sharing performed by Hadoop (among its jobs) and the fair sharing in Mesos (among frameworks): During periods of time when Hadoop has more than 1/4 of the cluster, if any jobs are submitted to the other frameworks, there is a delay before Hadoop gets a new resource offer (because any freed up resources go to the framework farthest below its share), so any small job submitted during this time is delayed for a long time relative to its length. In contrast, when running alone, Hadoop can assign resources to the new job as soon as any of its tasks finishes. This problem with fair sharing is also seen in networks [44], and could be mitigated by running the small jobs on a separate framework or using a different allocation policy (e.g., using lottery scheduling instead of offering all freed resources to the framework with the lowest share).

Lastly, Torque is the only framework that performed worse, on average, on Mesos. The large `tachyon` jobs took on average two minutes longer, while the small ones took 20s longer. Some of this delay is due to Torque having to wait to launch 24 tasks on Mesos before starting each job, but the average time this takes is 12s. We believe that the rest of the delay is due to stragglers (slow nodes). In our standalone Torque run, we saw two jobs take about 60s longer to run than the others (Fig. 5.4d). We discovered that both of these jobs were using a node that performed slower on single-node benchmarks than the others (in fact, Linux reported 40% lower bogomips on it). Because `tachyon` hands out equal amounts of work to each node, it runs as slowly as the slowest node.

Framework	Job Type	Exec Time w/ Static Partitioning (s)	Avg. Speedup on Mesos
Facebook Hadoop Mix	selection (1)	24	**0.84**
	text search (2)	31	**0.90**
	aggregation (3)	82	**0.94**
	selection (4)	65	**1.40**
	aggregation (5)	192	**1.26**
	selection (6)	136	**1.71**
	text search (7)	137	**2.14**
	join (8)	662	**1.35**
Large Hadoop Mix	text search	314	**2.21**
Spark	ALS	337	**1.36**
Torque / MPI	small tachyon	261	**0.91**
	large tachyon	822	**0.88**

Table 5.4: *Performance of each job type in the macrobenchmark. Bins for the Facebook Hadoop mix are in parentheses.*

5.5.2 Overhead

To measure the overhead Mesos imposes when a single framework uses the cluster, we ran two benchmarks using MPI and Hadoop on an EC2 cluster with 50 nodes, each with 2 CPU cores and 6.5 GB RAM. We used the High-Performance LINPACK [18] benchmark for MPI and a WordCount job for Hadoop, and ran each job three times. The MPI job took on average 50.9s without Mesos and 51.8s with Mesos, while the Hadoop job took 160s without Mesos and 166s with Mesos. In both cases, the overhead of using Mesos was less than 4%.

5.5.3 Data Locality through Delay Scheduling

In this experiment, we evaluated how Mesos' resource offer mechanism enables frameworks to control their tasks' placement, and in particular, data locality. We ran 16 instances of Hadoop using 93 EC2 nodes, each with four CPU cores and 15 GB RAM. Each node ran a map-only scan job that searched a 100 GB file spread throughout the cluster on a shared HDFS file system and outputted 1% of the records. We tested four scenarios: giving each Hadoop instance its own 5-6 node static partition of the cluster (to emulate organizations that use coarse-grained cluster sharing systems), and running all instances on Mesos using either no delay scheduling, 1s delay scheduling or 5s delay scheduling.

Figure 5.7 shows averaged measurements from the 16 Hadoop instances across three runs of each scenario. Using static partitioning yields very low data locality (18%) because the Hadoop instances are forced to fetch data from nodes outside their partition. In contrast, running the Hadoop instances on Mesos improves data locality, even without delay scheduling, because each Hadoop instance has tasks on more nodes of the cluster (there

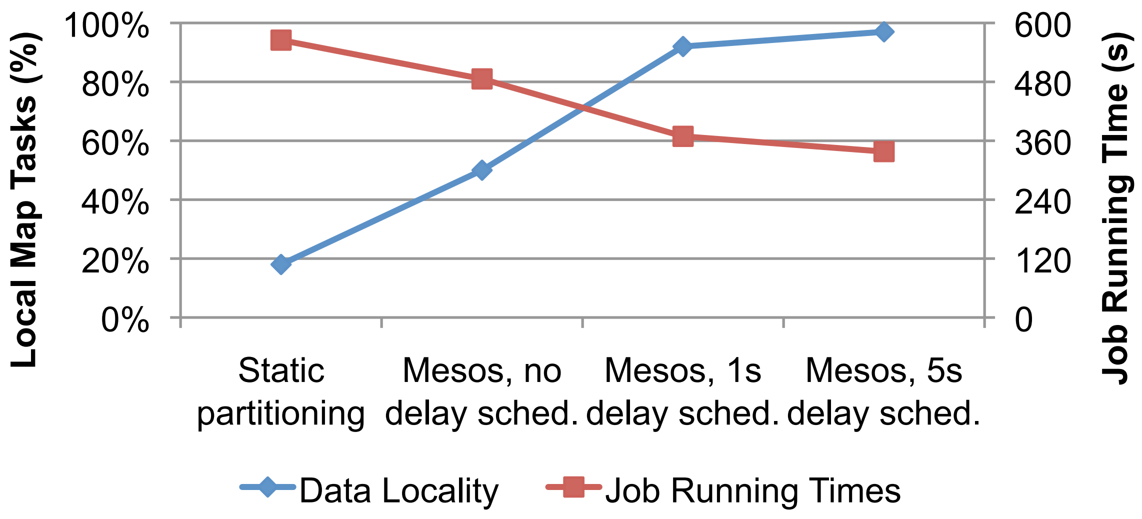


Figure 5.7: *Data locality and average job durations for 16 Hadoop instances running on a 93-node cluster using static partitioning, Mesos, or Mesos with delay scheduling.*

are four tasks per node), and can therefore access more blocks locally. Adding a 1-second delay brings locality above 90%, and a 5-second delay achieves 95% locality, which is competitive with running one Hadoop instance alone on the whole cluster. As expected, job performance improves with data locality: jobs run 1.7x faster in the 5s delay scenario than with static partitioning.

