Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling

Matei Zaharia
University of California, Berkeley
matei@berkeley.edu

Dhruba Borthakur
Facebook Inc
dhruba@facebook.com

Joydeep Sen Sarma
Facebook Inc
jssarma@facebook.com

Khaled Elmeleegy
Yahoo! Research
khaled@yahoo-inc.com

Scott Shenker
University of California, Berkeley
shenker@cs.berkeley.edu

Ion Stoica
University of California, Berkeley
istoica@cs.berkeley.edu

Abstract

As organizations start to use data-intensive cluster computing systems like Hadoop and Dryad for more applications, there is a growing need to share clusters between users. However, there is a conflict between *fairness* in scheduling and *data locality* (placing tasks on nodes that contain their input data). We illustrate this problem through our experience designing a fair scheduler for a 600-node Hadoop cluster at Facebook. To address the conflict between locality and fairness, we propose a simple algorithm called *delay scheduling*: when the job that should be scheduled next according to fairness cannot launch a local task, it waits for a small amount of time, letting other jobs launch tasks instead. We find that delay scheduling achieves nearly optimal data locality in a variety of workloads and can increase throughput by up to 2x while preserving fairness. In addition, the simplicity of delay scheduling makes it applicable under a wide variety of scheduling policies beyond fair sharing.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling.

General Terms Algorithms, Performance, Design.

1. Introduction

Cluster computing systems like MapReduce [18] and Dryad [23] were originally optimized for batch jobs such as web indexing. However, another use case has recently emerged: sharing a cluster between multiple users, which run a mix of long batch jobs and short interactive queries over a common data set. Sharing enables statistical multiplexing, leading to lower costs over building separate clusters for each group. Sharing also leads to data consolidation (colocation of disparate data sets), avoiding costly replication of data across clusters and letting users run queries across disjoint data sets efficiently. In this paper, we explore the problem of sharing a cluster between users while preserving the efficiency of systems like MapReduce – specifically, preserving *data locality*, the placement of computation near its input data. Locality is crucial for performance in large clusters because network bisection bandwidth becomes a bottleneck [18].

Our work was originally motivated by the MapReduce workload at Facebook. Event logs from Facebook's website are imported into a 600-node Hadoop [2] data warehouse, where they are used for a variety of applications, including business intelligence, spam detection, and ad optimization. The warehouse stores 2 PB of data, and grows by 15 TB per day. 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 through an SQL interface to Hadoop called Hive [3]. The system runs 7500 MapReduce jobs per day and is used by 200 analysts and engineers.

As Facebook began building its data warehouse, it found the data consolidation provided by a shared cluster highly beneficial. However, when enough groups began using Hadoop, job response times started to suffer due to Hadoop's FIFO scheduler. This was unacceptable for production jobs and made interactive queries impossible. To address this problem, we have designed the Hadoop Fair Scheduler, referred to in this paper as HFS.[1] HFS has two main goals:

- *Fair sharing:* divide resources using max-min fair sharing [7] to achieve statistical multiplexing. For example, if

EuroSys'10, April 13–16, 2010, Paris, France.

[1] HFS is open source and available as part of Apache Hadoop.

two jobs are running, each should get half the resources; if a third job is launched, each job's share should be 33%.

- *Data locality:* place computations near their input data, to maximize system throughput.

To achieve the first goal (fair sharing), a scheduler must reallocate resources between jobs when the number of jobs changes. A key design question is what to do with tasks (units of work that make up a job) from running jobs when a new job is submitted, in order to give resources to the new job. At a high level, two approaches can be taken:

1. *Kill* running tasks to make room for the new job.
2. *Wait* for running tasks to finish.

Killing reallocates resources instantly and gives control over locality for the new jobs, but it has the serious disadvantage of wasting the work of killed tasks. Waiting, on the other hand, does not have this problem, but can negatively impact fairness, as a new job needs to wait for tasks to finish to achieve its share, and locality, as the new job may not have any input data on the nodes that free up.

