Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

Benjamin Hindman, Andy Konwinski, Matei Zaharia,
Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To support the sophisticated schedulers of today's frameworks, Mesos introduces a distributed two-level scheduling mechanism called resource offers. Mesos decides *how many* resources to offer each framework, while frameworks decide *which* resources to accept and which computations to run on them. Our results show that Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks, can scale to 50,000 (emulated) nodes, and is resilient to failures.

1 Introduction

Clusters of commodity servers have become a major computing platform, powering both large Internet services and a growing number of data-intensive scientific applications. Driven by these applications, researchers and practitioners have been developing a diverse array of cluster computing frameworks to simplify programming the cluster. Prominent examples include MapReduce [18], Dryad [24], MapReduce Online [17] (which supports streaming jobs), Pregel [28] (a specialized framework for graph computations), and others [27, 19, 30].

It seems clear that new cluster computing frameworks[1] will continue to emerge, and that no framework will be optimal for all applications. Therefore, organizations will want to run *multiple frameworks in the same cluster*, picking the best one for each application. 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.

[1] By "framework," we mean a software system that manages and executes one or more jobs on a cluster.

Two common solutions for sharing a cluster today are either to statically partition the cluster and run one framework per partition, or to allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-grained resource sharing model, where nodes are subdivided into "slots" and jobs are composed of short *tasks* that are matched to slots [25, 38]. The short duration of tasks and the ability to run multiple tasks per node allow jobs to achieve high data locality, as each job will quickly get a chance to run on nodes storing its input data. Short tasks also allow frameworks to achieve high utilization, as jobs can rapidly scale when new nodes become available. Unfortunately, because these frameworks are developed independently, there is no way to perform fine-grained sharing *across* frameworks, making it difficult to share clusters and data efficiently between them.

In this paper, we propose Mesos, a thin resource sharing layer that enables fine-grained sharing across diverse cluster computing frameworks, by giving frameworks a common interface for accessing cluster resources.

The main design question for Mesos is how to build a scalable and efficient system that supports a wide array of both current and future frameworks. This 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 hundreds of jobs with millions of tasks. Finally, because all the applications in the cluster depend on Mesos, the system must be fault-tolerant and highly available.

One approach would be for Mesos to implement a centralized scheduler that takes as input 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 to solve an online optimization problem for millions of tasks. Even if such a scheduler were feasible, this complexity would have a negative impact on its scalability and resilience. Second, as new frameworks and new scheduling policies for current frameworks are constantly being developed [37, 38, 40, 26], 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 [25, 38], and moving this functionality to a global scheduler would require expensive refactoring.

Instead, Mesos takes a different approach: delegating control over scheduling to the frameworks. This is accomplished through a new abstraction, called a *resource offer*, which encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks. Mesos decides *how many* resources to offer each framework, based on an organizational policy such as fair sharing, while frameworks decide *which* resources to accept and which tasks to run on them. While this decentralized scheduling model may not always lead to globally optimal scheduling, we have found that it performs surprisingly well in practice, allowing frameworks to meet goals such as data locality nearly perfectly. In addition, resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

Mesos also provides other benefits to practitioners. First, even organizations that only use one framework can use Mesos to run multiple instances of that framework in the same cluster, or multiple versions of the framework. Our contacts at Yahoo! and Facebook indicate that this would be a compelling way to isolate production and experimental Hadoop workloads and to roll out new versions of Hadoop [11, 10]. 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.

We have implemented Mesos in 10,000 lines of C++. The system scales to 50,000 (emulated) nodes and uses ZooKeeper [4] for fault tolerance. To evaluate Mesos, we have ported three cluster computing systems to run over it: Hadoop, MPI, and the Torque batch scheduler. To validate our hypothesis that specialized frameworks provide value over general ones, we have also built a new framework on top of Mesos called Spark, optimized for iterative jobs where a dataset is reused in many parallel operations, and shown that Spark can outperform Hadoop by 10x in iterative machine learning workloads.