5.5.4 Spark Framework

We evaluated the benefit of running iterative jobs using the specialized Spark framework developed on top of Mesos (Section 5.4.4) over the general-purpose Hadoop framework. We used a logistic regression job implemented in Hadoop by machine learning researchers in our lab, and wrote a second version of the job using Spark. We ran each version separately on 20 EC2 nodes, each with four CPU cores and 15 GB RAM. Each experiment used a 29 GB data file and varied the number of logistic regression iterations from 1 to 30 (see Figure 5.8).

With Hadoop, each iteration takes 127s on average, because it runs as a separate MapReduce job. In contrast, with Spark, the first iteration takes 174s, but subsequent iterations only take about 6 seconds, leading to a speedup of up to 10x for 30 iterations. This happens because the cost of reading the data from disk and parsing it is much higher than the cost of evaluating the gradient function computed by the job on each iteration. Hadoop incurs the read/parsing cost on each iteration, while Spark reuses cached blocks of parsed data and only incurs this cost once. The longer time for the first iteration in Spark is due to the use of slower text parsing routines.

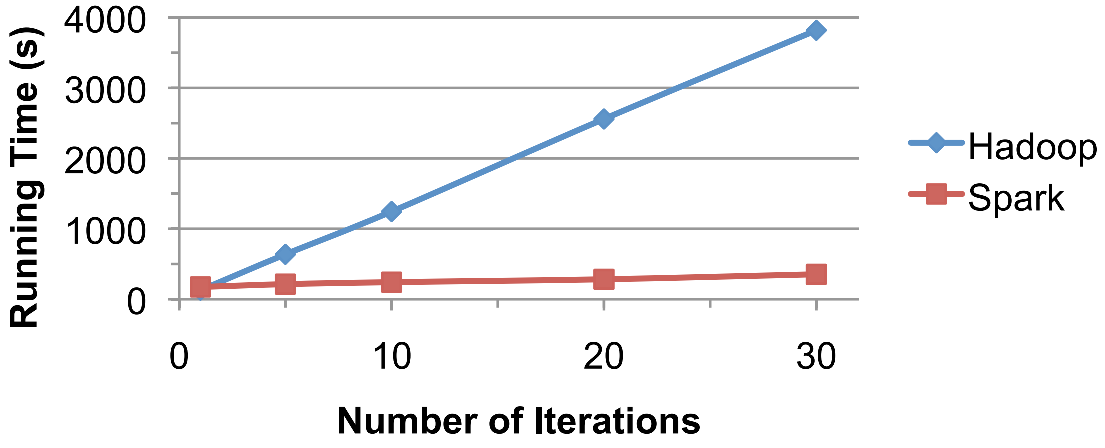


Figure 5.8: *Hadoop and Spark logistic regression running times.*

5.5.5 Elastic Web Farm

To demonstrate an interactive framework dynamically scaling on Mesos we ran an elastic web farm on Mesos. We used HTTPerf [37] to generate increasing and then decreasing load on the web farm. As the average load on each server reaches 150 sessions/second, the elastic web farm framework signals to Mesos that it is willing to accept more resources and launches another Apache instance. We ran experiments using four EC2 instances with eight CPU cores and 6.5 GB RAM. Figure 5.9 shows the web farm dynamically adapts the number of web servers to the offered load (sessions/second at the load balancer) to ensure that the load at each web server remains at or below 150 sessions/sec. The brief drops in sessions per second at the load balancer were due to limitations in the current `haproxy` implementation, which required the framework to restart `haproxy` to increase or decrease the number of Apache servers.

5.5.6 Mesos Scalability

To evaluate Mesos' scalability, we emulated large clusters by running up to 50,000 slave daemons on 99 Amazon EC2 nodes, each with eight CPU cores and six GB RAM. We used one EC2 node for the master and the rest of the nodes to run slaves. During the experiment, each of 200 frameworks running throughout the cluster continuously launches tasks, starting one task on each slave that it receives a resource offer for. Each task sleeps for a period of time based on a normal distribution with a mean of 30 seconds and standard deviation of 10s, and then ends. Each slave runs up to two tasks at a time.

