Nexus: A Common Substrate for Cluster Computing

Benjamin Hindman,   Andy Konwinski,   Matei Zaharia,
Ali Ghodsi,   Anthony D. Joseph,   Scott Shenker,   Ion Stoica

University of California, Berkeley

{benh,andyk,matei,alig,adj,shenker,stoica}@cs.berkeley.edu

Abstract

Cluster computing has become mainstream, resulting in the rapid creation and adoption of diverse cluster computing frameworks. We believe that no single framework will be optimal for all applications, and that organizations will instead want to run *multiple* frameworks in the same cluster. Furthermore, to ease development of new frameworks, it is critical to identify common abstractions and modularize their architectures. To achieve these goals, we propose Nexus, a low-level substrate that provides isolation and efficient resource sharing across frameworks running on the same cluster, while giving each framework maximum control over the scheduling and execution of its jobs. Nexus fosters innovation in the cloud by letting organizations run new frameworks alongside existing ones and by letting framework developers focus on specific applications rather than building one-size-fits-all frameworks.

1   Introduction

Cluster computing has become mainstream. Industry and academia are running applications ranging from finance to physics on clusters of commodity servers [6], fueled by open-source platforms like Hadoop [3] and cloud services like EC2 [1]. Driven by this wide range of applications, researchers and practitioners have been developing a multitude of cluster computing frameworks. MapReduce [23] provided a simple, low-level programming model. Sawzall [39] and Pig [38] developed higher-level programming models on top of MapReduce. Dryad [32] provided a more general execution layer – data flow DAGs. Recently, Google announced Pregel [35], a specialized framework for graph computations.

It seems clear that frameworks providing new programming models, or new implementations of existing models, will continue to emerge, and that no single framework will be optimal for all applications. Conse-

quently, organizations will want to run *multiple frameworks*, choosing the best framework for each application. Furthermore, for economic reasons, organizations will want to run these frameworks *in the same cluster*. Sharing a cluster between frameworks increases utilization and allows the frameworks to share large data sets that may be too expensive to replicate.

To allow frameworks to share resources, developers have so far taken the approach of building frameworks on top of a common execution layer that performs sharing. For example, Pig turns SQL-like queries into series of Hadoop jobs. Unfortunately, this approach may limit performance – for example, Pig cannot pipeline data between MapReduce stages because they are separate Hadoop jobs. Other efforts have proposed more general execution layers, such as Dryad [32], on which a variety of frameworks can run. However, general execution layers are more complex than specialized ones, and they still incur the risk that a new programming model cannot be expressed over the execution layer – for example, the Bulk Synchronous Processes model used in Pregel, where long-lived processes exchange messages, cannot be expressed as an acyclic data flow in Dryad.

The problem with the single execution layer approach is that a single entity is performing two tasks: isolating resources *between* jobs and managing execution *within* a job. We think that a far better approach, following the exokernel model [25], is to define a small resource isolating kernel that is independent of any framework, and let each framework control its own internal scheduling and execution. In this paper, we propose Nexus, an isolation and resource sharing layer for clusters based on this design. Nexus only places a minimal set of requirements on frameworks to enable efficient resource sharing. Beyond that, it aims to give frameworks maximum control over their scheduling and execution.

Sharing clusters is not a new problem. Multiple cluster schedulers have been developed in the High Performance Computing (HPC) and Grid communities [42, 41, 30, 47,

19]. As a general rule, these systems ask users to submit jobs to a queue and to request a number of resources (machines, cores, etc) for each job.[1] Jobs are then launched when enough machines become free. While this model works well for batch-oriented HPC workloads, it faces two important challenges in the environments that frameworks like Hadoop are used in:

1. **Dynamic workloads:** Clusters are being used for interactive ad-hoc queries and time-sensitive production applications in addition to batch jobs. For example, Facebook's Hadoop data warehouse is used for both production jobs and arbitrary user jobs [45]. Having production jobs wait in line behind a large user job would be unacceptable, as would reserving part of the cluster for production jobs and leaving it idle when they are absent. Instead, schedulers for Hadoop [45] and Dryad [33] implement fair sharing, varying jobs' allocations dynamically as new jobs are submitted. This is possible because Hadoop and Dryad jobs consist of small independent tasks, so the number of machines a job is running on can change during its lifetime.

2. **Framework semantics:** MapReduce and Dryad achieve high performance by placing computations on the nodes that contain their input data. While some grid schedulers let jobs specify locality constraints at the level of geographic sites, most do not let jobs control their placement at the level of nodes. The problem is that the schedulers are not aware of framework *semantics* that should be taken into account in scheduling decisions.

These challenges are caused by two core problems: First, the *granularity* at which existing schedulers allocate resources is too coarse for dynamic workloads and data sharing. Second, framework *semantics* need to be taken into account in scheduling decisions. Nexus addresses these issues through two main design principles:

1. **Fine-grained sharing:** Nexus asks that frameworks split up their work into *tasks*, and makes scheduling decisions at the level of tasks. This is the only major requirement we make of frameworks. Nexus can run tasks from multiple frameworks on the same node, isolating them using various OS mechanisms. Tasks can be viewed as a form of cooperative time-sharing between frameworks.

2. **Two-level scheduling:** Nexus gives frameworks choice in which resources they use through a two-level scheduling model. At the first level, Nexus decides *how many* resources to give each framework based on an organizational *allocation policy* such as fair sharing. At the second level, frameworks

decide *which* of the available resources to use and which tasks to run on each machine. This is achieved through a mechanism called *resource offers*.

Although we have architected Nexus to let organizations define their own allocation policies, we have also defined one policy that will be widely applicable: a generalization of weighted fair sharing to multiple resources that takes into account the fact that frameworks may need different amounts of CPU, memory, and other resources.

Implications The model enabled by Nexus, where a single platform allows multiple frameworks to run on a cluster, has wide-ranging implications for cluster computing. First, this model *accelerates innovation* by letting framework developers build specialized frameworks targeted at particular problems instead of one-size-fits-all abstractions. For example, a researcher that develops a new framework optimized for machine learning jobs can give this framework to a company that primarily uses Hadoop and have it run alongside Hadoop.

Second, the *isolation* that Nexus provides between frameworks is valuable even to organizations that only wish to run a single software package, e.g. Hadoop. First, Nexus allows these organizations to run multiple versions of Hadoop concurrently, e.g. a stable version for production jobs and a faster but less stable version for experimental jobs. Second, organizations may wish to run one separate Hadoop instance per MapReduce job for fault isolation. The stability of the Hadoop master is a serious concern in large multi-user Hadoop clusters [12]; if the master crashes, it takes down all jobs. Nexus lets each job run its own MapReduce master, limiting the impact of crashes.[2] We believe that this second benefit of isolation could drive adoption of Nexus, facilitating the more important first benefit of accelerating innovation.

Finally, we are also exploring using Nexus to share resources between workloads other than data-intensive cluster computing frameworks. For example, we have developed an Apache web farm "framework" that runs multiple, load-balanced Apache servers as its tasks and changes the number of tasks it uses based on load. Sharing resources between front-end and back-end workloads is very attractive to web application providers that experience diurnal load cycles. We have also ported MPI to run over Nexus, allowing a variety of existing scientific applications to share resources with new frameworks.

Evaluation To evaluate Nexus, we have ported two popular cluster computing frameworks to run over it: Hadoop and MPI. To validate our hypothesis that specialized frameworks can provide value over general ones, we have also built a new framework on top of Nexus called Spark, optimized for iterative jobs where a data set is

---

[1]We survey HPC and grid schedulers in Section 8.

[2]The Nexus master is easier to make robust than the Hadoop master because it has a simpler role and it needs to change less often.

2

reused across many short tasks. This pattern is common in machine learning algorithms. Spark provides a simple programming interface and can outperform Hadoop by 8x in iterative workloads. To push the boundaries of the isolation and dynamic scheduling provided by Nexus, we have implemented a load-balanced elastic Apache web server farm. Finally, we have verified that resource offers let Hadoop achieve comparable data locality running on Nexus to running alone.

Contributions   Our main contributions are:

- A cluster computing architecture that separates resource isolation into a small kernel gives frameworks control over their execution and scheduling.

- A two-level scheduling model for fine-grained sharing of clusters based on two abstractions: tasks and resource offers.

- A generalization of weighted fair sharing to multiple resources.

Outline   This paper is organized as follows. Section 2 explains the environment that we have designed Nexus to run in and details its assumptions and goals. Section 3 presents the Nexus architecture. Section 4 presents our current scheduling policy – a generalization of weighted fair sharing for multiple resources. Section 5 describes our implementation of Nexus and of the frameworks we run over it. Section 6 presents experimental results. We include a discussion in Section 7, survey related work in Section 8, and conclude in Section 9.

