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

Benjamin Hindman
Andrew Konwinski
Matei Zaharia
Ali Ghodsi
Anthony D. Joseph
Randy H. Katz
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-87
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-87.html

May 26, 2010

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 [1]. 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 experimental results show that Mesos can achieve near-optimal locality when sharing the cluster among diverse frameworks, can scale up to 50,000 nodes, and is resilient to node 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 [25], Dryad [33], MapReduce Online [24] (which supports streaming jobs), Pregel [37] (a specialized framework for graph computations), and others [36, 17, 31].

It seems clear that new cluster computing frameworks 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. Sharing a cluster between frameworks is desirable for two reasons: it improves utilization through statistical multiplexing, and it lets applications share datasets that may be too expensive to replicate. In this paper, we explore the problem of efficiently sharing commodity clusters among diverse cluster computing frameworks.

An important feature of commodity clusters is that data is distributed throughout the cluster and stored on the same nodes that run computations. In these environments, reading data remotely is expensive, so it is important to schedule computations near their data. Consequently, sharing the cluster requires a *fine-grained* scheduling model, where applications take turns running computations on each node. Existing cluster computing frameworks, such as Hadoop and Dryad, already implement fine-grained sharing across their jobs by multiplexing at the level of tasks within a job [34, 48]. However, because these frameworks are developed independently, there is no way to perform fine-grained sharing across *different* frameworks, making it difficult to share clusters and data efficiently between frameworks.

In this paper, we propose Mesos, a thin resource management substrate that enables efficient fine-grained sharing across diverse cluster computing frameworks. Mesos gives diverse frameworks a common interface for running fine-grained tasks in a cluster.

The main challenge Mesos must address is that different frameworks have different scheduling needs. Each framework has scheduling preferences specific to its programming model, based on the dependencies between its tasks, the location of its data, its communication pattern, and domain-specific optimizations. Furthermore, frameworks' scheduling policies are rapidly evolving [46, 48, 50, 35]. Therefore, designing a generic scheduler to support all current and future frameworks is hard, if not infeasible. Instead, Mesos takes a different approach: giving frameworks control over their scheduling.

Mesos lets frameworks choose which resources to use through a distributed scheduling mechanism called *resource offers*. Mesos decides *how many* resources to offer each framework, based on an organizational pol-

[1] We will continue to improve this Technical Report. The latest version is available at `http://mesos.berkeley.edu/tr-mesos.pdf`.

icy such as fair sharing, but frameworks decide *which* resources to accept and which computations to run on them. We have found that resource offers are flexible enough to let frameworks achieve a variety of placement goals, including data locality. In addition, resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

Mesos's flexible fine-grained sharing model also has other advantages. First, Mesos can support a wide variety of applications beyond data-intensive computing frameworks, such as client-facing services (e.g., web servers) and compute-intensive parallel applications (e.g., MPI), allowing organizations to consolidate workloads and increase utilization. Although not all of these applications run tasks that are fine-grained in time, Mesos can share nodes in space between short and long tasks.

Second, even organizations that only use one framework can leverage Mesos to run multiple isolated instances of that framework in the same cluster, or multiple versions of the framework. For example, an organization using Hadoop might want to run separate instances of Hadoop for production and experimental jobs, or to test a new Hadoop version alongside its existing one.

Third, by providing a platform for resource sharing across frameworks, Mesos allows framework developers to build *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 nodes and uses Apache 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. Spark can outperform Hadoop by 10x in iterative machine learning workloads. Finally, to evaluate the applicability of Mesos to client-facing workloads, we have built an elastic Apache web server farm framework.

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 and characterizes the environments it works well in. We present our implementation of Mesos in Section 5 and evaluate it in Section 6. Section 7 surveys related work. We conclude with a discussion in Section 8.

2 Target Environment

As an example of a workload we aim to support, consider the Hadoop data warehouse at Facebook [5, 7].

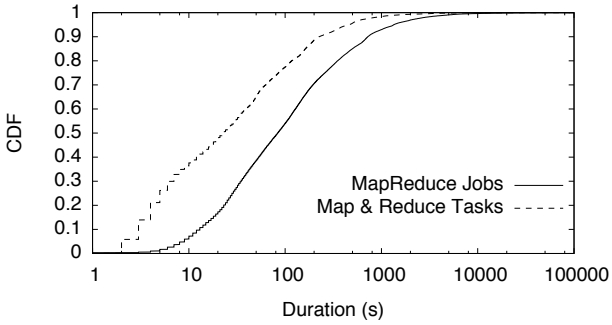


Figure 1: CDF of job and task durations in Facebook's Hadoop data warehouse (data from [48]).

Facebook loads logs from its web services into a 1200-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 to Hadoop called Hive [3]. Most jobs are short (the median 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 make scheduling decisions at the level of map and reduce tasks and to optimize data locality [48]. Unfortunately, this means that the cluster can only run Hadoop jobs. If a user wishes to write a new 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.[2] Mesos aims to enable fine-grained sharing between *multiple* cluster computing frameworks, while giving these frameworks enough control to achieve placement goals such as data locality.

In addition to sharing clusters between "back-end" applications such as Hadoop and MPI, Mesos also enables an organizations to share resources between "front-end" workloads, such as web servers, and back-end workloads. This is attractive because front-end applications have variable load patterns (e.g. diurnal cycles and spikes), so there is an opportunity to scale them down at during periods of low load and use free resources to speed up back-end workloads. We recognize that there are obstacles beyond node scheduling to collocating front-end and back-end workloads at very large scales, such as the difficulty of isolating network traffic [30]. In Mesos, we focus on defining a resource access

[2]This problem is not hypothetical; our contacts at Yahoo! and Facebook report that users want to run MPI and MapReduce Online [13, 12].