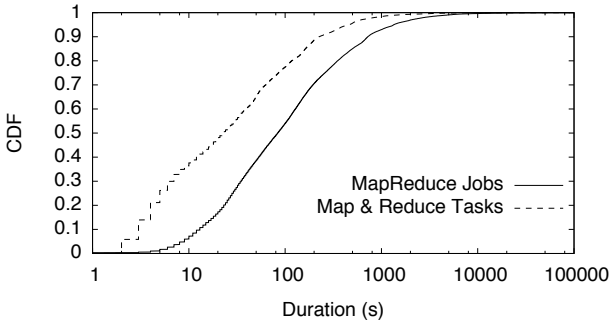


Figure 1: CDF of job and task durations in Facebook's Hadoop data warehouse (data from [38]).

This paper is organized as follows. Section 2 details the data center environment that Mesos is designed for. Section 3 presents the architecture of Mesos. Section 4 analyzes our distributed scheduling model (resource offers) and characterizes the environments that it works well in. We present our implementation of Mesos in Section 5 and evaluate it in Section 6. We survey related work in Section 7. Finally, we conclude in Section 8.

2 Target Environment

As an example of a workload we aim to support, consider the Hadoop data warehouse at Facebook [5]. Facebook loads logs from its web services into a 2000-node Hadoop cluster, where they are used for applications such as business intelligence, spam detection, and ad optimization. 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 an SQL interface called Hive [3]. Most jobs are short (the median job being 84s long), and the jobs are composed of fine-grained map and reduce tasks (the median task being 23s), as shown in Figure 1.

To meet the performance requirements of these 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 [38]. Unfortunately, this means that the cluster can only run Hadoop jobs. If a user wishes to write an ad targeting algorithm in 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) [11, 10]. Mesos aims to provide fine-grained sharing between *multiple* cluster computing frameworks to enable these usage scenarios.

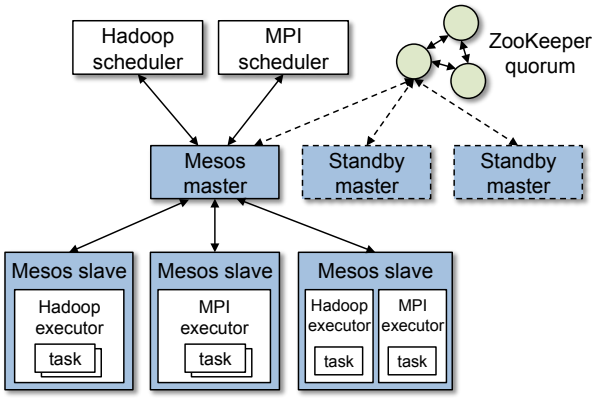


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

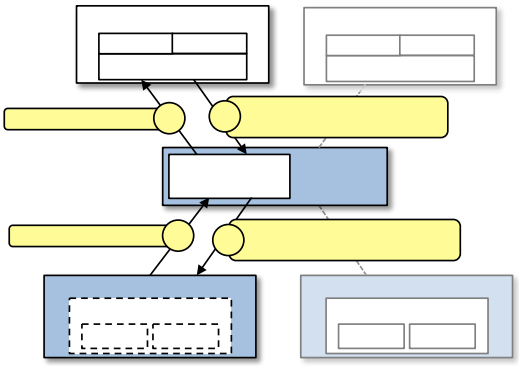


Figure 3: Resource offer example.

3 Architecture

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

3.1 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 problems in the cluster (e.g., achieving data locality, dealing with faults), and 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.

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 [20]. Putting this functionality in libraries rather than in Mesos allows Mesos to remain small and flexible, and lets the libraries evolve independently.

3.2 Overview

Figure 2 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 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 3 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), the framework's scheduler replies to the master with information about two tasks to run on the slave, using $\langle 2$ CPUs, 1 GB RAM$\rangle$ for the first task, and $\langle 1$ CPUs, 2 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 thin 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 in order 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 rejec-

tion 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 whitelist of nodes it can 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 (*e.g.,* in MapReduce and Dryad workloads), 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 [38], 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 (§3.3) and resource isolation (§3.4). We then describe filters and several other mechanisms that make resource offers scalable and robust (§3.5). Finally, we discuss fault tolerance in Mesos (§3.6) and summarize the Mesos API (§3.7).