2   Assumptions and Goals

In this section, we explain the data center environment and workload that Nexus is targeted for, the assumptions it makes, and the goals it seeks to achieve.

2.1   Workload

Our target environment for Nexus is clusters of commodity machines shared by multiple users for analytics workloads. Examples include the back-end clusters at large web companies that MapReduce was developed for, research clusters at universities, and scientific clusters such as the Google/IBM/NSF Cluster Exploratory [11].

As an example of a workload we aim to support, consider the data warehouse at Facebook [4, 8]. Facebook loads logs from its production applications into a 600-node Hadoop cluster, where they are used for applications such as ad targeting, spam detection, and ad-hoc business intelligence queries. The workload includes "production" jobs that directly impact customers, such as identifying spam, long-running "experimental" jobs such as tests for new spam detection algorithms, and "interactive" jobs where an analyst submits a query (e.g. "what fraction of Spanish users post videos") and expects an answer within minutes. If, at some point in time, only a single job is running in the cluster, this job should be allocated all of the resources. However, if a production job, or a job from another user, is then submitted, resources need to be given to the new job within tens of seconds.

To implement dynamic resource sharing, Facebook uses a fair scheduler within Hadoop that works at the granularity of map and reduce tasks [45]. Unfortunately, this means that the scheduler can only handle Hadoop jobs. If a user wishes to write a new spam detection 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 data into it. The goal of Nexus is to enable dynamic resource sharing, including policies such as fair sharing, between *distinct* cluster computing frameworks.

2.2   Allocation Policies

Nexus decides how many resources to allocate to each framework using a pluggable *allocation module*. Organizations may write their own allocation modules, or use the ones we have built. Because of its use in Hadoop and Dryad schedulers [45, 5, 33], the policy we have focused on most is weighted fair sharing. One of our contributions over these Hadoop and Dryad schedulers is that we allow tasks to have heterogeneous resource requirements. We have developed a generalization of weighted fair sharing for multiple resources that we present in Section 4. No matter which allocation policy is used, our goal is to be able to reallocate resources *rapidly* across frameworks when new jobs are submitted, so that jobs can start within tens of seconds of being submitted. This is necessary to support interactive and production jobs in environments like the data warehouse discussed above.

2.3   Frameworks

Organizations will use Nexus to run cluster computing *frameworks* such as Hadoop, Dryad and MPI. As explained in the Introduction, we expect that multiple, isolated instances of each framework will be running.

Nexus aims to give frameworks maximum flexibility, and only imposes a small set of requirements on them to support resource sharing. Most importantly, Nexus asks frameworks to divide their work into units called *tasks*. Nexus makes scheduling decisions at task boundaries – when a task from one framework finishes, its resources can be given to another framework. However, it leaves it up to the frameworks to choose which task to run on which node and which resources to use, through a mechanism called *resource offers* described in Section 3.2.

To allow resources to be reallocated quickly when new frameworks join the system, Nexus expects frameworks to make their tasks short (tens of seconds to minutes in length) when possible. This is sufficiently short that in

a large cluster, there will be tens of tasks finishing per second. For example, in a 600-node cluster with 4 tasks per node and an average task length of one minute, there will be 40 tasks finishing per second, and a new framework could achieve a share of 10% of the cluster (i.e. 240 tasks) in 6 seconds. To encourage frameworks to use short tasks, we make tasks cheap to launch by reusing "executor" processes from the same framework between tasks, as explained in Section 3.1. Interestingly, frameworks such as Hadoop already use short tasks for fault-tolerance: First, having short tasks reduces the time it takes to recover from a failed task. Second, if a node that contains outputs from multiple tasks fails, these tasks can be re-run in parallel on the other nodes in the cluster [23].

If a new framework is not being allocated resources quickly enough because there are too many long tasks running, we also reserve the right to kill tasks from running frameworks after an administrator-specified time-out. Cluster computing frameworks already need to tolerate losing tasks due to hardware failures, so requiring them to tolerate task killing is not onerous. Frameworks with long-lived tasks, such as MPI jobs, may use check-pointing [29] to recover and scale down after tasks are killed. Alternatively, their users can ask the framework to use a smaller share of the cluster than the user's allocation so that it never needs to be killed.

2.4 Summary

We detail the assumptions and goals of Nexus below.

Assumptions:

1. **Frameworks decompose work into tasks.** If rapid reallocation of resources in response to workload changes is desired, then frameworks should either make their tasks short (seconds to minutes) or avoid exceeding their share of the cluster.

2. **Frameworks tolerate losing tasks.** Nexus may kill tasks to enforce scheduling policies.

3. **Single administrative domain.** Our current implementation does not attempt to make security and privacy guarantees in the case of adversarial users.

4. **Allocation policies are at the level of frameworks.** In order to have frameworks map onto organizational entities such as users, separate users are expected to use separate framework instances.

Goals:

1. **Maximum flexibility for frameworks.** Frameworks should be given maximum flexibility in how they schedule and execute their tasks. Scheduling concerns that are common in data-intensive frameworks, such as data locality, should be supported.

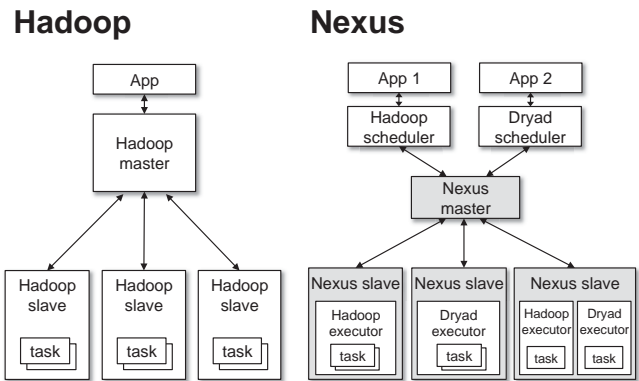2. **Dynamic, responsive scheduling.** Frameworks' allocations should change as new jobs are submitted,

Figure 1: Comparing the architectures of Hadoop and Nexus. The shaded regions make up Nexus. Hadoop can run on top of Nexus, alongside Dryad.

through policies such as fair sharing, priority, etc.

3. **Performance isolation** for processes from different frameworks running on the same node.

4. **Support for pluggable allocation policies.**

5. **Support for heterogeneous resource demands.** Different tasks, even within a framework, may need different amounts of resources (CPU, memory, etc).

6. **Support for heterogeneous nodes.** Cluster nodes may have different amounts of CPU, memory, etc.

3 Nexus Architecture

3.1 System Components

Nexus consists of a *master* process managing a number of *slave* daemons, one on each node in a cluster. Frameworks wishing to use Nexus each register a *scheduler* process with the Nexus master in order to be assigned resources (registration also associates a user with the framework, for use by allocation policies). Resource allocation is performed through a mechanism called *resource offers* that we describe in Section 3.2, which lets frameworks pick which available resources to use. When a framework accepts resources on a particular slave, it can launch tasks to use the resources by giving Nexus an opaque *task descriptor* for each task. Each descriptor is passed to a framework-specific *executor* process that Nexus starts on the slave,[3] which runs the task. The executor is shared by all of a framework's tasks on a given slave, allowing the framework to amortize initialization costs and to keep data cached in memory between tasks.[4] An executor is free to run each task in its own thread, or

---

[3]The executor is fetched from a shared file system such as HDFS.
[4]If a slave accrues idle executors from too many frameworks, however, Nexus may kill some of them when the slave is low on memory.

to spawn any number of separate processes for each task, depending on the level of isolation it desires between its tasks. Executors from different frameworks are isolated from one another, as described in Section 3.3. Finally, as executors finish tasks, they send *status updates* to inform the master that resources can be re-allocated.

Nexus reports failures to frameworks but expects them to implement their own recovery mechanisms. In the event of a task failure, an executor can send a status update signaling the loss of the task. In the event of an executor failure, the executor and its child processes are killed and the framework's scheduler is informed of the lost executor. Finally, in the event of a node failure, the framework's scheduler is informed of a lost slave.

Figure 1 shows the components of Nexus and compares them with those of Hadoop. Many elements of Nexus—a scheduler, executors, tasks, and status updates—map closely to elements of Hadoop and other cluster computing frameworks such as Dryad. Nexus factors these elements out into a common layer, reducing the burden on framework developers and giving frameworks a common API for accessing resources so that they can share clusters. The close mapping of Nexus components to components of existing frameworks also makes porting frameworks to Nexus fairly straightforward. For example, our port of Hadoop reuses Hadoop's master as a scheduler and Hadoop's slave as an executor, and just changes the communication between them to pass through Nexus.