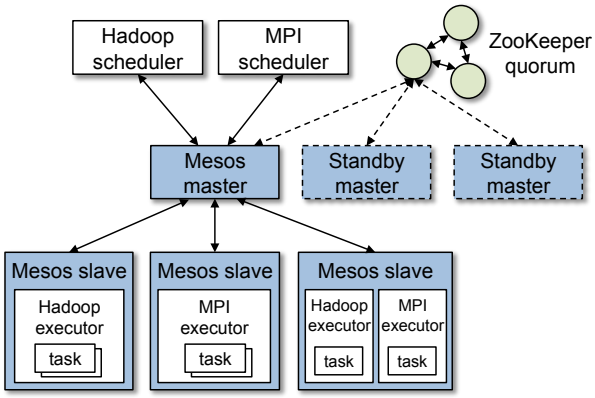


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

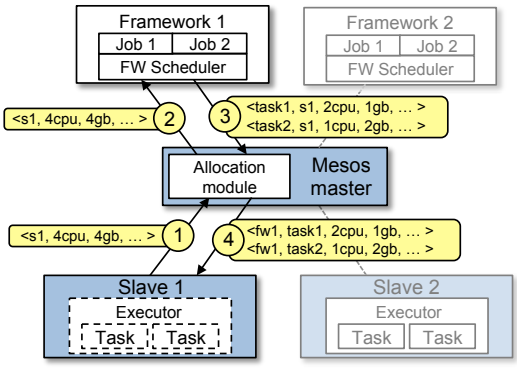


Figure 3: Resource offer example.

interface that allows these applications to coexist, and we hope to leverage isolation solutions developed by others.

3 Architecture

We begin our description of Mesos by presenting 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

Recall that Mesos aims to provide a stable and scalable core that diverse frameworks can run over to share cluster resources. 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, and otherwise push control 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., 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 make 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 [27]. 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 components of Mesos. The system consists a *master* process that manages *slave* daemons running on each cluster node. We use ZooKeeper [4] to make the master fault tolerant, as we shall describe in Section 3.6. Frameworks running on Mesos consist 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. The main role of the Mesos master is to offer available resources on slaves to framework schedulers through *resource offers*. Each resource offer contains a list of free resources on multiple slaves. Multiple offers describing disjoint resource sets can be outstanding at each time. A pluggable allocation module in the master determines *how many* resources to offer to each framework. Frameworks' schedulers select *which* of the offered resources to use, and describe *tasks* to launch on those resources.

Figure 3 shows an example of how a framework gets scheduled to run a task. 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 policy module, which tells it that framework 1 should be offered all available resources. In step (2) the master sends a resource offer describing what is available on slave 1 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 $\langle 2 \text{ CPUs}, 1 \text{ GB RAM} \rangle$ 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-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, 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.

The key aspect of Mesos that lets frameworks achieve placement goals is the fact that they can reject resources. In particular, we have found that a simple policy called delay scheduling [48], in which frameworks wait for a limited time to acquire preferred nodes, yields nearly optimal data locality. We report these results in Section 6.3.

In this section, we describe how Mesos performs two major roles: allocation (performed by allocation modules in the master) and isolation (performed by the slaves). We include a discussion of one allocation policy we have developed for fairly sharing multiple resources. We then

describe elements of our architecture that let resource offers work robustly and efficiently in a distributed system.

3.3 Resource Allocation

Mesos delegates allocation decisions to a pluggable *allocation module*, so that organizations can tailor allocation to their needs. In normal operation, Mesos takes advantage of the fine-grained nature of tasks to only reallocate resources when tasks finish. This usually happens frequently enough to let new frameworks start within a fraction of the average task length: for example, if a framework's share is 10% of the cluster, it needs to wait on average 10% of the mean task length to receive its share. Therefore, the allocation module only needs to decide which frameworks to offer free resources to, and how many resources to offer to them. However, our architecture also supports long tasks by allowing allocation modules to specifically designate a set of resources on each node for use by long tasks. Finally, we give allocation modules the power to *revoke* (kill) tasks if resources are not becoming free quickly enough.

In this section, we start by describing one allocation policy that we have developed, which performs fair sharing between frameworks using a definition of fairness for multiple resources called Dominant Resource Fairness. We then explain Mesos's mechanisms for supporting long tasks and revocation.

3.3.1 Dominant Resource Fairness (DRF)

Because Mesos manages *multiple* resources (CPU, memory, network bandwidth, etc.) on each slave, a natural question is what constitutes a fair allocation when different frameworks prioritize resources differently. We carried out a thorough study of desirable fairness properties and possible definitions of fairness, which is detailed in [29]. We concluded that most other definitions of fairness have undesirable properties. For example, Competitive Equilibrium from Equal Incomes (CEEI) [44], the preferred fairness metric in micro-economics [39], has the disadvantage that the freeing up of resources might punish an existing user's allocation. Similarly, other notions of fairness violated other desirable properties, such as *envy-freedom*, possibly leading to users gaming the system by hoarding resources that they do not need.

To this end, we designed a fairness policy called *dominant resource fairness* (DRF), which attempts to equalize each framework's fractional share of its *dominant resource*, which is the resource that it has the largest fractional share of. For example, if a cluster contains 100 CPUs and 100 GB of RAM, and framework F_1 needs 4 CPUs and 1 GB RAM per task while F_2 needs 1 CPU and 8 GB RAM per task, then DRF gives F_1 20 tasks (80 CPUs and 20 GB) and gives F_2 10 tasks (10 CPUs and 80 GB). This makes F_1's share of CPU equal to F_2's share of RAM, while fully utilizing one resource (RAM).

DRF is a natural generalization of max/min fairness [21]. DRF satisfies the above mentioned properties and performs scheduling in $O(\log n)$ time for n frameworks. We believe that DRF is novel, but due to space constraints, we refer the reader to [29] for details.

3.3.2 Supporting Long Tasks

Apart from fine-grained workloads consisting of short tasks, Mesos also aims to support frameworks with longer tasks, such as web services and MPI programs. This is accomplished by sharing nodes in space between long and short tasks: for example, a 4-core node could be running MPI on two cores, while also running MapReduce tasks that access local data. If long tasks are placed arbitrarily throughout the cluster, however, some nodes may become filled with them, preventing other frameworks from accessing local data. To address this problem, Mesos allows allocation modules to bound the total resources on each node that can run long tasks. The amount of long task resources still available on the node is reported to frameworks in resource offers. When a framework launches a task, it marks it as either long or short. Short tasks can use any resources, but long tasks can only use up to the amount specified in the offer.

Of course, a framework may launch a long task without marking it as such. In this case, Mesos will eventually revoke it, as we discuss next.