Our principal result in this paper is that, counterintuitively, an algorithm based on waiting can achieve both high fairness and high data locality. We show first that in large clusters, tasks finish at such a high rate that resources can be reassigned to new jobs on a timescale much smaller than job durations. However, a strict implementation of fair sharing compromises locality, because the job to be scheduled next according to fairness might not have data on the nodes that are currently free. To resolve this problem, we relax fairness slightly through a simple algorithm called *delay scheduling*, in which a job waits for a limited amount of time for a scheduling opportunity on a node that has data for it. We show that a very small amount of waiting is enough to bring locality close to 100%. Delay scheduling performs well in typical Hadoop workloads because Hadoop tasks are *short* relative to jobs, and because there are *multiple locations* where a task can run to access each data block.

Delay scheduling is applicable beyond fair sharing. In general, any scheduling policy defines an order in which jobs should be given resources. Delay scheduling only asks that we sometimes give resources to jobs out of order to improve data locality. We have taken advantage of the generality of delay scheduling in HFS to implement a hierarchical scheduling policy motivated by the needs of Facebook's users: a top-level scheduler divides slots between users according to weighted fair sharing, but users can schedule their own jobs using either FIFO or fair sharing.

Although we motivate our work with the data warehousing workload at Facebook, it is applicable in other settings. Our Yahoo! contacts also report job queueing delays to be a big frustration. Our work is also relevant to shared academic Hadoop clusters [8, 10, 14], and to systems other than Hadoop. Finally, one consequence of the simplicity of delay scheduling is that it can be implemented in a distributed fashion; we discuss the implications of this in Section 6.

This paper is organized as follows. Section 2 provides background on Hadoop. Section 3 analyzes a simple model of fair sharing to identify when fairness conflicts with locality, and explains why delay scheduling can be expected to perform well. Section 4 describes the design of HFS and our implementation of delay scheduling. We evaluate HFS and delay scheduling in Section 5. Section 6 discusses limitations and extensions of delay scheduling. Section 7 surveys related work. We conclude in Section 8.

2. Background

Hadoop's implementation of MapReduce resembles that of Google [18]. Hadoop runs over a distributed file system called HDFS, which stores three replicas of each block like GFS [21]. Users submit *jobs* consisting of a map function and a reduce function. Hadoop breaks each job into *tasks*. First, *map tasks* process each input block (typically 64 MB) and produce *intermediate results*, which are key-value pairs. There is one map task per input block. Next, *reduce tasks* pass the list of intermediate values for each key and through the user's reduce function, producing the job's final output.

Job scheduling in Hadoop is performed by a master, which manages a number of slaves. Each slave has a fixed number of *map slots* and *reduce slots* in which it can run tasks. Typically, administrators set the number of slots to one or two per core. The master assigns tasks in response to *heartbeats* sent by slaves every few seconds, which report the number of free map and reduce slots on the slave.

Hadoop's default scheduler runs jobs in FIFO order, with five priority levels. When the scheduler receives a heartbeat indicating that a map or reduce slot is free, it scans through jobs in order of priority and submit time to find one with a task of the required type. For maps, Hadoop uses a locality optimization as in Google's MapReduce [18]: after selecting a job, the scheduler greedily picks the map task in the job with data closest to the slave (on the same node if possible, otherwise on the same rack, or finally on a remote rack).

3. Delay Scheduling

Recall that our goal is to statistically multiplex clusters while having a minimal impact on fairness (*i.e.* giving new jobs their fair share of resources quickly) and achieving high data locality. In this section, we analyze a simple fair sharing algorithm to answer two questions:

1. How should resources be reassigned to new jobs?
2. How should data locality be achieved?

To answer the first question, we consider two approaches to reassigning resources: *killing* tasks from existing jobs to make room for new jobs, and *waiting* for tasks to finish to assign slots to new jobs. Killing has the advantage that it is instantaneous, but the disadvantage that work performed by the killed tasks is wasted. We show that waiting imposes little impact on job response times when jobs are longer than the average task length and when a cluster is shared between

many users. These conditions hold in workloads at Facebook and Yahoo!, so we have based HFS on waiting.

Having chosen to use waiting, we turn our attention to locality. We identify two locality problems that arise when fair sharing is followed strictly – head-of-line scheduling and sticky slots. In both cases, a scheduler is forced to launch a task from a job without local data on a node to maintain fairness. We propose an algorithm called delay scheduling that *temporarily relaxes fairness* to improve locality by asking jobs to wait for a scheduling opportunity on a node with local data. We analyze how the amount of waiting impacts locality and job response times.