3.3 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 [21] and one that implements strict priorities. Similar policies are used in Hadoop and Dryad [25, 38].

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. However, if a cluster becomes filled by long tasks, e.g., due to a buggy job or a greedy framework, the allocation module can also *revoke* (kill) tasks. Before killing a task, Mesos gives its framework a grace period to clean it up.

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 (*e.g.,* MapReduce), it is harmful for frameworks with interdependent tasks (*e.g.,* 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.

3.4 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 [9] and Solaris Projects [13]. 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.

3.5 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.

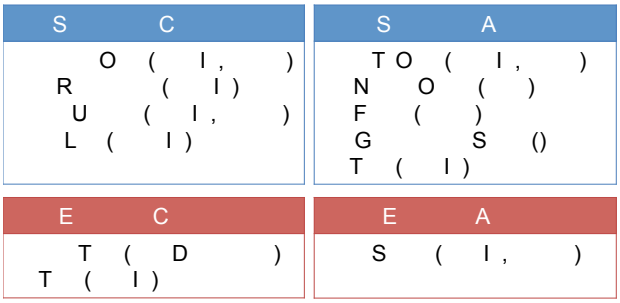


Table 1: Mesos API functions for schedulers and executors.

3.6 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.

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.

3.7 API Summary

Table 1 summarizes the Mesos API. The "callback" columns list functions that frameworks must implement, while "actions" are operations that they can invoke.

4 Mesos Behavior

In this section, we study Mesos's behavior for different workloads. Our goal is not to develop an exact model of the system, but to provide a coarse understanding of its behavior, in order to characterize the environments that Mesos's distributed scheduling model works well in.

In short, we find that Mesos performs very well when frameworks can scale up and down elastically, tasks durations are homogeneous, and frameworks prefer all nodes equally (§4.2). When different frameworks prefer different nodes, we show that Mesos can emulate a centralized scheduler that performs fair sharing across frameworks (§4.3). In addition, we show that Mesos can handle heterogeneous task durations without impacting the performance of frameworks with short tasks (§4.4). We also discuss how frameworks are incentivized to improve their performance under Mesos, and argue that these incentives also improve overall cluster utilization (§4.5). We conclude this section with some limitations of Mesos's distributed scheduling model (§4.6).

4.1 Definitions, Metrics and Assumptions

In our 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 using nodes as soon as it acquires them and release them as soon its task finish. 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 in order 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 mandatory resources to become free.[2] 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.

4.2 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 com-

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

	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 2: 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 under the scheduling policy, and βT represents the time it takes a job to complete assuming the framework gets all k slots at once.

putation time. That is, when the framework has k slots, it takes its job βT time to finish.

Table 2 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. Due to lack of space, we present only the results here and include derivations in [23].

Framework ramp-up time: If task durations are constant, it will take framework f at most T time to acquire k slots. This is simply because during a T interval, every slot will become available, which will enable Mesos to offer the framework all k of its preferred slots. If the duration distribution is exponential, the expected ramp-up time can be as high as $T \ln k$ [23].

Job completion time: The expected completion time[3] of an elastic job is at most $(1+\beta)T$, which is within T (*i.e.,* the mean task duration) of the completion time of the job when it gets all its slots instantaneously. Rigid jobs achieve similar completion times for constant task durations, but exhibit much higher completion times for exponential job durations, *i.e.,* $(\ln k+\beta)T$. This is simply because it takes a framework $T \ln k$ time on average to acquire all its slots and be able to start its job.

System utilization: Elastic jobs fully utilize their allocated slots, because they can use every slot as soon as they get it. As a result, assuming infinite demand, a system running only elastic jobs is fully utilized. Rigid frameworks achieve slightly worse utilizations, as their jobs cannot start before they get their full allocations, and thus they waste the resources held while ramping up.