3.3.3 Revocation

As described earlier, in an environment with fine-grained tasks, Mesos can reallocate resources quickly by simply waiting for tasks to finish. However, if a cluster becomes filled by long tasks, e.g., due to a buggy job or a greedy framework, Mesos can also revoke (kill) tasks. Before killing a task, Mesos gives its framework a *grace period* to clean it up. Mesos asks the respective executor to kill the task, but kills the entire executor and all its tasks if it does not respond to the request. We leave it up to the allocation module to implement 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 or stateless web servers), 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. However, if this model is found to be too simple, it is also possible to let frameworks specify priorities for their tasks, so that the allocation module can try to kill only low-priority tasks.

Second, to decide when to trigger revocation, allocation modules must know which 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*.

In our current implementation, we use operating system container technologies, specifically Linux containers [10] and Solaris projects [16], to achieve isolation. These technologies can limit the CPU, physical memory, virtual memory, network bandwidth, and (in new Linux kernels) IO bandwidth usage of a process tree. In addition, they support dynamic reconfiguration of a container's resource limits, which is necessary for Mesos to be able to add and remove resources from an executor as it starts and finishes tasks. In the future, it would also be attractive to use virtual machines as containers. However, we have not yet done this because current VM technologies add non-negligible overhead in data-intensive workloads and have limited support for dynamic reconfiguration.

3.5 Making Resource Offers Scalable and Robust

Because task scheduling in Mesos is a distributed process in which the master and framework schedulers communicate, 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 support two types of filters: "only offer nodes from list L" and "only offer nodes with at least R resources free". A resource that fails a filter is treated exactly like a rejected resource. By default, any resources rejected during an offer have a temporary 5-second filter placed on them, to minimize the programming burden on developers who do not wish to manually set filters.

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

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.

Scheduler Callbacks	Scheduler Actions
resource_offer(offer_id, offers) offer_rescinded(offer_id) status_update(task_id, status) slave_lost(slave_id)	reply_to_offer(offer_id, tasks, needs_more_offers) request_offers() set_filters(filters) get_safe_share() kill_task(task_id)
Executor Callbacks	**Executor Actions**
launch_task(task_descriptor) kill_task(task_id)	send_status(task_id, status)

Table 1: Mesos API functions for schedulers and executors. The "callback" columns list functions that frameworks must implement, while "actions" are operations that they can invoke.

We also note that even without the use of filters, Mesos can make tens of thousands of resource offers per second, because the scheduling algorithm it must perform (fair sharing) is highly efficient.

3.5.1 API Summary

Table 1 summarizes the Mesos API. The only function that we have not yet explained is the `kill_task` function that a scheduler may call to kill one of its tasks. This is useful for frameworks that implement backup tasks [25].

3.6 Fault Tolerance

Since the master is a centerpiece of our architecture, we have made it fault-tolerant by pushing state to slaves and schedulers, making the master's state *soft state*. At runtime, multiple Mesos masters run simultaneously, but only one master is the leader. The others masters act as hot standbys ready to take over if the current leader fails. ZooKeeper [4] is used to implement leader election among masters. Schedulers and slaves also find out about the current leader through ZooKeeper. Upon the failure of the master, the slaves and schedulers connect to the newly elected master and help restore its state. We also ensure that messages sent to a failed master are resent to the new one through a thin communication layer that uses sequence numbers and retransmissions.

Aside from handling master failures, Mesos reports task, slave and executor failures to frameworks' schedulers. Frameworks can then react to failures using policies of their choice.

Finally, to deal with scheduler failures, Mesos can be extended to allow a framework to register multiple schedulers such that if one fails, another is notified by the Mesos master and takes over. Frameworks must use their own mechanisms to share state between their schedulers.

4 Mesos Behavior

In this section, we study Mesos's behavior for different workloads. 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. When different frameworks prefer different nodes, we show that Mesos can emulate a centralized scheduler that uses fair sharing across frameworks. In addition, we show that Mesos can handle heterogeneous task durations without impacting the performance of frameworks with short tasks. We also discuss how frameworks are incentivized to improve their performance under Mesos, and argue that these incentives also improve overall cluster utilization. Finally, we conclude this section with some limitations of Mesos's distributed scheduling model.

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 four attributes:

- *Scale up:* Frameworks can elastically increase their allocation to take advantage of free resources.
- *Scale down:* Frameworks can relinquish resources without significantly impacting their performance.
- *Minimum allocation:* Frameworks require a certain minimum number of slots before they can start using their slots.
- *Task distribution:* The distribution of the task durations. We consider both homogeneous and heterogeneous distributions.

We also differentiate between two types of resources: *required* and *preferred*. We say that a resource is *required* if a framework must acquire it in order to run. For example, we consider a graphical processing unit (GPU) a required resource if there are frameworks that must use a GPU to run. Likewise, we consider a machine with a public IP address a required resource if there are frameworks that need to be externally accessible. In contrast, we say that a resource is *preferred* if a framework will perform "better" using it, but can also run using other resources. For example, a framework may prefer using a node that locally stores its data, but it can remotely access the data from other nodes if it must.

We assume that all required resources are explicitly allocated, *i.e.,* every type of required resource is an element in the resource vector in resource offers. Furthermore, provided a framework never uses more than its guaranteed allocation of required resources, explicitly accounting for these resources ensures that frameworks will not get deadlocked waiting for them to become available.

For simplicity, in the reminder of this section we assume that all tasks run on identical slices of machines, which we call *slots*. Also, unless otherwise specified, we assume that each framework runs a single job.

Next, we use this simple model to estimate the framework ramp-up time, the job completion time, and the job resource utilization. We emphasize that our goal here is not to develop a detailed and exact model of the system, but to provide a coarse understanding of the system behavior.

4.2 Homogeneous Tasks

In the rest of this section we consider a task duration distribution with mean T_s. For simplicity, we present results for two distributions: constant task sizes and exponentially distributed task sizes. We consider a cluster with n slots and a framework, f, that is entitled to k slots. We assume the framework runs a job which requires $\beta k T_s$ computation time. Thus, assuming the framework has k slots, it takes the job βT_s time to finish. When computing the completion time of a job we assume that the last tasks of the job running on the framework's k slots finish at the *same* time. Thus, we relax the assumptions about the duration distribution for the tasks of framework f. This relaxation does not impact any of the other metrics, *i.e.,* ramp-up time and utilization.

We consider two types of frameworks: *elastic* and *rigid*. An elastic framework can scale its resources up and down, *i.e.,* it can start using slots as soon as it acquires them, and can release slots as soon its task finish. In contrast, a rigid framework can start running its jobs only after it has allocated all its slots, *i.e.,* a rigid framework is a framework where the minimum allocation is equal to its full allocation.