We note that Nexus imposes no storage or communication abstractions on frameworks. We expect data to be shared through a distributed file system such as GFS [27] or Hadoop's HDFS [3] running on each slave alongside Nexus. Tasks are free to communicate using sockets.

3.2 Resource Allocation

The main challenge with Nexus's two-level scheduling design is ensuring that each framework gets resources that it wishes to use. For example, a Hadoop job might want to run tasks on the nodes that contain its input file, an iterative job might prefer to run tasks on nodes where it already has executors to re-use data cached in memory, and an MPI job might need 4 GB of RAM per task.

Our solution to this problem is a decentralized two-level scheduling mechanism called *resource offers*. At the first level, an *allocation module* in the master decides which framework to offer resources to when there are free resources in the system, following an organizational policy such as fair sharing. At the second level, frameworks' schedulers may accept or reject offers. If a framework rejects an offer, it will continue to be "below its share," and it will therefore be offered resources first in the future when resources on new nodes become free.

In detail, whenever there are free resources, Nexus performs the following steps:

1. **Allocate:** The allocation module determines which framework(s) to offer the free resources to and how much to offer to each framework.

2. **Offer:** Frameworks are sent a list of resource offers. A resource offer is a *(hostname, resources)* pair, where "resources" is a vector containing the number of free CPU cores, GB of RAM, and potentially other resources on a given machine.

3. **Accept/Reject:** Frameworks respond to offers with a possibly empty list of *(task descriptor, hostname, resources)* tuples. A framework is free to claim as many total resources on a machine as were in the offer and to divide these between several tasks.

A few more details are needed to complete the model. First, frameworks will often *always* reject certain resource offers. To short-circuit the rejection process, frameworks can provide *filters* to the master. Two types of filters we support are "only offer machines from list L" and "only offer machines with at least R resources free". A filter may last indefinitely, or get decommissioned after a framework-specified timeout. A framework may also choose to update its filters at any time. By default, any resources rejected during an offer have a temporary 1-second filter placed on them, to minimize the programming burden on framework developers who do not wish to constantly update their filters.

Second, to accommodate frameworks with large resource requirements per task, Nexus allows administrators to set a *minimum offer size* on each slave, and it will not offer any resources on that slave until this minimum amount is free. Without this feature, a framework with large resource requirements (e.g. 2 CPUs and 4 GB RAM per task) might starve in a cluster that is filled by tasks with small requirements (e.g. 1 CPU and 1 GB RAM), because whenever a small task finishes and its resources are re-offered, the framework with large requirements cannot accept the offer but frameworks with smaller requirements can. We currently ask administrators to manually configure per-machine minimum offer sizes, because there is a tradeoff between the largest task supported and the amount of fragmentation incurred when resources wait idle for a large enough offer to form. We are investigating making this process more automatic.

3.2.1 Example

To illustrate how resource offers work, suppose that two MapReduce frameworks, F1 and F2, each wish to run a map function on a large data set that is distributed across all of a cluster's nodes. Suppose that the both frameworks' tasks require 1 CPU core and 1 GB of RAM each, and that each node has 2 cores and 2 GB of RAM. Finally, suppose that the cluster's allocation policy is fair

sharing: each framework should get an equal number of tasks.[5] Then, the allocation module might implement the following algorithm: "whenever resources become free, offer them to the framework with the fewest resources whose filters do not block the offer."

When the first framework, say F1, registers, it is offered all of the resources in the cluster and starts tasks everywhere. After F2 registers, it is offered the resources that free up as tasks from F1 finish, until both frameworks have an equal number of resources. At this point, some nodes will be running one task from each framework, some nodes will be running two tasks from F1, and some nodes will be running two tasks from F2. For some time, both frameworks will stay running in the same set of locations, because whenever a task from a particular framework finishes, its resources will be offered back to the same framework (because that framework is now below its fair share), and the framework will launch a new task on the same node. If this situation continued indefinitely, data locality would suffer because F1 would never get to run on nodes that only F2 has tasks on, and vice-versa; we call this problem *sticky slots* [45]. However, once one of the frameworks, say F1, finishes reading all the data that it wanted to read on a particular node, it starts filtering out resources on this node. These resources are offered to F2, which accepts them. F2 now has a higher share of the cluster than F1. Therefore, when *any* of F2's tasks finishes, its resources are offered to F1. Consequently, F1 will be able to take resources on nodes that F2 was previously monopolizing, in effect "swapping places" with F1. Both frameworks will thus get a chance to run on all nodes and achieve high data locality.

3.2.2 Discussion

Our resource offer mechanism differs from the scheduling mechanisms used in most cluster schedulers because it is decentralized. An alternative approach would be to have each framework give Nexus a set of preferences about resources it wants, specified in some "preference language", and have Nexus match frameworks with resources using a centralized algorithm.

At first, the centralized approach appears attractive because it gives Nexus global knowledge about frameworks' needs. However, this approach has an important disadvantage: *preferences that cannot be expressed in the preference language cannot be accounted for*. Because Nexus aims to support a wide variety of both current and future frameworks, it seems unlikely that a single preference language could be designed that is expressive enough for all frameworks, easy for developers to use, and useful for making scheduling decisions.

In fact, even the preferences of a fairly simple frame-

work like MapReduce are complex: A MapReduce job first wants to run a number of map tasks, each on one of the nodes that has a replica of its input block. However, if nodes with local data cannot be obtained, the job prefers running tasks in the same rack as one of the input blocks. After some fraction of maps finish, the job also wants to launch reduce tasks, which may have different CPU and memory requirements than maps, to start fetching map outputs. Finally, until all the maps are finished, the job never wants to end up with *only* reduces running; if Nexus started sending the job only resource offers with CPU and memory amounts suitable for reduces, then the job would never finish.

Advocates of a preference language based approach argue that scheduling decisions based on global knowledge will be better than any distributed decisions made by frameworks responding to offers. However, we have found that for many preferences, such as data locality, resource offers achieve nearly optimal performance (e.g. 95% of tasks run on the nodes that contain their input). We discuss this in Section 6.4.

Our resource offer approach has the added advantage that it is simple and efficient to implement, allowing the Nexus master to be scalable and reliable. We hypothesize that the ability to run a wide variety of frameworks on a single, stable platform will offset any small losses in performance caused by using resource offers.

3.3 Framework Isolation

Nexus aims to provide performance isolation on slave nodes between framework executors. Given the fine grained nature of our tasks, an isolation mechanism should have the following properties:

- An isolation mechanism *should not* impose high overheads for executor startup and task execution.

- An isolation mechanism *should* allow Nexus to dynamically change resource allocations after the executors have been started, as the number of tasks an executor is running may change during its lifetime.

While isolation is a first-order concern of our architecture, we believe we should be able leverage existing isolation mechanisms. Unfortunately, even given the substantial amount of attention performance isolation has received, no solution has emerged that is optimal for both properties. Given that sites will have varying isolation needs, Nexus is designed to be able to support multiple isolation mechanisms through pluggable modules. We explain the mechanism we currently use in Section 5.1.

3.4 Resource Revocation

We discuss the mechanism of resource revocation in this section, and defer to Section 4.3 for an analysis of when

---

[5]We discuss how to define weighted fair sharing for multiple resources in depth in Section 4.

revocation is needed, and the magnitude of revocation that might need to be done.

Revocation policies will depend greatly on site-specific needs and desires. In general, however, the Nexus allocation module, will initiate revocation when it wants to ameliorate a disparity in some framework's fair share. This may occur (a) when a new framework registers or (b) when an existing framework, who is below its fair share, revives resource offers when there are insufficient, or no, available resources.

Similar to previous work, Nexus uses *visible* resource revocation [25]. That is, the Nexus allocation module informs a framework that it needs to return a specified amount of resources to the cluster within a specified timeout (which might be statically configured, or dynamically determined). We augment this basic revocation scheme, however, by including revocation constraints; a revocation request will specify some subset of nodes from which the resources must be returned, where that subset may be one node, or the entire cluster. If a framework fails to return enough resources by the timeout, the Nexus scheduler may choose to reclaim resources itself. It may, however, choose to reclaim only a subset of the resources in the original request. A framework scheduler receives "task killed" updates for all tasks during this process.