4.3 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 central scheduler that 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 [36], 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

best,-330(ta86)ion dh8-244(ta86)

[3]When computing job completion time we assume that the last tasks of the job running on the framework's k slots finish at the same time.

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 many 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.

4.5 Framework Incentives

Mesos implements a decentralized scheduling model, where each framework decides which offers to accept. As with any decentralized 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.

4.6 Limitations of Distributed Scheduling

Although we have shown that distributed scheduling works well in a range of workloads relevant to current cluster environments, like any decentralized approach, it can perform worse than a centralized scheduler. We have identified three limitations of the distributed model:

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 frame-

works cannot predict task times and must be able to handle failures and stragglers [18, 40, 38]. These policies are easy to implement over resource offers.

5 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` [7] 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 [9] or Solaris projects [13] 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 [8] 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 [33], and the MPICH2 implementation of MPI [16]. None of these ports required changing these frameworks' APIs, so all of them can run unmodified user programs. In addition, we built a specialized framework for iterative jobs called Spark, which we discuss in Section 5.3.

5.1 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 [38] 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.

5.2 Torque and MPI Ports

We have ported the Torque cluster resource manager 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 3 lines of Torque source code to allow it to elastically scale up and down on Mesos depending on the jobs in its queue.

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.

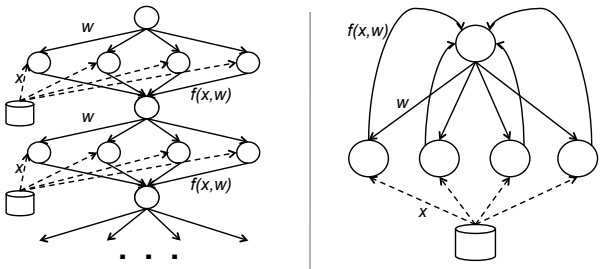
Because jobs that run on Torque (e.g. MPI) may not be fault tolerant, Torque avoids having its tasks revoked by not accepting resources beyond its guaranteed allocation.

In addition to the Torque framework, we also created a Mesos MPI "wrapper" framework, written in 200 lines of Python code, for running MPI jobs directly on Mesos.

5.3 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. We built a specialized framework called Spark [39] optimized for these workloads.

One example of an iterative algorithm used in machine learning is logistic regression [22]. This algorithm seeks to find a line that separates two sets of labeled data points. The algorithm starts with a random line w. Then, on each iteration, it computes the gradient of an objective



a) Dryad b) Spark

Figure 4: 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.

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 4a, 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 4b. 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, we were able to keep its implementation small (about 1300 lines of code), yet still capable of outperforming Hadoop by $10\times$ for iterative jobs. In particular, using Mesos's API saved us the time to write a master daemon, slave daemon, and communication protocols between them for Spark. The main pieces we had to write were a framework scheduler (which uses delay scheduling for locality) and user APIs.

6 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.

6.1 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:

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 3: Job types for each bin in our Facebook Hadoop mix.

- 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 4 CPU cores and 15 GB of RAM.

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

6.1.1 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 [38]. The workload consists of 100 jobs 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.[4] The jobs themselves were from the Hive benchmark [6], 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 3) 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 2 minutes after each job ended to submit the next. The job we used was alternating least squares

[4] We scaled down the largest jobs in [38] to have the workload fit a quarter of our cluster size.