Table 2 summarizes the job completion times and the utilization for the two types of frameworks and for the two types of task length distributions. We discuss each case next.

4.2.1 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.

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

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

Table 2: The job completion time and utilization for both elastic and rigid frameworks, and for both constant task durations and task durations that follow an exponential distribution. The framework starts with zero slots. k represents the number of slots the framework is entitled under the given allocation, and βT_s represents the time it takes a job to complete assuming the framework gets all k slots at once.

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

$$T_s \times (1 + 1/2.. + 1/k) \simeq T_s \ln k. \quad (1)$$

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 2).

In the case of the exponential distribution, the expected completion time of the job is $T_s(1+\beta)$ (see Table 2). 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.

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

System utilization Wasting allocated slots has also a negative impact on the utilization. If the tasks duration is constant, and the framework acquires a slot every T_s/k on average, the framework will waste roughly $T_s k/2$ computation time during the ramp-up phase. Once it acquire all slots, the framework will use $\beta k T_s$ to complete its job. The utilization achieved by the framework in this case is then $\beta k T_s/(kT_s/2 + \beta k T_s) \simeq \beta/(1/2+\beta)$.

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

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

$\beta/(\ln k - 1 + \beta)$ (see Table 2).

4.2.3 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 in this case when compared to a centralized 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 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_s interval. This is simple because during a T_s 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 x 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 the weighted fair allocation policy where the weight associated with a framework is its intended allocation, s_i. In other words, assuming that each framework has enough demand, framework i will get $x \times s_i/(\sum_{i=1}^{m} s_i)$.

As an example, consider 30 slots that are preferred by two frameworks with intended allocations $s_1 = 200$ slots, and $s_2 = 100$ slots, respectively. Assume that framework 1 requests $r_1 = 40$ of the preferred slots, while framework 2 requests $r_2 = 20$ of these slots. Then, the preferred slots are allocated according to weighted fair sharing, *i.e.,* framework 1 receives $x \times s_1/(s_1+s_2) = 20$ slots, and framework 2 receives the rest of 10 slots. If the demand of framework 1 is less than its share, *e.g.,* $r_1 = 15$, then framework 1 receives 15 slots (which satisfies its demand), and framework 2 receives the rest of 15 slots.

The challenge with 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 fair allocation of the preferred slots described above: simply offer slots to frameworks proportionally to their intended allocations. In particular, when a slot becomes available, Mesos offers 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. Note that this scheme is similar to lottery scheduling [45]. Furthermore, note that since each framework i receives roughly s_i slots during a time interval T_s, the analysis of the ramp-up and completion times in Section 4.2 still holds.

There are also allocation policies other than fair sharing that can be implemented without knowledge of framework preferences. For example, a strict priority scheme (*e.g.,* where framework 1 must always get priority over framework 2) can be implemented by always offering resources to the high-priority framework first.

4.3 Heterogeneous Tasks

So far we have assumed that frameworks have homogeneous task duration distributions. In this section, we discuss heterogeneous tasks, in particular, tasks that are short and long, where the mean duration of the long tasks is significantly longer than the mean of the short tasks. We show that by sharing nodes in space as discussed in Section 3.3.2, Mesos can accommodate long tasks without impacting the performance of short tasks.

A heterogeneous workload can hurt frameworks with short tasks by increasing the time it takes such frameworks to reach their allocation and to acquire preferred slots (relative to the frameworks' total job durations). In particular, a node preferred by a framework with short tasks may become fully filled by long tasks, greatly increasing a framework's waiting time.

To address this problem, Mesos differentiates between short and long slots, and bounds the number of long slots on each node. This ensures there are enough short tasks on each node whose slots become available with high frequency, giving frameworks better opportunities to quickly acquire a slot on one of their preferred nodes. In addition, Mesos implements a revocation mechanism that does not differentiate between long and short tasks once a framework exceeds its allocation. This makes sure that the excess slots in the system (*i.e.,* the slots allocated by frameworks beyond their intended allocations) are all treated as short slots. As a result, a framework under its intended allocation can acquire slots as fast as in a system where all tasks are short. At the same time, Mesos ensures that frameworks running their tasks on long slots and not exceeding their guaranteed allocations won't have their tasks revoked.

4.4 Framework Incentives

Mesos implements a decentralized scheduling approach, where each framework decides which offers to accept or reject. As with any decentralized system, it is important to understand the incentives of various entities in the system. In this section, we discuss the incentives of a framework to improve the response times of its jobs.

Short tasks: A framework is incentivized to use short tasks for two reasons. First, it will be able to allocate any slots; in contrast frameworks with long tasks are restricted to a subset of slots. Second, using small tasks minimizes the wasted work if the framework loses a task, either due to revocation or simply due to failures.

No minimum allocation: The ability of a framework to use resources as soon as it allocates them–instead of waiting to reach a given minimum allocation–would allow the framework to start (and complete) its jobs earlier. Note that the lack of a minimum allocation constraint implies the ability of the framework to *scale up*, while the reverse is not true, *i.e.,* a framework may have both a minimum allocation requirement and the ability to allocate and use resources beyond this minimum allocation.

Scale down: The ability to scale down allows a framework to grab opportunistically the available resources, 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 account for all the resources that a framework owns when deciding which framework to offer resources to next.

We note that these incentives are all well aligned with our goal of improving utilization. When frameworks use short tasks, Mesos can reallocate resources quickly between them, reducing the need for wasted work due to revocation. If frameworks have no minimum allocation and can scale up and down, 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.

4.5 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.

There is another possible bad outcome if allocation modules reallocate resources in a naive 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 up 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 that slave until this minimum amount is free.

Note that the wasted space due to both suboptimal bin packing and fragmentation 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 (e.g., having a cap on task resources) will be able to achieve high utilization even with a distributed scheduling model.

Interdependent framework constraints: It's possible to construct scenarios where, because of esoteric interdependencies between frameworks' performance, only a single global allocation of the cluster resources performs well. We argue such scenarios are rare in practice. In the model discussed in this section, where frameworks only have preferences over placement, we showed that allocations approximate those of optimal schedulers.