Using this revocation mechanism allows the Nexus allocation module to provide maximal flexibility for a framework when that flexibility is affordable. Often, this will provide a framework some freedom to pick which tasks to kill (if it needs to kill any tasks at all: a framework may have been offered resources that it hasn't yet accepted that satisfy the revocation constraints).

4   Fair Scheduling

As described in Section 3, Nexus is a two-level scheduler that lets users influence scheduling decisions by their selection of offers. This is done when a user, which Nexus considers to be below its *fair share*, is offered slots. The user can then reject undesirable offers, and accept desirable ones. In this section, we try to answer the question of what the fair share of a user should be. This is complicated by that there are multiple different resources, and users have possibly different demands and constraints on them. We provide fairness properties, scheduling algorithms, and a mechanism for killing.

Our model is that the system consists of a discretized amount of different resources, such as 32 CPU slots, 256 GB memory, and 10 GB of disk space. Each user implicitly, or explicitly, defines a *demand vector*, specifying the per-task resource usage, e.g. $\langle 2, 5, 100 \rangle$, implying tasks should each be given 2 CPUs etc. At runtime, the above
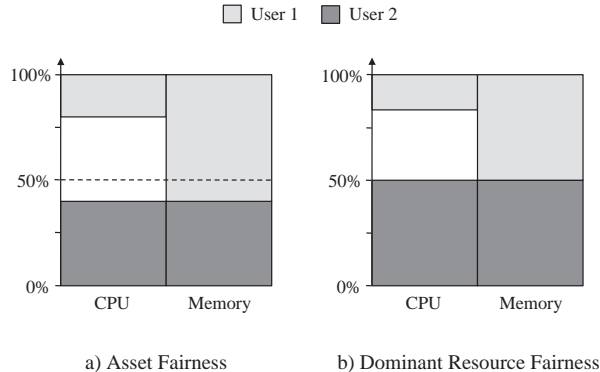

Figure 2: Example with two frameworks with demand vectors $\langle 1, 3 \rangle$ and $\langle 1, 1 \rangle$. a) Allocation under Asset Fairness, equalizing their total resource usage. b) Allocation under Dominating Resource Fairness, equalizing the share of their dominating resource.

vectors change dynamically. Thus, Nexus' scheduler approximates this model.

We will refer to the following canonical example throughout this section. Consider a cluster with 300 CPUs and 300 GB memory, and two users with demand vectors $\langle 1, 3 \rangle$ and $\langle 1, 1 \rangle$, respectively.

First attempt: Asset Fairness.   One intuitive scheduling policy we tried is to account for all resources that a user uses. We refer to this as *asset fairness*. The goal would then be to schedule users such that every user's sum of all resources is the same. This might seem natural since the usage of a chunk of each resource can be equated with a fixed cost, which then implies that all users would be given resources for the same amount of budget.

Asset fairness can, however, give rise to undesirable outcomes. In our canonical example, asset fairness will give the first user 60 tasks and the second user 120 tasks. The first user will use $\langle 60, 180 \rangle$ resources, while the second will use $\langle 120, 120 \rangle$ (see Figure 2). While each user uses a total of 240 resources, the second user has got less than half (150) of both resources. We believe that this could be considered unfair, making the second user's owner inclined to buy a separate cluster of dimension $\langle 150, 150 \rangle$, using it all by itself.

4.1   Dominating Resource Fairness

The last example highlights that users might not care about the sum of their allocated resources, but instead care about their number of tasks. In other words, users care about the resource that they relatively demand most of, since that is the resource they will be allocated most of. We define a user's *dominating resource* to be the resource that it *percentage-wise* demands most of, e.g. with a total 10 CPUs and 40GB memory, a user that de-

mands 1 CPU and 2 GB memory per task has CPU as its dominating resource, as that is the resource that dominates its relative demand.

It is, thus, natural to attempt to give all users equal amounts of their dominating resource. We call this, *dominating resource fairness*. This is achieved by assigning to each user i a *dominating share* x_i, which is i's share of its dominating resource, i.e. $x_i = \max_j\{s_{ij}\}$, where s_{ij} is user i's fractional share of resource j.[6] The scheduler allocates resources to the user with least x_i.[7]

If we consider our canonical example, dominating resource fairness will allocate the two users 50 and 150 tasks, respectively. Thus, the resource usage of the two users becomes, $\langle 50, 150 \rangle$, and $\langle 150, 150 \rangle$, respectively (see Figure 2). Hence, both users get half of their dominating resource, i.e. $x_i = 0.5$.

Properties. Dominating resource fairness satisfies a *share guarantee* property that we consider crucial: each user receives $\frac{1}{n}$ fraction of at least one of its resources. Informally, share guarantee can be interpreted as:

> *Each user will get at least as much resources as it would get by running its own cluster.*

Dominating resource fairness satisfies share guarantee, modulo fractional task allocations. To see that, note that a user which has $\frac{1}{n}$ of one of its resources surely has $\frac{1}{n}$ of its dominating resource, and vice versa. Share guarantee, thus, ensures that a user gets $\frac{1}{n}$ of its dominant resource. If every user is given exactly $\frac{1}{n}$ of all resources, every user will be able to get $\frac{1}{n}$ of their dominating resource.

We compare dominating resource fairness with previous work of fairness in networking. Dominating resource fairness is an adaptation of *max-min fairness* to cluster environments [20, pg 448]. An allocation according to dominating resource fairness is max-min fair in the sense that any increase in a user p's tasks will be at the expense of a decrease in another user q's task, where p already had more of its dominating resource than q had of its dominating resource. To see this, note that dominating resource fairness is an approximation of *progressive filling* [20, pg 450], in which all users' usage of their dominating resource is increased at the same rate, while proportionally increasing its other resources, until some resource is exhausted, in which case those users' allocation is finalized, and this is repeated recursively for remaining users.

Another way to understand dominating resource fairness is through Jain's Fairness Index (JFI) [34]:

6Asset fairness can be achieved by instead assigning $x_i = \sum_j s_{ij}$.

7The Nexus schedulers actually return a list of users sorted in increasing order by x_i. The reason for the list is that Nexus' scheduler attempts to always schedule the first user in the list for which there are available resources.

$\frac{(\sum_i x_i)^2}{n \sum_i {x_i}^2}$, where n is the number of users in the cluster, and x_i is their resource share. JFI was originally intended for sharing a single resource, but we let x_i be the dominating share of user i. Thus, a maximum JFI of 1.0 corresponds to scheduling resources according to dominating resource fairness, as it implies that every user has the same amount of their dominating resource. Thus, dominating resource fairness can be seen as a greedy algorithm for maximizing JFI.

Share guarantee only ensures that each user gets a $\frac{1}{n}$ share. After each user has got $\frac{1}{n}$ fraction of its dominating resource, there might be room for more tasks. This is especially the case with heterogeneous demand vectors, e.g. two users with demands $\langle 1, 9 \rangle$, $\langle 9, 1 \rangle$ can both be allocated 90% of their dominating resource. We next explain different ways to allocate the resources that remain after the share guarantee has been satisfied.

Maximizing Utilization. It is possible to use dominating resource fairness until each user has received $\frac{1}{n}$ of it its dominating resource, and thereafter use another scheduler that opts for maximum utilization of the resources. We found one scheduler particularly efficient for maximizing utilization. It is a greedy scheduler whose goal is to schedule the user that will give the most even utilization of all resources. This is done by calculating for each user, u, what the utilization of every resource would be if u would be allocated another task according to its demand vector. It then calculates the the variance of utilization across the resources, and returns the list of users sorted in increasing order of variance. We compared this to a linear program that scheduled users for optimal utilization, and found that it performs close to the optimum. Nevertheless, scheduling for utilization might not be desirable as we show next.

Gaming the Scheduler. A scheduler that optimizes for maximum utilization can always be gamed by a user that shapes its demand vector to be identical to the remaining resources. For example, if the remaining resources are $\langle 6, 18, 24 \rangle$, a user can make its demand vector $\langle 1, 3, 4 \rangle$, ensuring that an allocation for it can perfectly use all remaining resources.

Dominating resource fairness is, however, harder to game. If a user demands more of some resource that it does not need, it is likely to get hurt in scheduling. If the surplus demand changes its dominating resource, its dominating share will be higher as soon as it is allocated, penalizing it during scheduling, even though it cannot use the surplus resource. If the extra demand does not change its dominating resource, it is still possible that it gets hurt in scheduling because its surplus demand cannot be satisfied. Gaming can, however, be possible in certain specific scenarios. For instance, a user, u, might

get ahead by requesting a surplus resource, which when allocated makes other users' demand vectors insatiable, allowing u to get ahead of them during scheduling.