Once the cluster reached steady-state (i.e., the 200 frameworks achieve their fair shares and all resources were allocated), we launched a test framework that runs a single ten second task and measured how long this framework took to finish. This allowed us to calculate the extra delay incurred over 10s due to having to register with the master, wait for a resource

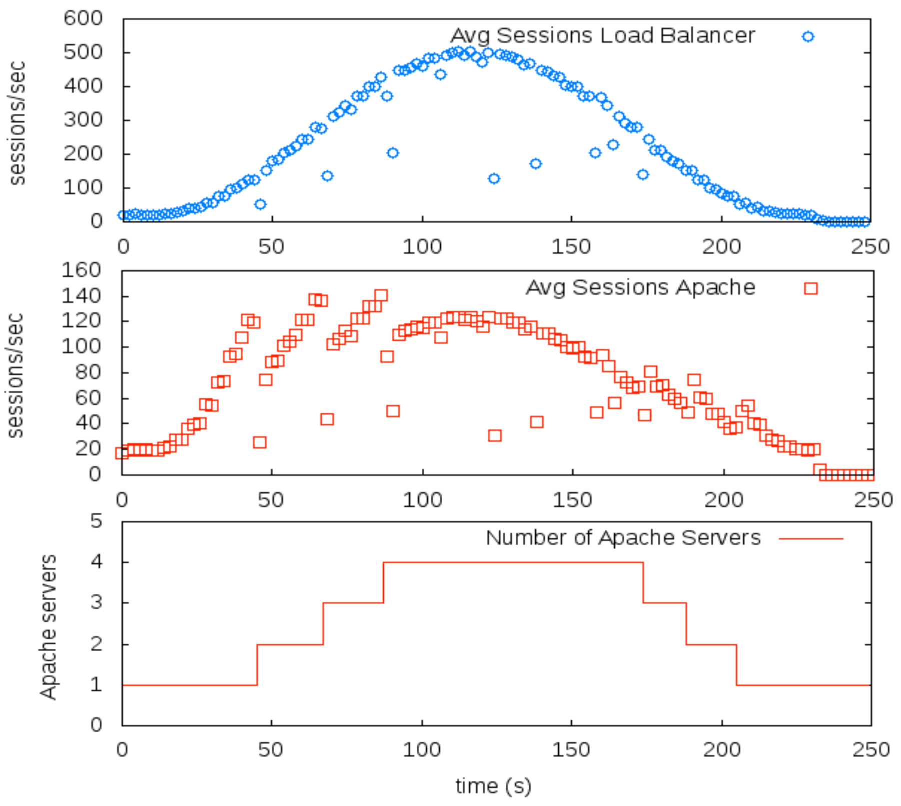


Figure 5.9: *The average session load on the load balancer, the average number of sessions on each web server, and the number of web servers running over time.*

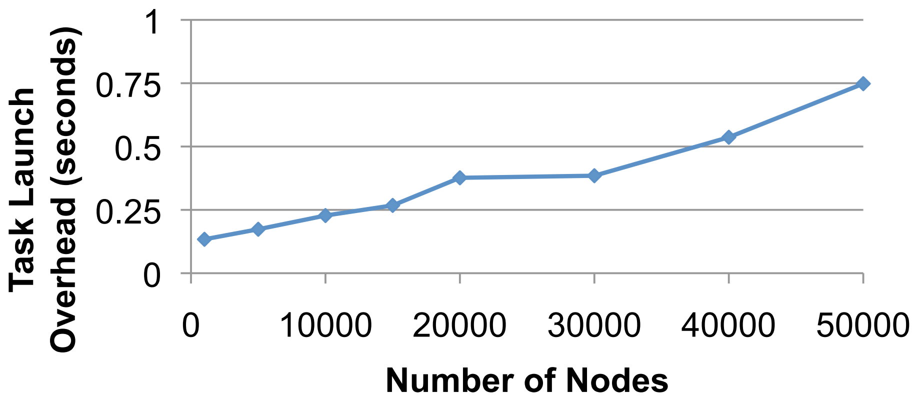


Figure 5.10: *Mesos master's scalability versus number of slaves.*

offer, accept it, wait for the master to process the response and launch the task on a slave, and wait for Mesos to report the task as finished.

We plot this extra delay in Figure 5.10, showing averages of five runs. We observe that the overhead remains small (less than one second) even at 50,000 nodes. In particular, this overhead is much smaller than the average task and job lengths in data center workloads (see Section 2.2). Because Mesos was also keeping the cluster fully allocated, this indicates that the master kept up with the load placed on it. Unfortunately, the EC2 virtualized environment limited scalability beyond 50,000 slaves, because at 50,000 slaves the master was processing 100,000 packets per second (in+out), which has been shown to be the current achievable limit on EC2 [15].

5.5.7 Failure Recovery

To evaluate recovery from master failures, we conducted an experiment with 200 to 4000 slave daemons on 62 EC2 nodes with four cores and 15 GB RAM. We ran 200 frameworks that each launched 20-second tasks, and two Mesos masters connected to a 5-node ZooKeeper quorum. We synchronized the two masters' clocks using NTP and measured the mean time to recovery (MTTR) after killing the active master. The MTTR is the time for all of the slaves and frameworks to connect to the second master. In all cases, the MTTR was between four and eight seconds, with 95% confidence intervals of up to 3s on either side.

5.5.8 Performance Isolation

As discussed in Section 5.2.4, Mesos leverages existing OS isolation mechanism to provide performance isolation between different frameworks' tasks running on the same slave. While these mechanisms are not perfect, a preliminary evaluation of Linux Containers [10] shows promising results. In particular, using Containers to isolate CPU usage between a MediaWiki web server (consisting of multiple Apache processes running PHP) and a "hog" application (consisting of 256 processes spinning in infinite loops) shows on average only a 30% increase in request latency for Apache versus a 550% increase when running without Containers. We refer the reader to [36] for a fuller evaluation of OS isolation mechanisms.

5.6 Mesos Chapter Summary

In this chapter, we described and evaluated Mesos, our real world implementation of a DPS Cluster Scheduling system. We showed the flexibility benefits of Mesos by easily porting a variety of existing cluster frameworks to run on of it. Additionally, we showed Mesos scaling to 50,000 nodes. Finally, we showed the utilization benefits of Mesos.