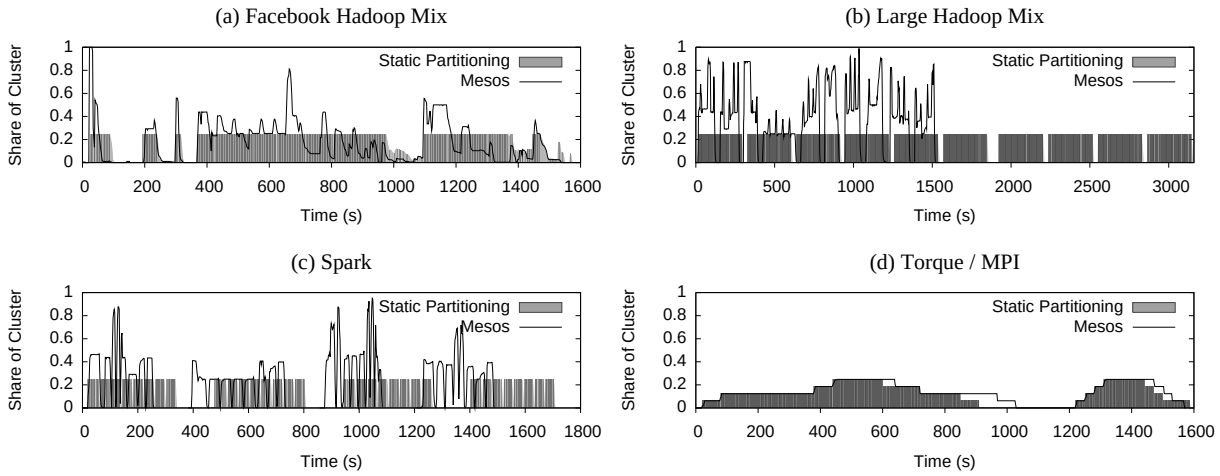


Figure 5: Comparison of cluster shares (fraction of CPUs) over time for each of the frameworks in the Mesos and static partitioning macrobenchmark scenarios. On Mesos, frameworks can scale up when their demand is high and that of other frameworks is low, and thus finish jobs faster. Note that the plots' time axes are different (*e.g.,* the large Hadoop mix takes 3200s with static partitioning).

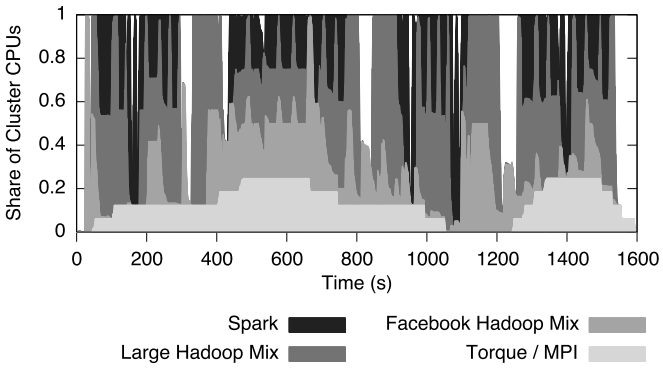


Figure 6: Framework shares on Mesos during the macrobenchmark. By pooling resources, Mesos lets each workload scale up to fill gaps in the demand of others. In addition, fine-grained sharing allows resources to be reallocated in tens of seconds.

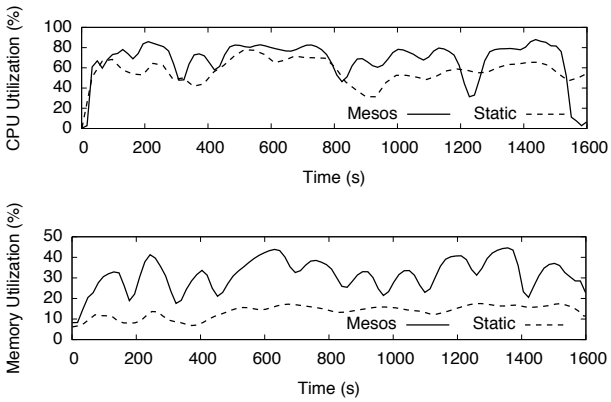


Figure 7: Average CPU and memory utilization over time across all nodes in the Mesos cluster vs. static partitioning.

(ALS), a collaborative filtering algorithm [42]. 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 [35] 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.

6.1.2 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 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 6. 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 7.

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 compares the resource allocation over time of each framework in the shared and statically partitioned clusters. Shaded areas show the allocation in the stat-

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 4: 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.