Framework complexity: Using resources offers may make framework scheduling more complex. We argue, however, that this difficulty is not in fact onerous. First, whether using Mesos or a centralized scheduler, frameworks need to know their preferences; in a centralized scheduler, the framework would need to express them to the scheduler, whereas in Mesos, it needs to 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 node failures and slow nodes. These policies are easily implemented using the resource offer mechanism.

5 Implementation

We have implemented Mesos in about 10,000 lines of C++. The system runs on Linux, Solaris and Mac OS X.

Mesos applications can be programmed in C, C++, Java, Ruby and Python. We use SWIG [15] to generate interface bindings for the latter three languages.

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 leverage Apache ZooKeeper [4] to perform leader election, as described in Section 3.6. Finally, our current frameworks use HDFS [2] to share data.

Our implementation can use Linux containers [10] or Solaris projects [16] to isolate applications. We currently isolate CPU cores and memory.[3]

We have implemented five frameworks on top of Mesos. First, we have ported three existing cluster systems to Mesos: Hadoop [2], the Torque resource sched-

[3] Support for network and IO isolation was recently added to the Linux kernel [9] and we plan to extend our implementation to isolate these resources too.

uler [42], and the MPICH2 implementation of MPI [22]. 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 called Spark, that we discuss in Section 5.3. Finally, to show the applicability of running front-end frameworks on Mesos, we have developed a framework that manages an elastic web server farm which we discuss in Section 5.4.

5.1 Hadoop Port

Porting Hadoop to run on Mesos required relatively few modifications, because Hadoop concepts such as map and reduce tasks correspond cleanly to Mesos abstractions. In addition, the Hadoop "master", known as the JobTracker, and Hadoop "slaves", known as TaskTrackers, naturally fit into the Mesos model as a framework scheduler and executor.

We first modified the JobTracker to schedule MapReduce tasks using resource offers. Normally, the JobTracker schedules tasks in response to heartbeat messages sent by TaskTrackers every few seconds, reporting the number of free slots in which "map" and "reduce" tasks can be run. The JobTracker then assigns (preferably data local) map and reduce tasks to TaskTrackers. To port Hadoop to Mesos, we reuse the heartbeat mechanism as described, but dynamically change the number of possible slots on the TaskTrackers. When the JobTracker receives a resource offer, it decides if it wants to run any map or reduce tasks on the slaves included in the offer, using delay scheduling [48]. If it does, it creates a new Mesos task for the slave that signals the TaskTracker (i.e. the Mesos executor) to increase its number of total slots. The next time the TaskTracker sends a heartbeat, the JobTracker can assign the runnable map or reduce task to the new empty slot. When a TaskTracker finishes running a map or reduce task, the TaskTracker decrements it's slot count (so the JobTracker doesn't keep sending it tasks) and reports the Mesos task as finished to allow other frameworks to use those resources.

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, which 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, we added 1,100 lines of new code to Hadoop.

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 "wrapper" and a Mesos executor "wrapper" written in 360 lines of Python that invoke different components of Torque as appropriate. In addition, we had to modify 3 lines of Torque source code in order to allow it to elastically scale up and down on Mesos depending on the jobs in its queue.

The scheduler wrapper first configures and launches a Torque server and then periodically monitors the server's job queue. While the queue is empty, the scheduler wrapper refuses all resource offers it receives. Once a job gets added to Torque's queue (using the standard `qsub` command), the scheduler wrapper informs the Mesos master it can receive new offers. As long as there are jobs in Torque's queue, the scheduler wrapper accepts as many offers to satisfy the constraints of its jobs. When the executor wrapper is launched it starts a Torque backend daemon that registers with the Torque server. When enough Torque backend daemons have registered, the torque server will launch the first job in the queue. In the event Torque is running an MPI job the executor wrapper will also launch the necessary MPI daemon processes.

Because jobs that run on Torque are typically not resilient to failures, Torque never accepts resources beyond its guaranteed allocation to avoid having its tasks revoked. The scheduler wrapper will accept up to its guaranteed allocation whenever it can take advantage of those resources, such as running multiple jobs simultaneously.

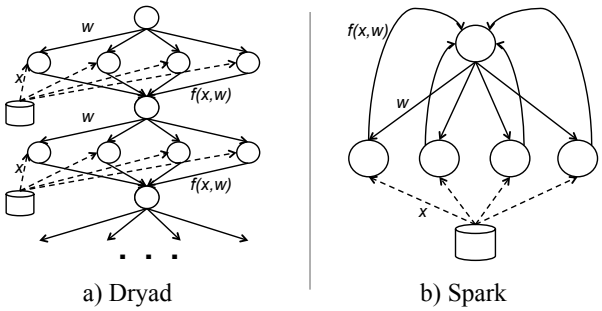
In addition to the Torque framework, we also created a Mesos "wrapper" framework, written in about 200 lines of Python code, for running MPI jobs directly on Mesos.

5.3 Spark Framework

To show the value of simple but specialized frameworks, we built Spark, a new framework for *iterative jobs* that was motivated from discussions with machine learning researchers at our institution.

One iterative algorithm used frequently in machine learning is logistic regression [11]. An implementation of logistic regression in Hadoop must run each iteration as a separate MapReduce job, because each iteration depends on values computed in the previous round. In this case, every iteration must re-read the input file from disk 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 into memory 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 data set in memory at each executor, and then run multiple iterations on this cached data. This caching is



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 processes across iterations, only loading data once.

achieved in a fault-tolerant manner: if a node is lost, Spark remembers how to recompute its slice of the data.

Spark leverages Scala to provide a language-integrated syntax similar to DryadLINQ [47]: users invoke parallel operations by applying a function on a special "distributed dataset" object, and the body of the function is captured as a closure to run as a set of tasks in Mesos. Spark then schedules these tasks to run on executors that already have the appropriate data cached, using delay scheduling. By building on-top-of Mesos, Spark's implementation only required 1300 lines of code.

Due to lack of space, we have limited our discussion of Spark in this paper and refer the reader to [49] for details.

5.4 Elastic Web Server Farm

We built an elastic web farm framework that takes advantage of Mesos to scale up and down based on external load. Similar to the Torque framework, the web farm framework uses a scheduler "wrapper" and executor "wrapper". The scheduler wrapper launches an haproxy [6] 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.

6 Evaluation

We evaluated Mesos by performing a series of experiments using Amazon's EC2 environment.