In the next chapter we motivate and introduce an alternative cluster scheduling architecture called Replicated State Scheduling that represents the next evolutionary step in our taxonomy of cluster scheduling architectures.

Chapter 6

Replicated State Scheduling

In this Chapter, we motivate, present, and evaluate an alternative to Partitioned State Scheduling called Replicated State Scheduling (RSS) that was proposed by researchers at Google [11], in which scheduling domains are allowed to overlap. Allowing scheduling domains to overlap allows scheduling agents read access to the entirety of cluster state while they make their scheduling decisions. However, the cost of this additional flexibility for the scheduling agents is the overhead of resolving conflicts that result from concurrent writes to cluster state.

The rest of this chapter is structured as follows. First, in Section 6.1, we motivate and describe Replicated State Scheduling. Then in Section 6.2, we detail the RSS architectural components, and a number of alternatives for minimizing and resolving scheduling transaction conflicts. In Section 6.3 we extend our Monte Carlo simulation framework to evaluate the scalability of RSS and measure the cost of scheduling agent interference (i.e., failed transactions). We also compare and contrast partitioned and replicated state scheduling.

6.1 Replicated State Scheduling

To motivate Replicated State Scheduling, we first revisit the Partitioned State Scheduling approach we introduced in chapter 4. One of the major drawbacks of Partitioned State Scheduling is that scheduling domains must be selected before the scheduling agent performs its task-resource assignments, thereby potentially restricting the "goodness of fit" that might be achieved by the scheduling agent in its task-resource assignments. As we discussed in Section 4.2.2, that approach is in many ways analogous to pessimistic concurrency control in databases. Now we will look at an alternate approach, which is analogous to optimistic concurrency control in databases, where scheduling domains are allowed to overlap each other.

The original design of Mesos, and its choice to use PSS overlooked the growing need of certain class of scheduling agents. Namely, those responsible for placing long running tasks

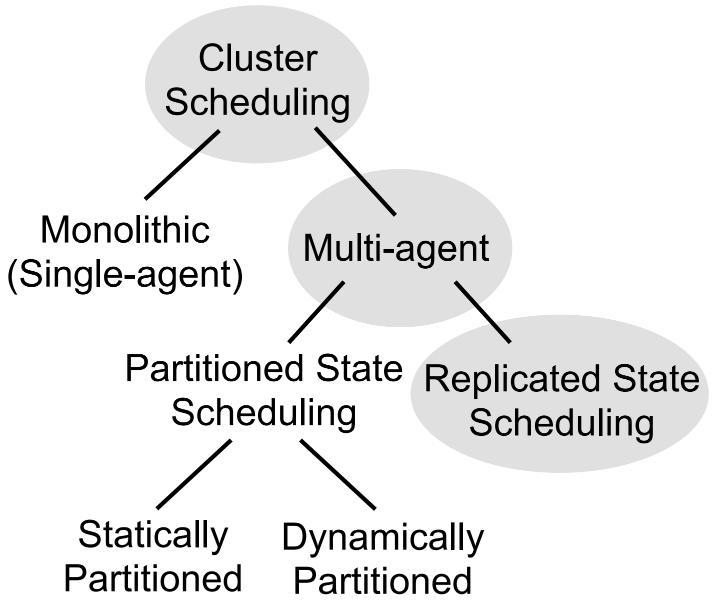

Cluster Scheduling
Monolithic (Single-agent)
Multi-agent
Partitioned State Scheduling
Replicated State Scheduling
Statically Partitioned
Dynamically Partitioned

Figure 6.1: *Our cluster scheduler taxonomy with Replicated State Scheduling (RSS) highlighted.*

that provide low latency service request interfaces (such as web applications, memory or disk based storage systems, or database services like key value stores or RDBMSs). These scheduling agents attempt to optimize task-resource assignments (e.g., for failure recovery or to withstand planned resource outages like upgrades to machines). To do so, they typically:

- Treat all of the cell as their scheduling domain (e.g., to spread tasks across racks/machines and maintain statistical models of failure scenarios)
- Have long scheduling decision times due to complex scheduling policies (e.g., linear algebra, convex optimization, constraint matching, calendaring)

We now present a natural extension of PSS called Replicated State Scheduling (RSS), a new multi-agent scheduling architecture in which all scheduling agents operate over the same scheduling domain, i.e., all job managers maintain their own full private copy of cluster state. Figure 6.1 shows where RSS fits into the taxonomy of architectures we introduced in Chapter 2.

6.2 Overview of RSS Architecture

In RSS, there is one *common cluster state* maintained by the meta-scheduling agent, that acts as a resilient master copy of the resource allocations in the cluster. In addition, each job manager maintains its own *private cluster state*, which it synchronizes with common cluster state before making each job scheduling decision, i.e., creating a job transaction. Figure 6.2 is a conceptual diagram of RSS containing the components of the architecture.