Framework	Job Type	Exec Time w/ Static Partitioning (s)	Avg. Speedup on Mesos
Faceb k Had Mi	e ec i (1)	24	**0.84**
	e ea ch (2)	31	**0.90**
	agg ega i (3)	82	**0.94**
	e ec i (4)	65	**1.40**
	agg ega i (5)	192	**1.26**
	e ec i (6)	136	**1.71**
	e ea ch (7)	137	**2.14**
	j i (8)	662	**1.35**
La ge Had Mi	e ea ch	314	**2.21**
S a k	ALS	337	**1.36**
T e / MPI	a ach	261	**0.91**
	a ge ach	822	**0.88**

Table 5: Performance of each job type in the macrobenchmark. Bins for the Facebook Hadoop mix are in parentheses.

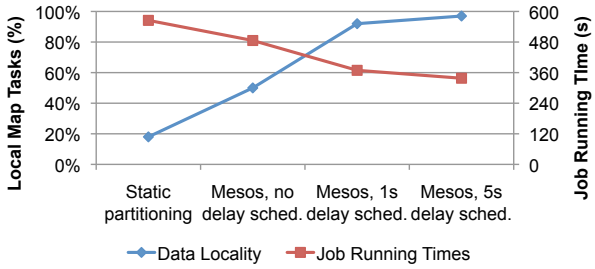


Figure 8: Data locality and average job durations for 16 Hadoop instances running on a 93-node cluster using static partitioning, Mesos, or Mesos with delay scheduling.

ically 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 4 and 5 show a breakdown of job performance for each framework. In Table 4, we compare the aggregate performance of each framework, defined as the sum 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 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 hierarchical fair sharing is also seen in networks [34], 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 2 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. 5d). 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.

6.2 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 [15] 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%.

6.3 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 4 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 8 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 are 4 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.

6.4 Spark Framework

We evaluated the benefit of running iterative jobs using the specialized Spark framework we developed on top of Mesos (Section 5.3) 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 4 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 9).

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.

6.5 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 8 CPU cores and 6 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

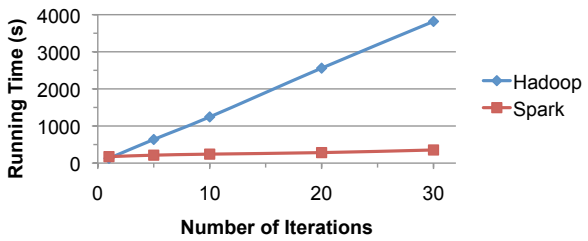


Figure 9: Hadoop and Spark logistic regression running times.

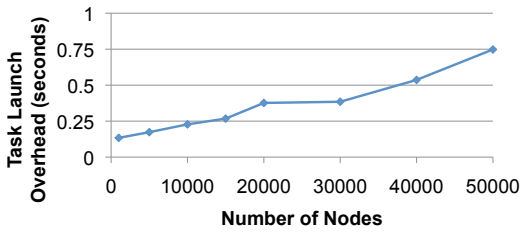


Figure 10: Mesos master's scalability versus number of slaves.

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 10 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 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 10, showing averages of 5 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). 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 [12].

6.6 Failure Recovery

To evaluate recovery from master failures, we conducted an experiment with 200 to 4000 slave daemons on 62 EC2 nodes with 4 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 4 and 8 seconds, with 95% confidence intervals of up to 3s on either side.

6.7 Performance Isolation

As discussed in Section 3.4, 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 [9] 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 [29] for a fuller evaluation of OS isolation mechanisms.

7 Related Work

HPC and Grid Schedulers. The high performance computing (HPC) community has long been managing clusters [33, 41]. However, their target environment typically consists of specialized hardware, such as Infiniband and SANs, where jobs do not need to be scheduled local to their data. Furthermore, each job is tightly coupled, often using barriers or message passing. Thus, each job is monolithic, rather than composed of fine-grained tasks, and does not change its resource demands during its lifetime. For these reasons, HPC schedulers use centralized scheduling, and require users to declare the required resources at job submission time. Jobs are then given coarse-grained allocations of the cluster. Unlike the Mesos approach, this does not allow jobs to locally access data distributed across the cluster. Furthermore, jobs cannot grow and shrink dynamically. In contrast, Mesos supports fine-grained sharing at the level of tasks and allows frameworks to control their placement.