4.2 Weighted Fair Sharing

We have so far assumed that resources are to be shared equally among the users. In practise, it is desirable to be able to weight the sharing. The motivation for this is that different organizational entities might have contributed different amount of resources to the cluster or that there is a payment scheme in which different cluster owners pay for parts of the cluster. In either case, sharing could be weighted on a users basis, as well as on a resource basis. For example, some organization might have contributed with more memory resources, and should therefore have a higher share of the memory resources.

Weighted fair sharing is implemented in Nexus similarly to Lottery Scheduling [43]. Every user i has a vector of weights of positive real numbers w_{ij}, where $1 \leq j \leq r$ for r resources. The weight w_{ij} expresses that user i's fair proportion of resource j is $\frac{w_{ij}}{\sum_k w_{kj}}$.

The definition of a dominating share for user i now changes to $x_i = \max_j \{ \frac{s_{ij}}{w_{ij}} \}$. Thus, share guarantee says that a user i will get at least $\frac{w_{ij}}{\sum_k w_{kj}}$ fraction of its dominating resource j. Note that dominating resource scheduling now uses the new definition of a dominating share, x_i.

If $w_{ij} = 1.0$ for all users and resources, weighted fair sharing reduces to equal fair sharing. Another use case is to assign user-wide weights. For example, if $w_{ij} = k$ for all j, while $w_{kj} = 1.0$ for all $k \neq i$ and all j, then user i will have twice the proportion of all resources compared to every other user in the system. We have found that the latter use often suffices, as one would like to weight on a per-user level, and not on a per-resource level.

4.3 Resource Revocation

It is the job of the scheduler to decide which users' tasks should be revoked, and on what machines. This is complicated by that the granularity of tasks can be a value below the specified minimum offer size, `min_offer`. If care is not taken, the revoked tasks might not make enough room for a new task that is of dimension `min_offer`. In the worst case, a user above its fair share might be running very small tasks on every machine, such that even if all its tasks were revoked, there might not be room for a new task.

Nexus' avoids the above scenario by assigning to each slave machine a fairness score indicating how fair the allocation would be if `min_offer` resources would be freed on that machine through killing. It thereafter selects the most fair machine for revokation, and repeats the process to free more resources. When a user filter exists, Nexus assigns a minimum score to those machines, ensuring that they will be selected for revokation last.

The fairness score is based on JFI and is calculated for each slave m by sorting the users running on m in decreasing order of their dominating share. Nexus then traverses the list of users and marks their tasks for revocation until room has been made for a `min_offer`. Thereafter, a cluster-wide JFI is calculated, discounting the marked tasks against users' shares, and assigns the score to m.

Nexus' will pick the slave with highest JFI for resource revokation. It will send notification to the marked users and give them a grace period to kill tasks on that machine. If only a single user's tasks are marked for revocation, Nexus offers that user the freedom to kill that amount of its tasks on any machine it is running on. After the grace period expires, Nexus selects the marked tasks and kills them.

The above scheme attempts to achieve maximum fairness while minimizing resource revokation. Note that in the typical case where the user u that is most above its fair share is running more than `min_offer` on a single machine, the above scheme will always always pick u for resource revocation. This is because the slave that u is running on will receive the highest JFI score.

5 Implementation

We have implemented Nexus in approximately 5,000 lines of C/C++. The implementation, which was designed for any 32-bit and 64-bit POSIX-compliant operating system, has been tested on both Linux and Solaris.

To reduce the complexity of our implementation, we use a library which provides an actor-based programming model that uses efficient asynchronous techniques for I/O [9]. We use the library and its built-in message serialization to perform all of the communication between the different components of Nexus.

The current implementation assumes the use of socialized utilities, such as HDFS and MapOutputServer, and only supports resource offers for CPU cores and memory (i.e., it does not provide storage and I/O isolation.)

An important benefit of using C/C++ to implement Nexus has been our ability to easily interoperate with other languages. In particular, we use the Simplified Wrapper and Interface Generator (SWIG) to generate interfaces and bindings in Ruby, Python, and Java. As it turns out, none of our example frameworks are written against the C/C++ API.

We use the rest of this section to accomplish two things: first, we elaborate on how we provide performance isolation between framework executors and second we discuss in detail how frameworks can be implemented to run on Nexus.

Nexus easily interoperates with other languages, in particular, we use the Simplified Wrapper and Interface Generator (SWIG) to generate interfaces and bindings in Ruby, Python, and Java.

5.1 Framework Isolation

Recall from Section 3.3, that a good isolation mechanism for Nexus should (a) have low overheads for executor startup and task execution and (b) have the ability to let Nexus change resource allocations dynamically.

Given those constraints, we present possible mechanisms below:

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

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

Cpusets, Containers, Zones, etc. Modern operating systems are begining to provid mechanisms to isolate entire process trees. For example, Linux supports cpusets and cgroups for CPU isolation [16], and Linux containers [15] are aimed to provide more comprehensive isolation. These mechanisms tend to be very lightweight and are dynmically configurable while a process is running (similar to ulimit and setrlimit).

Solaris provides a relatively advanced set of mechanisms for resource isolation [14], which allows, for example, one to set cumulative limits on CPU share, resident set size, and OS objects such as threads, on a process tree. A nice property of the Solaris mechanisms is that you can configure, at least for some resources, the ability to let idle resources get used by processes that have reached their limits.

For our current implementation, we choose to use the Solaris resource management mechanisms. This allows Nexus to isolate CPU usage and memory usage per executor process tree.

We use the Solaris resource management mechanisms [14] to enable Nexus to isolate CPU usage and memory usage per executor process tree. Solaris provides a relatively advanced set of mechanisms for resource isolation which allows, for example, one to set cumulative limits on CPU share, resident set size, and OS objects such as threads, on a process tree. A nice property of the Solaris mechanisms is that you can enable, at least for some resources, idle resources to be used by processes that have reached their limits.

One contributing factor in our decision to use Solaris was our desire to run large-scale tests on Amazon EC2, since Linux containers would have required patching the Linux kernel, which is not allowed by EC2.

As operating system isolation mechanisms improve, they should only strengthen the performance isolation guarantees that Nexus can provide. We explore how well the performance isolation mechanisms worked in Section 6.

5.2 Frameworks

We have ported Hadoop and the MPICH [21] implementation of MPI to run on Nexus. Neither of these ports required changing the existing interfaces, so existing run unmodified. In addition, we built a new framework from scratch for writing machine learning applications as well as a framework that elastically scales Apache [2] web servers. We describe the implementation details of each framework below.

5.2.1 Hadoop

Porting Hadoop to run on Nexus required minimal modification of Hadoop internals because Hadoop's concepts such as tasks map cleanly onto Nexus abstractions. We used Hadoop's existing master, the JobTracker, as our Nexus scheduler, and we used Hadoop's slave daemon, the TaskTracker, as our executor. We needed to implement two major changes:

- Factoring out the map output server from the Task-Tracker. Normally, Hadoop has each TaskTracker serve local map outputs to reduces, but if the Task-Tracker runs as an executor, it may occasionally be killed. We made the map output server a separate daemon shared across Hadoop instances.

- Changing scheduling to use Nexus's resource offer mechanism, as we describe below.

In normal operation, Hadoop schedules tasks on its slaves in response to heartbeat messages that slaves send every 3 seconds to report their status. Each slave has a number of "map slots" and "reduce slots" in which it can run map and reduce tasks respectively. In each heartbeat, the slave reports its total number of slots of each type and the number of slots it has free. The JobTracker gives the slave new tasks if there are free slots, preferentially choosing tasks with data local to the slave.

The heartbeat mechanism provides a good opportunity to add support for Nexus because it presents a simple interface to "pull" tasks for a particular slave from the JobTracker. In our port, we take advantage of this interface to maximize the amount of Hadoop code we reuse. Whenever Nexus makes a resource offer on a particular slave to our Nexus scheduler (which is located in the Job-Tracker), we check whether there are any runnable map

or reduce tasks. If there are, we accept the offer (modulo a detail on data locality which we get to later), and we send a message to the slave that the offer was on to have it increase its number of map or reduce slots. Whether we choose to add a map slot or a reduce slot depends on which tasks are available; if both are available, we keep a configurable ratio of map and reduce slots on each slave (by default 1:1). After the slave receives the Nexus-level task and increases its slot count, the next time it sends a heartbeat to the master, Hadoop's standard scheduling algorithm is invoked to pick a Hadoop-level map or reduce task to run. Then, whenever the slave sends a heartbeat to the master saying that a task is done, our shim layer also decrements the slave's slot count and reports the task as finished to Nexus. Finally, if we create a slot but the slave is not assigned a task within two heartbeats (perhaps because jobs finished), it removes the slot and also sends a task-done report to Nexus.