6.1 Macrobenchmark

To evaluate the primary goal of Mesos, which is enabling multiple diverse frameworks to efficiently share the same cluster, we ran a macrobenchmark consisting of a mix of fine and course-grained frameworks: three Hadoop frameworks and a single Torque framework. We submitted a handful of different sized jobs to each of these four frameworks. Within the Hadoop frameworks, we ran

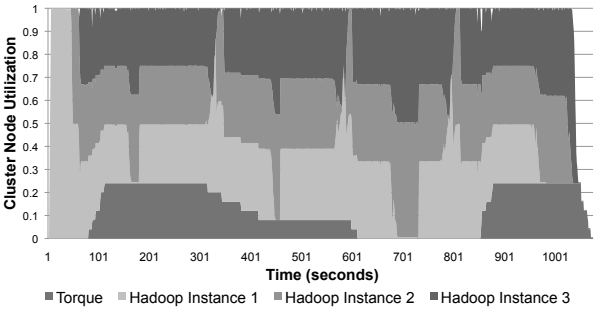


Figure 5: Cluster utilization over time on a 50-node cluster with three Hadoop frameworks each running back-to-back MapReduce jobs, plus one Torque framework running a mix of 8 node and 24 node HPL MPI benchmark jobs.

Application	Duration Alone	Duration on Mesos
MPI LINPACK	50.9s	51.8s
Hadoop WordCount	159.9s	166.2s

Table 3: Overhead of MPI and Hadoop benchmarks on Mesos.

back-to-back WordCount MapReduce jobs, and within the Torque framework we submitted jobs running the High-Performance LINPACK [19] benchmark (HPL).

This experiment was performed using 50 EC2 instances, each with 4 CPU cores and 15 GB RAM. Figure 5 shows node utilization over time. The guaranteed allocation for the Torque framework was 48 cores (1/4 of the cluster). Eight MPI jobs were launched, each using 8 nodes, beginning around time 80s. The Torque framework launches tasks that use only one core. Six of the HPL jobs were able to run as soon as they were submitted, bringing the Torque framework up to its guaranteed allocation of 48 cores. Thereafter, two of them were placed in the job queue and were launched as soon as the first and second HPL jobs finished (around time 315s and 345s). The Hadoop instances tasks fill in any available remaining slots, keeping cluster utilization at 100%.

6.2 Overhead

To measure the overhead Mesos imposes on existing cluster computing frameworks, we ran two benchmarks

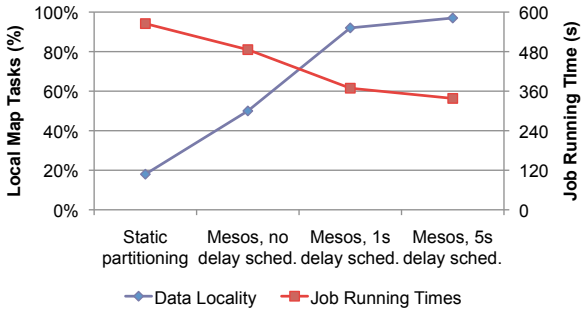


Figure 6: Data locality and average job running times for 16 Hadoop instances on a 93-node cluster using static partitioning, Mesos, or Mesos with delay scheduling.

using the MPI (not with Torque) and Hadoop. These experiments were performed using 50 EC2 instances, each with 2 CPU cores and 6.5 GB RAM. We used the High-Performance LINPACK [19] benchmark for MPI and the WordCount workload for Hadoop. Table 3 shows average running times across three runs of MPI and Hadoop both with and without Mesos.

6.3 Data Locality through Fine-Grained Sharing and Resource Offers

In this experiment, we demonstrate 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 instances, each with 4 CPU cores and 15 GB RAM. Each instance ran a map-only scan job that searched a 100 GB file spread throughout the cluster on a shared HDFS file system. Each job printed 1% of the input records as output. 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 sharing mechanisms), and all instances on Mesos using either no delay scheduling, 1s delay scheduling or 5s delay scheduling.

Figure 6 shows averaged measurements from all 16 Hadoop instances across 3 runs of each scenario. Using static partitioning yields very low (18%) data locality 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 takes turns accessing all the nodes in the cluster. Adding a 1-second delay brings locality above 90%, and a 5-second delay achieves 95% locality, which is competitive with running Hadoop alone. As expected, the average performance of each Hadoop instance improves with locality: jobs run 1.7x faster in the 5s delay scenario than with static partitioning.

6.4 Benefit of Specialized Frameworks

We evaluated the benefit of running iterative jobs using the specialized Spark framework (Section 5.3) over the general-purpose Hadoop framework. We used a logistic regression job implemented in Hadoop by local machine learning researchers, and implemented a second version using Spark and Mesos. We ran each version separately on 20 EC2 instances, 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 20 (see Figure 7).

With Hadoop, each iteration takes 127s on average, while with Spark, the first iteration takes 174s, but subsequent iterations take about 6 seconds. Basically, the time to evaluate the function for each iteration is dominated by the time to read the input data from HDFS and parse it into floating point numbers. Hadoop incurs the read/parsing cost for each iteration, while Spark reuses cached blocks of parsed data and only incurs the cost once (using slower text-parsing routines), leading to a 10.7x speedup on 30 iterations.

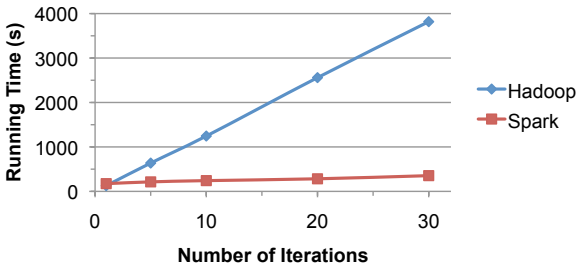


Figure 7: Hadoop and Spark logistic regression running times.

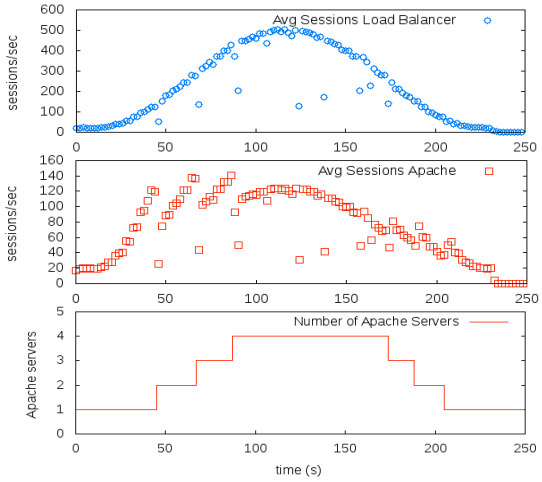


Figure 8: The average session load on the load balancer, the average number of sessions across each web server, and the number of web servers running over time.