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. Mesos could well be used within a virtual organization inside a larger grid.

Public and Private Clouds. Virtual machine clouds such as Amazon EC2 [1] and Eucalyptus [31] share common goals with Mesos, such as isolating applications while providing a low-level abstraction (VMs). However, they differ from Mesos in several important ways. First, their relatively coarse grained VM allocation model leads to less efficient resource utilization and data sharing than in Mesos. Second, these systems generally do not let applications specify placement needs beyond the size of VM they require. In contrast, Mesos allows frameworks to be highly selective about task placement.

Quincy. Quincy [25] is a fair scheduler for Dryad that uses a centralized scheduling algorithm for Dryad's DAG-based programming model. In contrast, Mesos provides the lower-level abstraction of resource offers to support *multiple* cluster computing frameworks.

Condor. The Condor cluster manager uses the ClassAds language [32] to match nodes to jobs. Using a resource specification language is not as flexible for frameworks as resource offers, since not all requirements may be expressible. Also, porting existing frameworks, which have their own schedulers, to Condor would be more difficult than porting them to Mesos, where existing schedulers fit naturally into the two-level scheduling model.

Next-Generation Hadoop. Recently, Yahoo! announced a redesign for Hadoop that uses a two-level scheduling model, where per-application masters request resources from a central manager [14]. The design aims to support non-MapReduce applications as well. While details about the scheduling model in this system are currently unavailable, we believe that the new application masters could naturally run as Mesos frameworks.

8 Conclusion and Future Work

We have presented Mesos, a thin management layer that allows diverse cluster computing frameworks to efficiently share resources. Mesos is built around 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. Together, these elements let Mesos achieve high utilization, respond quickly to workload changes, and cater to diverse frameworks while remaining scalable and robust. We have shown 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.

In future work, we plan to further analyze the resource offer model and determine whether any extensions can improve its efficiency while retaining its flexibility. In particular, it may be possible to have frameworks give richer hints about offers they would like to receive. Nonetheless, we believe that below any hint system, frameworks should still have the ability to reject offers and to choose which tasks to launch on each resource, so that their evolution is not constrained by the hint language provided by the system.

We are also currently using Mesos to manage resources on a 40-node cluster in our lab and in a test deployment at Twitter, and plan to report on lessons from

these deployments in future work.

9 Acknowledgements

We thank our industry colleagues at Google, Twitter, Facebook, Yahoo! and Cloudera for their valuable feedback on Mesos. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, a National Science Foundation Graduate Research Fellowship,[5] the Swedish Research Council, and the following Berkeley RAD Lab sponsors: Google, Microsoft, Oracle, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

References

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

[6] Hive performance benchmarks. http://issues.apache.org/jira/browse/HIVE-396.

[7] LibProcess Homepage. http://www.eecs.berkeley.edu/~benh/libprocess.

[8] Linux 2.6.33 release notes. http://kernelnewbies.org/Linux_2_6_33.

[9] Linux containers (LXC) overview document. http://lxc.sourceforge.net/lxc.html.

[10] Personal communication with Dhruba Borthakur from Facebook.

[11] Personal communication with Owen O'Malley and Arun C. Murthy from the Yahoo! Hadoop team.

[12] RightScale blog. blog.rightscale.com/2010/04/01/benchmarking-load-balancers-in-the-cloud.

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

[14] The Next Generation of Apache Hadoop MapReduce. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen.

[15] 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.

[16] 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.

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

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[19] 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.

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

[21] 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.

[5] Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

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

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, UC Berkeley, May 2010.

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

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

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

[27] 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*, SoCC '10, 2010.

[28] 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*, pages 135–146, 2010.

[29] 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.

[30] 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.

[31] 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.

[32] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2:129–138, April 1999.

[33] G. Staples. TORQUE resource manager. In *Proc. Supercomputing '06*, 2006.

[34] 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*, pages 249–262, 1997.

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

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

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

[38] 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.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. HotCloud '10*, 2010.

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

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

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