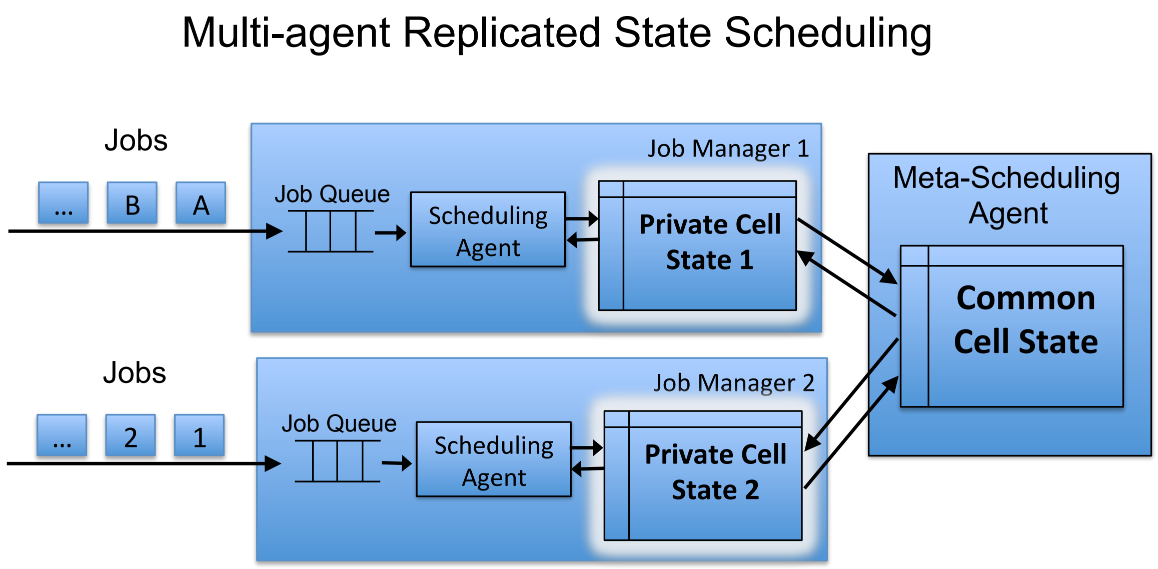


Figure 6.2: *A conceptual diagram showing Replicated State scheduling. Each Job Manager now maintains its own private copy of cluster state, and the meta-scheduling agent is responsible for managing updates between private and common cluster states according to a meta-scheduling policy.*

6.2.1 Role of the Job Manager

Remember from Section 2.3.2 that a *job transaction* contains a list of task-resource assignments. We explore alternative semantics for transaction and conflict detection in detail in Section 6.2.2, but first we will describe the scheduling process from a job manager's perspective. The following is a detailed description of the steps taken by a job manager that constitute the *job transaction lifecycle* in an RSS system:

1. If job queue is not empty, remove next job from job queue.
2. **Sync:** Begin a transaction by synchronizing private cluster state with common cluster state.
3. **Schedule:** Engage scheduling agent to attempt to create task-resource assignments for all tasks in job, modifying private cluster state in the process.
4. **Submit:** Attempt to commit job transaction (i.e., all task-resource assignments for the job) from private cluster state back to common cluster state. Job transaction can succeed or fail.
5. Record which task-resource assignments were successfully committed to common cluster state.
6. If any tasks in job remain unscheduled—either because no suitable resources were

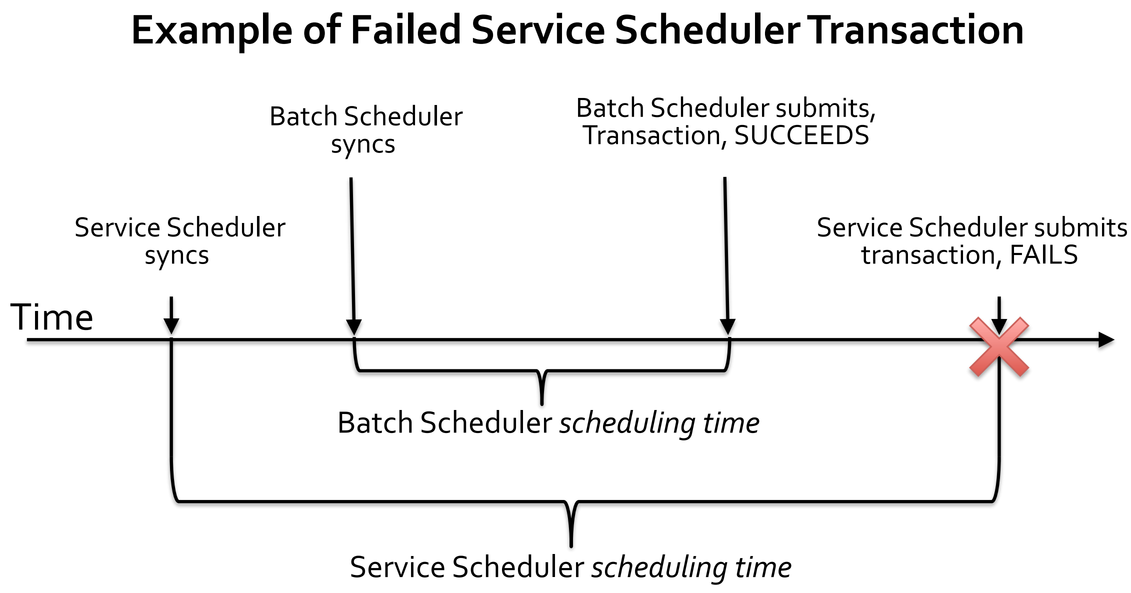

Example of Failed Service Scheduler Transaction
Batch Scheduler syncs
Batch Scheduler submits, Transaction, SUCCEEDS
Service Scheduler syncs
Service Scheduler submits transaction, FAILS
Time
Batch Scheduler *scheduling time*
Service Scheduler *scheduling time*

Figure 6.3: *A timeline showing two overlapping transaction lifecycles leading to a transaction conflict; one from a Batch Scheduler and the other from a Service Scheduler. The schedulers issue transactions; here the Batch Scheduler makes decisions quickly while the Service Scheduler is slower. Consequently some of the Service Scheduler's transactions fail, and need to be retried.*

found for the task during the "schedule" stage or the task-resource assignment experienced a—insert job back into job queue to be handled again in a future transaction