A final detail deals with data locality. When considering which slaves to launch map tasks on, we take a policy of waiting up to t seconds to find a slave that contains local data for one of the yet-unlaunched map tasks in the system. If no such slave is found, we accept a non-local task. The wait time is reset only if we ever get a local task again. This simple mechanism, called *delay scheduling*, is explained in our previous work [45], where it was found to provide near-perfect data locality in a fair scheduler for Hadoop.

In total, our changes to Hadoop were 1100 lines of code, of which 1000 are new files adding the shim layer and the map output server.

5.2.2 MPICH

MPI is a language-independent message-passing API used to implement parallel programs that run primarily on supercomputers and clusters. MPI works by launching daemons on machines that will participate in the computation. A user then executes the standard `mpiexec` to run a program on each of the machines running daemons.

Rather than make invasive changes in the implementation of MPICH, we created a framework "wrapper" around `mpiexec` that registers with the Nexus master, launches a local MPI "master" daemon, accepts resource offers large enough to run instances of the specified program, and then has the executor launch an MPI daemon that connects back to the "master" daemon. Once the wrapper has acquired enough resources to run the program, it invokes `mpiexec` which uses the local MPI daemon to begin the distributed MPI computation. Note that because MPI uses the MPI daemon's to launch more processes, nothing extra needs to be done to ensure subsequently launched processes on the slaves will be properly isolated.

The simplicity of this approach is a direct consequence of how a majority of MPI jobs are executed. Since few MPI jobs actually fork and launch computation dynamically, we choose not to provide any mechanism for doing so, and we implemented a very simple scheduling policy – accept any offered resources that are large enough to launch the program.

Currently, the wrapper launches MPI jobs conservatively. That is, it never attempts to use more than its fair share of the resources to avoid resource revocation. Mechanisms such as Berkeley Lab Checkpoint/Restart [29] could be used to support revocation.

The entire wrapper was written using our Python interface and bindings for Nexus in about 200 lines of code.

5.2.3 Spark

Nexus 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 machine learning researchers at our institution ran on Hadoop and found performed poorly – *iterative* jobs, where a data set is reused across a number of iterations. We built a framework called Spark optimized for these workloads.

Example Job A simple example of an iterative algorithm used in machine learning is logistic regression [10]. This algorithm seeks to find the line that best separates two clusters of labeled data points. The algorithm starts with a random separating line, w. Then, on each iteration, it computes the gradient of an objective function that measures how well the line is separating the points, and shifts w in the direction of this gradient. The gradient computation amounts to evaluating a function $f(x, w)$ on each data point x and summing these 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. Dryad could express the whole job as a data flow DAG as shown in Figure 3 a). However, it still has the problem that each iteration must reload the data file from disk; there is no way in Dryad to ask one vertex to evaluate $f(x, w)$ for multiple values of w, as this requires cyclic data flow. Spark's execution is shown in Figure 3 b). Spark uses the long-lived nature of Nexus executors to cache a slice of the data set in memory at each executor, and then run multiple iterations on this same cached data.

Spark Details Spark is implemented in Scala [13], a high-level object-oriented/functional programming language that runs over the JVM. Users write jobs in Scala, and invoke Spark through a language-integrated syntax
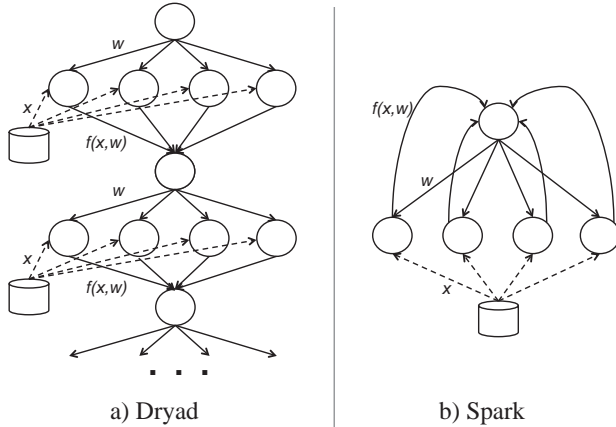
a) Dryad   b) Spark

Figure 3: Comparison of execution of the logistic regression job in Dryad and Spark. Solid lines represent data flow within the cluster computing framework. Dashed lines represent reads from a distributed file system. In Spark, the same worker processes are reused across iterations and reads only happen once.

```
val data = spark.hdfsTextFile(...).map(readPoint _).cache()
var w = Vector.random(D)
for (i <- 1 to ITERATIONS) {
  val gradient = spark.accumulator(Vector.zeros(D))
  val W = w
  for (p <- data) {
    val scale = (1/(1+Math.exp(-p.y*(W dot p.x)))-1)*p.y
    gradient += scale * p.x
  }
  w -= gradient.value
}
println("Result: " + w)
```

Figure 4: Implementation of logistic regression in Spark. The body of the inner loop (highlighted) runs in parallel on Nexus.

similar to DryadLINQ [44]: users write a for loop over a special "distributed data set" object, and the body of the for loop is passed as closure to run as a Nexus task on the appropriate node for each slice of the data.[8] Spark allows users to ask for a parallel data set to be read from the Hadoop Distributed File System, transformed, and cached in memory at executors. It then schedules tasks to run on executors that already have the appropriate data cached, using a delay scheduling algorithm similar to our Hadoop port. Figure 4 shows how the logistic regression algorithm is expressed in Spark.

Spark is implemented in 800 lines of code, but can outperform a Hadoop implementation of logistic regression by 8x, as shown in Section 6.3. Due to lack of space, we limit our discussion of Spark in this paper and refer the reader to `http://www.cs.berkeley.edu/~matei/spark` for more details.

This for-loop syntax integration has previously been used by a Scala API for Hadoop [28].

	Average time (s)
MPI	50.85
MPI on Nexus	51.79

Table 1: Overhead of running the MPI LINPACK benchmark on Nexus

	Average time (s)
Hadoop	159.87
Hadoop on Nexus	166.19

Table 2: Overhead of running the WordCount Hadoop workload on Nexus

5.2.4 Elastic Web Farm

To explore how to implement a more interactive framework, we built an elastic web farm. Sharing resources between web servers and other frameworks is a natural desire; when the web load is low the machines can be used for launching tasks from other frameworks and as the web load increases more instances of the web server can be launched.

We used the load balancer haproxy [7] as our front-end and Apache as our back-end, however any web server would have been sufficient. Similar to the MPI wrapper framework, we created a framework which wraps the launching and killing of web servers on nodes. The wrapper queries the front-end for load statistics, and uses that to decide whether or not to launch, or teardown, servers. The wrapper's only scheduling constraint is that it only launches one Apache instance per machine, and it uses filters to assist the Nexus master in this respect.

One unfortunate aspect of using haproxy as the front-end was that it does not provide good mechanisms to do hot-swapping of its configuration files. Instead, haproxy needs to be restarted for each reconfigure, which can often cause certain connections to be terminated.

Like the MPI wrapper, this wrapper was written using our Python interface and bindings for Nexus and is about 250 lines of Python.

6 Evaluation

We evaluated Nexus by performing a series of experiements using Amazon's EC2.

6.1 Overhead

To measure the overhead Nexus imposes on frameworks we ran two benchmarks using Hadoop and MPI. These experiments were performed on EC2 using 50 nodes, each with 2 CPU cores and 6.5 GB of memory. We used the WordCount workload for Hadoop and we used the High-Performance LINPACK [18] benchmark for MPI. Tables 2 and 1 show the average running time across three runs of Hadoop and MPI both with and without

Nexus, respectively. As these results show, the overhead of Nexus is statistically insignificant, when it exists at all.

6.2 Dynamic Resource Sharing

We wanted to evaluate how well Nexus performs with multiple different frameworks running simultaneously. This experiment was performed on EC2 with 70 nodes, each with 2 CPU cores and 6.5 GB of memory.

For this experiement we decided to run Hadoop, Spark, and MPI and allocate resources according to dominating resource fairness (where each framework has an equal weight). Similar to above, we used the WordCount workload for Hadoop and the LINPACK benchmark for MPI. The Spark framework was running a job performing logistic regression. The logistic regression job performs a series of iterations, in between each of which it does not use any resources in the cluster.