6.5 Elastic Web Farm

To demonstrate an interactive framework dynamically scaling on Mesos we ran an elastic web farm on Mesos. We used HTTPerf [38] 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 4 EC2 instances with 8 CPU cores and 6.5 GB RAM. Figure 8 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.

6.6 Mesos Scalability

To evaluate Mesos' scalability, we emulated large clusters by using 99 Amazon EC2 instances, each with 8 CPU cores and 6 GB RAM. We used one EC2 instance for the master with the remaining instances each running multiple slaves. During the experiment, each of 200 frameworks randomly distributed throughout the cluster continuously launches one task at a time for each slave that it receives an offer. Each task sleeps for a specified period of time, based on a normal distribution around 10 seconds with a standard deviation of 2, and then ends. We choose 10 seconds to help demonstrate Mesos scalability under high load. In practice, average task runtime will be higher, yielding lower load on the master. Once the cluster reaches steady-state (i.e., the 200 frameworks achieve their fair share and all cluster resources are in use), we launch a single framework from a random slave within the cluster that runs a single 10 second task. Note that each slave reports 8 CPU cores and 6 GB RAM, and each framework runs tasks that consume 4 CPU cores and 3 GB RAM. Figure 9 shows the average of 5 runtimes for launching the single framework after reaching steady-state.

Our initial Mesos implementation (labeled "reactive" in Figure 9) failed to scale beyond 15,000 slaves because it attempted to allocate resources immediately after every task finished. While the allocator code is fairly simple, it dominated the master's execution time, causing latencies to increase. To amortize allocation costs, we modified Mesos to perform allocations in batch intervals. To demonstrate how well this implementation scaled we ran the same experiment as before, however we also tried variations with average task lengths of 10 and 30 seconds (these are labeled as "10 s" and "30 s" in Figure 9). As the graph shows, with 30 second average task length Mesos imposes less than 1 second of additional overhead on frameworks. Unfortunately, the EC2 virtualized environment limited scalability beyond 50,000 slaves, as at 50,000 slaves the master was processing 100,000 packets per second (in+out), which has been shown to be the current achievable limits in EC2[14].

6.7 Fault Tolerance

To evaluate recovery from master failures, we conducted an experiment identical to the scalability experiment, except that we used 20 second task lengths, and two Mesos masters connected to a 5 node ZooKeeper quorum using a default tick timer set to 2 seconds. We ran the experiment using 62 EC2 instances, each with 4 CPU cores and 15 GB RAM. 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. Figure 10 shows the average MTTR with a 95% confidence interval for different cluster sizes. Although not shown, we experimented with different sizes of ZooKeeper quorums (between 1 and 11). The results consistently showed that MTTR is always below 15 seconds, and typically 7-8 seconds.

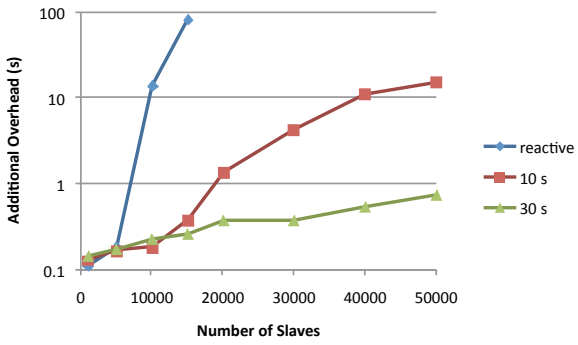


Figure 9: Mesos master's scalability versus number of slaves.

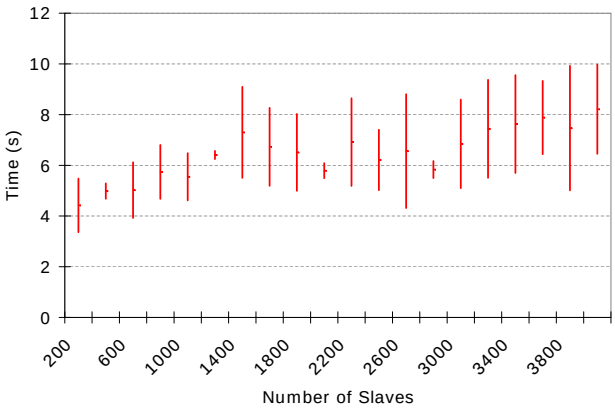


Figure 10: Mean time for the slaves and applications to reconnect to a secondary Mesos master upon master failure. The plot shows 95% confidence intervals.

7 Related Work

HPC and Grid schedulers. The high performance computing (HPC) community has long been managing clusters [42, 51, 28, 20]. Their target environment typically consists of specialized hardware, such as Infiniband, SANs, and parallel filesystems. Thus 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 smaller fine-grained tasks. Consequently, a job does not dynamically grow or shrink its resource demands across machines during its lifetime. Moreover, fault-tolerance is achieved through checkpointing, rather than recomputing fine-grained tasks. For these reasons, HPC schedulers use centralized scheduling, and require jobs to declare the required resources at job submission time. Jobs are then allocated course-grained allocations of the cluster. Unlike the Mesos approach, this does

not allow frameworks to locally access data distributed over the cluster. Furthermore, jobs cannot grow and shrink dynamically as their allocations change. In addition to supporting fine-grained sharing, Mesos can run HPC schedulers, such as Torque, as a frameworks, which then can schedule HPC workloads appropriately.

Grid computing has mostly focused on the problem of making diverse virtual organizations share geographically distributed and separately administered resources in a secure and inter-operable way. Mesos could well be used within a virtual organization, which is part of a larger grid that, for example, runs Globus Toolkit.

Public and Private Clouds. Virtual machine clouds, such as Amazon EC2 [1] and Eucalyptus [40] share common goals with Mesos, such as isolating frameworks while providing a low-level abstraction (VMs). However, they differ from Mesos in several important ways. First, these systems share resources in a coarse-grained manner, where a user may hold onto a virtual machine for an arbitrary amount of time. This makes fine-grained sharing of data difficult Second, these systems generally do not let applications specify placement needs beyond the size of virtual machine they require. In contrast, Mesos allows frameworks to be highly selective about which resources they acquire through resource offers.