A job manager repeats the above steps for each job scheduling attempt as long as there are jobs remaining to be scheduled. We call this the *job scheduling loop*. All job managers in the system execute their job scheduling loops concurrently. During the "schedule" phase, each job manager can make decisions according to their potentially out-of-date snapshot of the entire cluster, i.e., their private cluster state. Also, each has complete freedom to claim any cluster resources that are marked as available in their private cluster state by creating task-resource assignments. The creation of a task-resource assignment represents an edit to a row of cluster state. If the resources on the same machine are claimed by two job managers concurrently, i.e., transactions from both job managers contain a task-resource assignment with a common machine ID, then the two task-resource assignments may conflict, depending on the conflict detection mode in use. We will discuss conflict modes in detail in Section 6.2.2 below.

6.2.2 Role of the meta-scheduling agent

In RSS, the meta-scheduling agent plays a slightly different role than it does in DPS. Because scheduling agents are allowed to operate over the same scheduling domains concurrently, the meta-scheduling agent must detect and reject transactions that conflict. In DPS, on the other hand, the meta-scheduling agent actively ensures that scheduling domains do not overlap, thus conflicts cannot occur. The meta-scheduling agent can be used to enforce fairness or priority policies in both architectures, but in RSS action is taken passively in response to the submission of job transactions. Such transactions can be rejected either due to a conflict resulting from concurrency or a policy violation. This is in contrast to the meta-scheduling agent in PSS which actively controls and modifies scheduling domains as partitions of cluster state.

One could envision supporting resource offers in a RSS scheduler or perhaps extending Mesos to use RSS by having the Mesos master send "replicated" resource offers, i.e., the same resource offer to multiple Mesos frameworks. At the extreme end of the spectrum, Mesos would be sending a copy of the entire available state of the cluster to all frameworks in parallel. The resulting system would be very similar to Omega, the RSS system we explore here in collaboration with Google[1].

Responsibilities of the meta-scheduling agent:

- Attempt to execute transactions submitted by job managers according to transaction mode settings
- Detect conflicts according to conflict detection semantics and policies
- Enforce meta-scheduling policies

For each job transaction submitted, an RSS meta-scheduling agent performs the following:

1. reject task-resource assignments that would violate policies
2. reject task-resource assignments that conflict with previously accepted transactions
3. reject job all task-resource assignments in a job transaction if using all-or-nothing transaction semantics and at least one task-resource assignment was rejected

Conflict Detection Semantics

We support two types of conflict detection for use in the submit phase of the transaction lifecycle: *machine-granularity* and *resource-granularity.* First, with machine-granularity conflict detection, we reject a task-resource assignment if the row of cluster state representing the machine referenced by the assignment has has had any of its columns representing available resources decremented by the task-resource assignment of another transaction since the sync phase of the current transaction. Note that in this mode, an edit to the machine in which resources are freed on a machine, which happens when a task ends, does not cause a conflict. The advantages of this mode are simplicity and ease of implementation since sequence-numbers can be used to track cluster state row-edits. Alternately, resource-granularity conflict detection is slightly more sophisticated, and will only reject a task-resource assignment if it causes the row of cluster state to enter an invalid state, e.g., if

[1]See [42] for the full paper

it would try to reserve more CPUs for a task than the machine currently has available, which would result in that row/column of cluster state containing a negative number, since a machine can't have a negative number of available CPUs.

Transaction Granularity Semantics

We define two types of transaction granularities: *all-or-nothing* and *incremental.* All-or-nothing transactions are atomic. That is, if a single task-resource assignment conflicts then none of the task-resource assignments in the transaction are applied to common cluster state. Alternately, with incremental transactions, each task-resource assignment can fail independent of all others. We say a *transaction succeeds* if none of its constituent task-resource assignments conflict, and we say the *transaction fails* otherwise.

Discussion

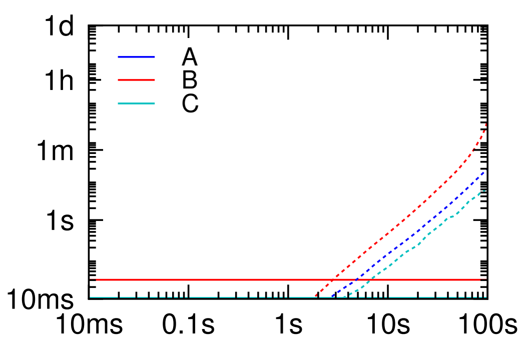
RSS offers even more opportunities for parallelism than DPS because now, not only can scheduling decisions be made in parallel, but scheduling decisions can be made in parallel *over the same set of resources.* For example, assume job manager A assigns task T_A to machine M, and job manager B concurrently assigns task T_B to machine M, where T_A and T_B each require 1 CPU and 1 GB mem. Also assume that at the beginning of both transactions M has 5 CPUs and 5 GB of mem currently available. Then using RSS with resource-granularity conflict detection, both task-resource assignments can succeed. Whereas in DPS, though job managers A and B could make task-resource assignments concurrently, each using their respective scheduling domains, they would still have to take turns accessing machine M serially since scheduling domains can't overlap. The primary disadvantage of RSS is in the form of wasted scheduling work due to failed transactions.

The performance of RSS architectures is ultimately determined by the frequency at which transactions fail and the costs of such failures. Next, we present an evaluation of such costs using an extension of our Monte Carlo simulator.