The results of this experiement are shown in 5. In this experiment, CPU was the bottleneck resource in the cluster. In this experiement we first launched the Spark framework, which was offered the entire cluster and accepted it. After about 15 seconds we launched Hadoop, which received its fair share (50% of the CPUS) within a few seconds. After about 50 seconds we launched MPI. Within a few seconds MPI was offered its fair share of the CPUS (approximately $\frac{1}{3}$), which it accepted and began running its tasks. At roughly this same point the Spark job completed its first iteration and released all of its resources, which Hadoop utilized almost instantly. Each time the Spark job began a new iteration, it was offered roughly $\frac{1}{3}$ of the CPU resources within a few tens of seconds. Finally, around 330 seconds the Hadoop job completed, and the Spark framework was offered, and accepted, the extra resources. Because of the nature of the MPI job, it was not able to benefit from any of the extra resources.

This experiment helps to illuminate how the share of the cluster can be shared dynamically between different frameworks. In this case, we show how quickly frameworks like Spark and Hadoop can utilize extra resources when they aren't being used while the MPI framework can continue executing below its fair share unharmed.

6.3 Benefit of Specialized Frameworks

In this experiment, we evaluate whether the specialized Spark framework described in Section 5, which is optimized for iterative jobs, provides a benefit over the general-purpose Hadoop framework. We use a logistic regression job implemented on top of Hadoop by machine learning researchers in our department to evaluate Spark. We implemented a second version of the job in Spark ourselves. We ran the job on a 11 GB data file on 43 EC2 machines with 8 cores each. The data file contained each point as plain text, which according to
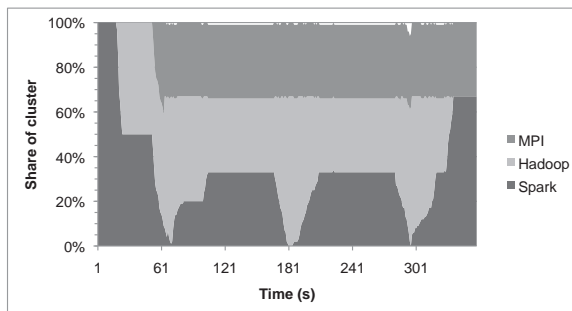

Figure 5: Timeline showing the shares of the cluster given to MPI, Hadoop and Spark in the dynamic resource sharing experiment.
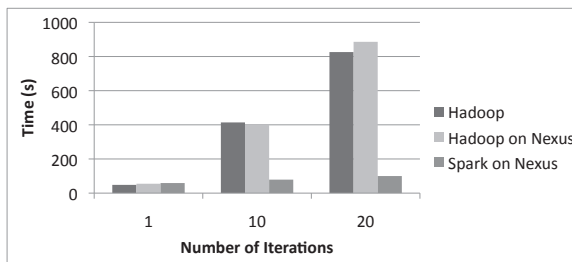

Figure 6: Comparing the running time of logistic regression when implemented using Spark vs. using Hadoop.

the machine learning researchers that wrote the job is a standard format in which machine learning data sets are provided. We varied the number of iterations of logistic regression from 1 to 20, and ran each job on a standalone Hadoop cluster, a Hadoop framework running on Nexus, and a Spark framework running on Nexus. Figure 6 shows the results, which averages values from 3 runs.

We see that each iteration of the job takes 41s on average on both Hadoop and Hadoop on Nexus. Some variation happens because the jobs are so short, and Hadoop's heartbeat intervals are 3 seconds so any variation in scheduling opportunities might result in a 3 second delay. In contrast, Spark takes 60s to run the first iteration (because it uses slower text-parsing routines), but each subsequent iteration is on average 2 seconds. This is because in the logistic regression job, the function $f(x, w)$ evaluated at each iteration is so inexpensive that the cost to read the input data from HDFS and parse the text into floating point numbers dominates the computation. Hadoop must incur this cost on each iteration, while Spark reuses blocks of parsed data cached in memory and only incurs the cost once. This leads to a 8.5x speedup on 20 iterations.

6.4 Resource Offers and Data Locality

In this experiment, we wanted to verify whether the resource offer mechanism in Nexus allows frameworks
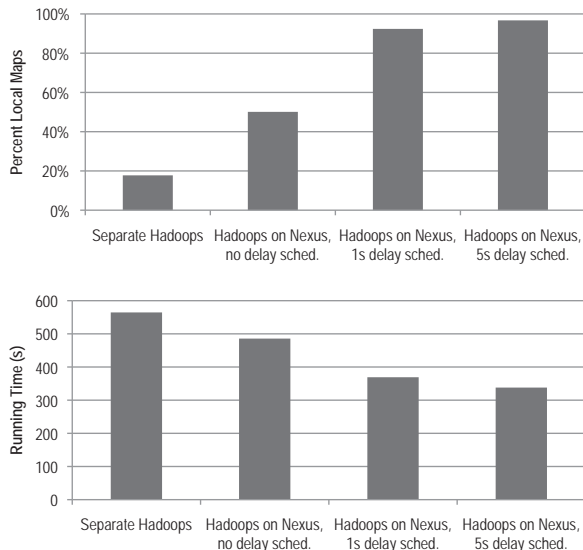
Figure 7: Results from data locality experiment. The top graph shows the percentage of local map tasks in each setting, while the bottom graph shows the average job running time in each setting.

to achieve control over their tasks' placement, and in particular high data locality. We ran 16 instances of Hadoop on a 93-node EC2 cluster with 4 cores per node. All of the instances were running a map-only filter job that read a 100 GB file striped throughout the cluster on a shared HDFS file system and outputted 1% of the records. We tested four scenarios: Separate Hadoop MapReduce clusters of 5-6 nodes each, to emulate organizations that use coarse-grained resource managers, and all instances on Nexus using either no delay scheduling, 1s delay scheduling or 5s delay scheduling.

The results are shown in Figure 7, which averages numbers from 3 runs of each scenario. We see that data locality is very low (18%) on the separate clusters. Running the Hadoop frameworks on Nexus improves locality even with no delay scheduling because each node is running tasks from 4 random Hadoop instances, so each Hadoop instance has tasks on more than 5-6 nodes. Adding delay scheduling brings locality above 90%, even with a 1-second delay. Five-second delay scheduling achieves 95% locality, which is similar to Hadoop running alone (for comparison, a single instance of Hadoop running on the whole cluster achieves 93% locality for the job we used). Finally, performance improves with locality, with the 5s delay scenario running jobs 1.7x faster than separate Hadoop instances.

6.5 Elastic Web Farm

Finally, we wanted to evaluate how well interactive frameworks can co-exist with other frameworks. This
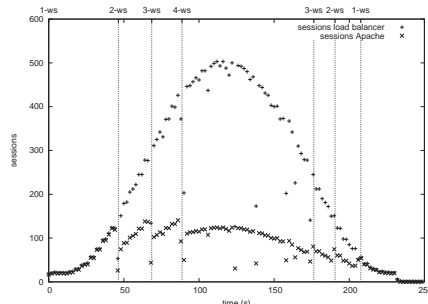

Figure 8: The average session load on the load balancer over time, as well as the average number of sessions across web servers.

included investigating how quickly these frameworks would be offered resources and how well the isolation mechanism we choose to use works. To do this we ran two experiments, one where the only framework running in the cluster was the elastic web farm and the other where we run both the elastic web farm and a "hog" framework. The hog framework is designed to consume available resources by launching tasks that use one or more threads to read and write a specified amount of data for a specified period of time. In this case, the hog framework launched tasks that created ten threads that each spent approximately two minutes continuously cycling through 64 MB of data. We ran our experiment using 4 EC2 nodes each with 8 CPU cores and 6.5 GB of memory. Note that to apply pressure to the isolation mechanism each each hog task would request only 1 CPU core and 512 MB of memory.

In both experiments we generated workload using HTTPerf [36]. Figure 8 shows the measured average number of sessions on the load balancer, as well as the average measured sessions across web servers when both the hog framework and the elastic web framework were running simultaneously. The vertical lines of the figure indicate when the number of web servers is alterned. The effect of this is seen on the average web server load, which stays low relative to the generated load.

Running the same workload without the hog framework showed no statistically significant differences. As mentioned in Section 5.2.4, connection errors occured during the hot-swapping of configuration files, however, both of the experiements suffered equally from this issue.

7 Discussion

7.1 Philosophy

Nexus shares many goals in common with operating systems. Nexus provides (1) a clean abstraction of tasks to frameworks, (2) performance and fault isolation between frameworks, and (3) fair multiplexing of CPU, memory,

and other resources available on slave nodes. These are some of the goals of traditional operating systems.