Quincy. Quincy [34] is a fair scheduler for Dryad. It uses a centralized scheduling algorithm based on min-cost flow optimization and takes into account both fairness and locality constraints. Resources are reassigned by killing tasks (similarly to Mesos resource killing) when the output of the min-cost flow algorithm changes. Quincy assumes that tasks have identical resource requirements, and the authors note that Quincy's min-cost flow formulation of the scheduling problem is difficult to extend to tasks with multi-dimensional resource demands. In contrast, Mesos aims to support *multiple* cluster computing frameworks, including frameworks with scheduling preferences other than data locality. Mesos uses a decentralized two-level scheduling approach lets frameworks decide where they run, and supports heterogeneous task resource requirements. We have shown that frameworks can still achieve near perfect data locality using delay scheduling.

Specification Language Approach. Some systems, such as Condor and Clustera [43, 26], go to great lengths to match users and jobs to available resources. Clustera provides multi-user support, and uses a heuristic incorporating user priorities, data locality and starvation to match work to idle nodes. However, Clustera requires each job to explicitly list its data locality needs, and does not support placement constraints other than data locality. Condor uses the ClassAds [41] to match node properties to job needs. This approach of using a resource specification language always has the problem that certain preferences might currently not be possible to express. For example, delay scheduling is hard to express with the current languages. In contrast, the two level scheduling and resource offer architecture in Mesos gives jobs control in deciding where and when to run tasks.

8 Conclusion

We have presented Mesos, a resource management substrate that allows diverse parallel applications to efficiently share a cluster. Mesos is built around two contributions: a fine-grained sharing model where applications divide work into smaller tasks, and a distributed scheduling mechanism called resource offers that lets applications choose which resources to run on. Together, these contributions let Mesos achieve high utilization, respond rapidly to workload changes, and cater to applications with diverse placement preferences, while remaining simple and scalable. We have shown that existing cluster applications can effectively share resources with Mesos, that new specialized cluster computing frameworks, such as Spark, can provide major performance gains, and that Mesos's simple architecture enables the system to be fault tolerant and to scale to 50,000 nodes.

Mesos is inspired by work on microkernels [18], exokernels [27] and hypervisors [32] in the OS community and by the success of the narrow-waist IP model [23] in computer networks. Like a microkernel or hypervisor, Mesos is a stable, minimal core that isolates applications sharing a cluster. Like an exokernel, Mesos aims to give applications maximal control over their execution. Finally, like IP, Mesos encourages diversity and innovation in cluster computing by providing a "narrow waist" API that lets diverse applications coexist efficiently.

In future work, we intend to focus on three areas. First, we plan to further analyze the resource offer model to characterize environments it works well in and determine whether any extensions can improve its efficiency while retaining its flexibility. Second, we plan to use Mesos as a springboard to experiment with cluster programming models. Lastly, we intend to build a stack of higher-level programming abstractions on Mesos to allow developers to quickly write scalable, fault-tolerant applications.

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. http://hadoop.apache.org/zookeeper/.

[5] Hadoop and hive at facebook. http://www.slideshare.net/dzhou/facebook-hadoop-data-applications.

[6] HAProxy Homepage. http://haproxy.1wt.eu/.

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

[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] Logistic Regression Wikipedia Page. http://en.wikipedia.org/wiki/Logistic_regression.

[12] Personal communication with Dhruba Borthakur from the Facebook data infrastructure team.

[13] Personal communication with Khaled Elmeleegy from Yahoo! Research.

[14] RightScale webpage. http://blog.rightscale.com/2010/04/01/benchmarking-load-balancers-in-the-cloud.

[15] Simplified wrapper interface generator. http://www.swig.org.

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

[17] Twister. http://www.iterativemapreduce.org.

[18] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, pages 93–113, 1986.

[19] 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: Proceedings of the 1990 conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[20] D. Angulo, M. Bone, C. Grubbs, and G. von Laszewski. Workflow management through cobalt. In *International Workshop on Grid Computing Environments–Open Access (2006)*, Tallahassee, FL, November 2006.

[21] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[22] 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 *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[23] D. D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, pages 106–114, 1988.

[24] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. Mapreduce online. In *7th Symposium on Networked Systems Design and Implementation*. USENIX, May 2010.

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

[26] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[27] 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.

[28] W. Gentzsch. Sun grid engine: towards creating a compute power grid. pages 35–36, 2001.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of heterogeneous resources in datacenters. Technical Report UCB/EECS-2010-55, EECS Department, University of California, Berkeley, May 2010.

[30] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[31] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, , and L. Zhou. Comet: Batched stream processing in data intensive distributed computing. Technical Report MSR-TR-2009-180, Microsoft Research, December 2009.

[32] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM Systems Journal*, 18(1):111–142, 1979.

[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, pages 59–72, 2007.

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

[35] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HOTOS '09: Proceedings of the Twelfth Workshop on Hot Topics in Operating Systems*. USENIX, May 2009.

[36] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Programming bulk-incremental dataflows. Technical report, University of California San Diego, http://cseweb.ucsd.edu/~dlogothetis/docs/bips_techreport.pdf, 2009.

[37] 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 *PODC '09*, pages 6–6, New York, NY, USA, 2009. ACM.

[38] D. Mosberger and T. Jin. httperf – a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[39] H. Moulin. *Fair Division and Collective Welfare*. The MIT Press, 2004.

[40] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications [Online]*, Chicago, Illinois, 10 2008.

[41] R. Raman, M. Solomon, M. Livny, and A. Roy. The classads language. pages 255–270, 2004.

[42] G. Staples. Torque - torque resource manager. In *SC*, page 8, 2006.

[43] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concur-*

rency and Computation - Practice and Experience, 17(2-4):323–356, 2005.

[44] H. Varian. Equity, envy, and efficiency. *Journal of Economic Theory*, 9(1):63–91, 1974.

[45] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 1+, Berkeley, CA, USA, 1994. USENIX Association.

[46] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 247–260, New York, NY, USA, 2009. ACM.

[47] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[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 2010*, April 2010.

[49] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.

[50] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. OSDI 2008*, Dec. 2008.

[51] S. Zhou. Lsf: Load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing*, Tallahassee, FL, December 1992.