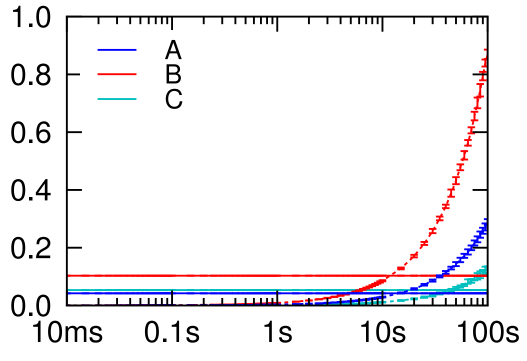
6.3 Monte Carlo RSS Simulation

We modified the Monte Carlo simulator used in Chapters 2 and 4 to simulate Omega-style RSS cluster scheduling.

Our setup here is similar to the setup of our Monte Carlo simulations of MSS and DPS. We again simulate one scheduler handling jobs labeled service, and one handling jobs labeled batch. Both job managers sit in their job scheduling loops. As long as a job is present in their job queue, a job manager takes the next available job from the queue, synchronizes its private cluster state with common cluster state, schedules the tasks of the job based on private cluster state. Then, after waiting for its particular value of scheduling decision time (we vary this value for the service scheduler only in our experiments), if at least one



(a) *Job wait time.*



(b) *Scheduler busy time fraction.*

Figure 6.4: *Results of simulation of 7 days running Replicated State Scheduler (Omega): performance as a function of job time$_{service}$ for clusters A, B, and C (see Table 3.1 for description of each cluster). Solid lines are Batch Schedulers, dotted lines are Service schedulers.*

task-resource assignment was created, the job manager submits the transaction to common cluster state.

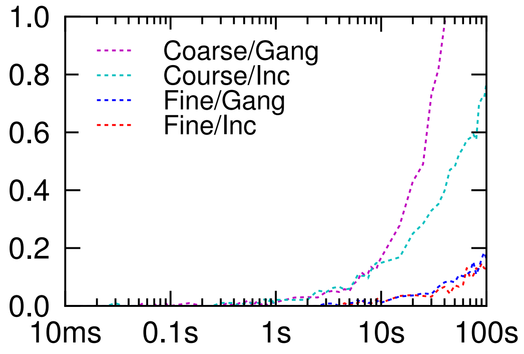
Figure 6.4a shows the results of a 7 day run of our Omega Monte Carlo simulator for clusters A, B, and C (see Table 3.1 for description of each cluster) using resource-granularity conflict detection and incremental transaction semantics. We can see that the average job wait times for the Omega approach are comparable to those from Mesos (Figure 4.4a) and multi-path monolithic (Figure 3.3a). This suggests that the negative impact of interference is limited, and confirmed by the graph of scheduler busy time fraction (Figure 6.4b).

We also ran experiments showing the impact of using machine-granularity conflict detection. Implementing this was straightforward, using a simple sequence number for each machine cluster state. As Figure 6.5 shows, this change results in a steep increase in conflict fraction, and consequently, scheduler busy time fraction, thus seriously impacting scalability.

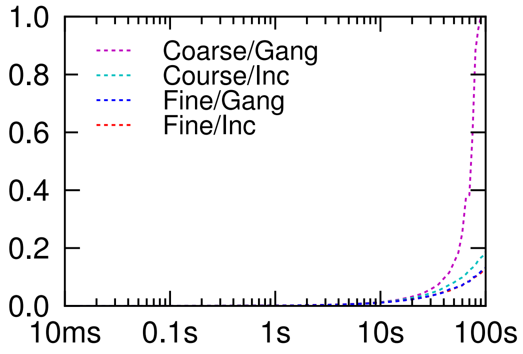
Figure 6.5 also shows the effects of using all-or-nothing transaction semantics on conflict rate and busy time fraction. This is an expensive option that should be used only when needed by a particular job.

6.4 Review of Replicated State Scheduling

In this chapter we have presented RSS, a natural extension of DPS we explored in collaboration with Google in which the scheduling domains of scheduling agents are copies of a common cluster state synchronized via optimistic concurrency control. We conducted a performance evaluation of the Omega model by extending our existing Monte Carlo simulation



(a) *Median daily conflict fraction.*



(b) *Median daily scheduler busy time fraction.*

Figure 6.5: *Results of simulation of 7 days running Replicated State Scheduler (Omega): effect of all-or-nothing transactions and resource-granularity conflict detection as a function of job time_service.*

framework and synthetic workloads. The evaluation shows that a parallel scheduler model based on optimistic concurrency control over shared state is a viable, attractive approach to cluster scheduling. Although the use of optimistic concurrency control in RSS will do strictly more work than the DPS approach based on a pessimistic scheme because some work may need to be re-done, we found that the amount of additional work is insignificant at reasonable operating points. Furthermore, the resulting benefits of increased parallelism and resource visibility to scheduling agents makes up for this.

Chapter 7

Conclusion and Future Work

We have presented a high level model of cluster scheduling and Monte Carlo simulation framework and used them to compare three cluster scheduling architectures. First, the popular Monolithic State Scheduling (MSS), then two new architectures: Dynamically Partitioned State Scheduling (DPS) and Replicated State Scheduling (RSS).