Our approach shares much in common with the exokernel [25], microkernels [17], and hypervisors [31]. Like an exokernel, Nexus aims to give frameworks as much control over their execution as possible via as low level abstraction as possible to achieve generality. Like a microkernel or hypervisor, Nexus is a stable, minimal core that provides performance and fault isolation to frameworks sharing a cluster.

7.2 Limitations and Future Work

I/O isolation   As mentioned in sections 3.3 and 5.1, while a primary goal of Nexus is resource isolation, in order to limit the scope of our research we chose to leverage existing work to achieve this goal. Similarly, in future work, we aim to leverage existing mechanisms to provide isolation and fair sharing of disk I/O and network bandwidth. Sharing network bandwidth is particularly challenging, as the use of existing quality of service mechanisms in the operating systems on nexus slaves is not sufficient because fair network bandwidth sharing must also be implemented at the aggregation switch level.

Fairness of individual resources   Nexus does not provide any fairness guarantees on the level of individual resources. Therefore, frameworks should not make assumptions about the provisioning of individual resources. In particular, *locality deadlocks* may occur if two different frameworks each accept a slot offer and thereafter wait for the other framework's slot to become free. Moreover, a framework that is below its fair share can accept a resource offer and leave those resources permanently idle. Nexus, will not guarantee that those resources are revoked. It might, however, revoke resources belonging to a framework that is above its fair share. However, framework schedulers can implement a timeout after which they accept any resource offer to ameliorate the problem.

Socialized services   Some cluster services, similar to those provided by Amazon Web Services (SQS, S3, etc.), might be useful to many frameworks and might be designed to run distributed throughout the cluster. A concrete example we have already encountered is the Hadoop Distributed File System, which Hadoop uses for MapReduce input and output. As future work we intend to allow Nexus to support *socialized services—*, which are services that require resources but are accessible by all frameworks. Such services are necessary in order to leverage sharing data sets and also to amortize the overhead of running such services. Much research remains to be done in order to moderate fair the access to socialized resources.

Utility libraries   Similar to exokernel, in addition to building and porting standalone frameworks (such as Hadoop and Spark), we also intend to provide *utility libraries*, i.e. implementations of common distributed system services. Our goal is to make creating new frameworks as painless as possible by factoring out as many common functionalities as possible. For example, existing libraries that encapsulate communication or message passing models for executors to use for communicating with each other (much as we used existing MPI implementation to create a new framework). Another example might be entire implementations of common framework schedulers, or building blocks for constructing such schedulers (e.g. a library that provides logic to do speculative execution [46]).

8   Related Work

8.1   Cluster Computing Frameworks

We have discussed and described a number of existing cluster computing frameworks. These include MapReduce [23], as well as Sawzall [39] and Pig [38], which were built on top of MapReduce. Also Dryad [32] and Clustera [24], which provide more general execution models. Nexus does not compete with any single cluster framework, but instead aims to provide a common substrate that a wide array of existing and future frameworks can build upon.

8.2   Infrastructure as a Service

Cloud infrastructures such as Amazon EC2 [1] and Eucalyptus [37] allow for sharing between users by allowing virtual machines in a shared cloud to be rented by the hour. In such an environment, it would be possible for separate frameworks to run concurrently on the same physical cluster by creating separate virtual clusters (i.e., EC2 allocations). However, VMs can take minutes to start and sharing data between separate virtual clusters is difficult to accomplish and can result in poor data locality. Nexus provides abstractions (i.e., tasks and slots) that eradicate the need for many applications to use heavyweight VMs. In addition, Nexus allows frameworks to select where to run tasks via the resource offer mechanism, allowing multiple frameworks to share data while achieving good data locality.

8.3   Cluster Scheduling Systems

Many resource managers and job schedulers for the cluster exist, such as Torque [41], Portable Batch System (PBS) [30], Sun's GridEngine [26], and Cobalt [19]. Nexus differs from these existing cluster resource managers and schedulers in the fine-grained nature of tasks and also in its two level scheduling of tasks in which frameworks play an interactive role via resource offers.

In contrast, most of these systems give each job a fixed block of machines at once, rather than letting a job's allocation scale up and down in a fine-grained manner, and do not provide frameworks much control over data locality.

Some systems, such as Condor and Clustera [42, 24] 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 locality needs (e.g. by listing its input files). Similarly, Condor uses the "ClassAd" mechanism [40] to match node properties to job needs. In these two systems, and more generally any system that provides a language for jobs to express preferences about resources they want to use, there will be job needs that cannot be expressed in the language. In contrast, The two level scheduling and resource offer architecture in Nexus gives jobs arbitrary flexibility in deciding where and when they run tasks.

8.4 Virtual Machines

While it is a goal of Nexus to provide resource isolation between frameworks, Nexus itself does not enforce resource isolation between tasks on slave nodes nodes. Rather, Nexus leverages existing mechanisms for resource isolation (i.e. see section 3.3 for a further discussion of this), and manages isolation in order to provide the higher level goal of .

8.5 Scheduling

Quincy [33] is a scheduler for Dryad. It uses centralized one level scheduling, task killing (similar to Nexus resource revocation), supports fair sharing at the level of fixed sized tasks, and accounts for data locality. In contrast, Nexus uses a two level scheduler with resource offers, and support dynamic multi resource task sizes.

9   Conclusion

We have described Nexus, a common substrate for cluster computing that provides isolation and efficient resource multiplexing of fine grained tasks across frameworks running on the same cluster. Nexus provides each framework freedom to implement its own programming model and dynamically schedule the execution of its jobs.

Nexus uses an interactive two-level scheduling architecture that enables fine-grained, dynamic sharing between frameworks via tasks and resource offers. By remaining pluggable, Nexus provides organizations flexibility in choosing a first level resource scheduling policy, but provides a default module implementing a novel weighted fair scheduler generalized for multiple resources.

We have presented an implementation of Nexus and of a varied set of frameworks such as MPI and Hadoop, that can run over it. We have presented experiments demonstrating frameworks sharing a single cluster, with high cluster utilization, good data locality and fair sharing.

To validate our hypothesis that specialized frameworks can provide value over general ones, we built a new machine learning framework on top of Nexus which can outperform Hadoop by 8 times on iterative workloads.

References

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.
[2] Apache Web Server Homepage. `http://httpd.apache.org/`.
[3] Hadoop. `http://lucene.apache.org/hadoop`.
[4] Hadoop and hive at facebook. `http://www.slideshare.net/dzhou/facebook-hadoop-data-applications`.
[5] Hadoop capacity scheduler. `http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html`.
[6] Hadoop PoweredBy Wiki. `http://wiki.apache.org/hadoop/PoweredBy`.
[7] HAProxy Homepage. `http://haproxy.1wt.eu/`.
[8] Hive – A Petabyte Scale Data Warehouse using Hadoop. `http://www.facebook.com/note.php?note_id=89508453919#/note.php?note_id=%89508453919`.
[9] LibProcess Homepage. `http://www.eecs.berkeley.edu/~benh/libprocess`.
[10] Logistic Regression Wikipedia Page. `http://en.wikipedia.org/wiki/Logistic_regression`.
[11] NSF Cluster Exploratory. `http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm`.
[12] Personal communication with Joydeep Sen Sarma from the Facebook data infrastructure team.
[13] Scala. `http://www.scala-lang.org`.
[14] Solaris Resource Management. `http://docs.sun.com/app/docs/doc/817-1592`.
[15] Linux containers (lxc) overview document. `http://lxc.sourceforge.net/lxc.html`, October 2009.
[16] Linux kernel cpusets documentation. `http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt`, October 2009.
[17] 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.
[18] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
[19] 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, Nov. 2006.
[20] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.
[21] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik,

P. Lemarinier, and F. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[22] L. Cherkasova and R. Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

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

[24] 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.

[25] 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.

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

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. pages 29–43, 2003.

[28] D. Hall. A Scalable Language, and a Scalable Framework. http://scala-blogs.org/2008/09/scalable-language-and-scalable.html.

[29] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *J. Phys.: Conf. Ser.*, 46(1):494+, 2006.

[30] R. L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

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

[32] 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.

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

[34] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODC '09*, pages 6–6, New York, NY, USA, 2009. ACM.

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

[37] 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.

[38] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[39] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

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

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

[42] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation - Practice and Experience*, 17(2-4):323–356, 2005.

[43] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceed-*

ings of the 1st USENIX conference on Operating Systems Design and Implementation, pages 1+, Berkeley, CA, USA, 1994. USENIX Association.

[44] 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.

[45] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, UC Berkeley.

[46] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments.

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