We also presented the design and implementation of Mesos, a real-world DPS cluster scheduler that allows diverse cluster computing frameworks to efficiently share resources. Mesos implements DPS with a focus on two design elements: a fine-grained sharing model at the level of tasks, and a distributed scheduling mechanism called resource offers that delegates scheduling decisions to the frameworks. Our evaluation showed that these elements let Mesos achieve high utilization, respond quickly to workload changes, and cater to diverse frameworks while remaining scalable and robust. We also showed that existing frameworks can effectively share resources using Mesos, that Mesos enables the development of specialized frameworks providing major performance gains, such as Spark, and that Mesos's simple design allows the system to be fault tolerant and to scale to 50,000 nodes.

Finally, we described and evaluated Replicated State Scheduling, a cluster scheduling architecture being explored by Google in Omega, their next generation cluster management system. We showed that a parallel scheduler model based on optimistic concurrency control over shared state provides some benefits over pessimistic concurrency used in DPS and quantified the costs of the added flexibility in terms of job wait time and scheduling utilization.

In future work, we plan to factor lessons from RSS and Omega back into Mesos. First, we will add RSS-style optimistic concurrency control to Mesos by having the allocation module make resource offers for overlapping sets of resources in parallel. In addition, we will explore ways by which frameworks can give richer hints about which offers they would like to receive.

Bibliography

[1] “Amazon EC2,” http://aws.amazon.com/ec2.

[2] “Apache Hadoop,” http://hadoop.apache.org.

[3] “Apache Hive,” http://hadoop.apache.org/hive.

[4] “Apache ZooKeeper,” hadoop.apache.org/zookeeper.

[5] “HAProxy Homepage,” http://haproxy.1wt.eu/.

[6] “Hive – A Petabyte Scale Data Warehouse using Hadoop,” http://www.facebook.com/note.php?note_id=89508453919.

[7] “Hive performance benchmarks,” http://issues.apache.org/jira/browse/HIVE-396.

[8] “LibProcess Homepage,” http://www.eecs.berkeley.edu/~benh/libprocess.

[9] “Linux 2.6.33 release notes,” http://kernelnewbies.org/Linux_2_6_33.

[10] “Linux containers (LXC) overview document,” http://lxc.sourceforge.net/lxc.html.

[11] “Omega Slide Deck by John Wilkes,” http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/university/relations/facultysummit2011/2011_faculty_summit_omega_wilkes.pdf.

[12] “Personal communication with dhruba borthakur from facebook.”

[13] “Personal communication with owen o’malley and arun c. murphy from the yahoo! hadoop team.”

[14] “Price of Amazon EC2 prices over time.” http://www.cs.washington.edu/homes/billhowe/aws_price_history/allsix.html.

[15] “RightScale blog,” blog.rightscale.com/2010/04/01/benchmarking-load-balancers-in-the-cloud.

[16] “Solaris Resource Management.” http://docs.sun.com/app/docs/doc/817-1592.

[17] “Linux kernel cpusets documentation,” http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt, October 2009.

[18] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, “LAPACK: a portable linear algebra library for high-performance computers,” in *Supercomputing ’90*, 1990.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[20] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[21] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "Mpich-v2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Supercomputing '03*, 2003.

[22] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor," in *ATEC '05*. USENIX Association, 2005, pp. 24–24.

[23] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, "MapReduce online," in *NSDI '10*, May 2010.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[25] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proc. HPDC '10*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *SOSP*, 1995, pp. 251–266.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types," in *NSDI*, 2011.

[28] Google Inc., "Google Cluster Trace," http://code.google.com/p/googleclusterdata/.

[29] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York, NY: Springer Publishing Company, 2009.

[30] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of NSDI 2011*, 2011. [Online]. Available: http://www.usenix.org/events/nsdi11/tech/full_papers/Hindman.pdf

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys 07*, 2007.

[32] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *SOSP*, November 2009.

[33] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "On availability of intermediate data in cloud computations," in *HOTOS*, May 2009.

[34] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *Proc. ACM symposium on Cloud computing*, ser. SoCC '10, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807138

[35] 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 *SIGMOD*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[36] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," in *ExpCS '07*, 2007.

[37] D. Mosberger and T. Jin, "httperf – a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, pp. 31–37, 1998.

[38] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *NSDI*, 2011.

[39] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *CCA '08*, 2008.

[40] R. Raman, M. Livny, and M. Solomon, "Matchmaking: An extensible framework for distributed resource management," *Cluster Computing*, vol. 2, pp. 129–138, April 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=592887.592921

[41] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Science and Technology Center for Cloud Computing, Tech. Rep. ISTC-CC-TR-12-101, April 2012.

[42] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Flexible, scalable schedulers for large compute clusters," Tech. Rep., Under submission.

[43] G. Staples, "TORQUE resource manager," in *Proc. Supercomputing '06*, 2006.

[44] I. Stoica, H. Zhang, and T. S. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *SIGCOMM '97*, 1997, pp. 249–262.

[45] J. Stone, "Tachyon ray tracing system," http://jedi.ks.uiuc.edu/~johns/raytracer.

[46] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: flexible proportional-share resource management," in *OSDI*, 1994.

[47] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *SOSP '09*, 2009, pp. 247–260.

[48] 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 *EuroSys 10*, 2010.

[49] M. Zaharia, M. Chowdhury, T. Das, D. Ankur, 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 *NSDI*, 2012.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *USENIX HotCloud*, 2010.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. OSDI '08*, 2008.

[52] S. Zhou, "LSF: Load sharing in large-scale heterogeneous distributed systems," in *Workshop on Cluster Computing*, 1992.

[53] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," in *AAIM*. Springer-Verlag, 2008, pp. 337–348.