For simplicity, we initially focus on one "level" of locality: placing tasks on the same node as their input data. We start taking into account rack locality in Section 3.6.

3.1 Naïve Fair Sharing Algorithm

A simple way to share a cluster fairly between jobs is to always assign free slots to the job that has the fewest running tasks. As long as slots become free quickly enough, the resulting allocation will satisfy max-min fairness [7]. To achieve locality, we can greedily search for a local task in this head-of-line job, as in Hadoop's FIFO scheduler. Pseudocode for this algorithm is shown below:

Algorithm 1 Naïve Fair Sharing

```
when a heartbeat is received from node n:
  if n has a free slot then
    sort jobs in increasing order of number of running tasks
    for j in jobs do
      if j has unlaunched task t with data on n then
        launch t on n
      else if j has unlaunched task t then
        launch t on n
      end if
    end for
  end if
```

We implemented this algorithm in our first version of HFS. We applied the algorithm independently for map slots and reduce slots. In addition, we only used the locality check for map tasks, because reduce tasks normally need to read roughly equal amounts of data from all nodes.

3.2 Scheduling Responsiveness

The first question we consider is how to reassign tasks when new jobs are submitted to a cluster. Ideally, we would like a job j whose fair share is F slots to have a response time similar to what it would get if it ran alone on a smaller, private cluster with F slots. Suppose that j would take J seconds to run on the private cluster. We calculate how long j takes to receive its share of slots if it is submitted to a shared cluster that uses waiting. If all slots in the cluster are full, the rate at which j is given slots will be the rate at which tasks finish. Suppose that the average task length is T, and that the cluster contains S slots. Then one slot will free up every T/S seconds on average, so j is expected to wait FT/S seconds to acquire all of its slots. This wait time will be negligible compared to j's running time as long as:[2]

$$J \gg \frac{F}{S}T \tag{1}$$

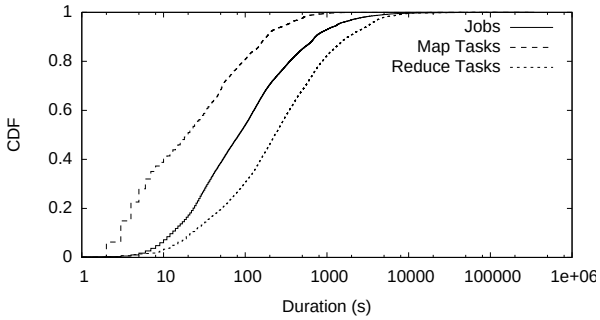


Figure 1: CDF of running times for MapReduce jobs, map tasks and reduce tasks in production at Facebook in October 2009.

Waiting will therefore not impact job response times significantly if at least one of the following conditions holds:

1. *Many jobs:* When there are many jobs running, each job's fractional share of the cluster, $f = \frac{F}{S}$, is small.
2. *Small jobs:* Jobs with a small number of tasks (we call these "small jobs") will also have a small values of f.
3. *Long jobs:* Jobs where $J > T$ incur little overhead.

In workload traces from Facebook, we have found that most tasks are short and most jobs are small, so slots can be reassigned quickly even when the cluster is loaded. Figure 1 shows CDFs of map task lengths, job lengths and reduce task lengths over one week in October 2009. The median map task is 19s long, which is significantly less than the median job length of 84s. Reduces are longer (the median is 231s), but this happens because most of jobs do not have many reduce tasks, so a few jobs with long reduces contribute a large portion of the CDF.

We have also calculated the rate at which slots became free during "periods of load" when most slots of a particular type were full. Map slots were more than 95% full 21% of the time, and on average, during these periods of load, 27.1 slots (out of 3100 total) freed up per second. Reduce slots were more than 95% full only 4% of the time, and during these periods, 3.0 slots out of 3100 freed up per second. Based on the trace, these rates are high enough to let 83% of jobs launch within 10 seconds, because 83% of jobs have fewer than 271 map tasks and 30 reduce tasks.

Finally, we have seen similar task and job lengths to those at Facebook in a 3000-node Yahoo! cluster used for data analytics and ad-hoc queries: the median job was 78s long, the median map was 26s, and the median reduce was 76s.

[2] It is also necessary that task finish times be roughly uniformly distributed in time. This is likely to happen in a large multi-user cluster because task durations are variable and jobs are submitted at variable times.

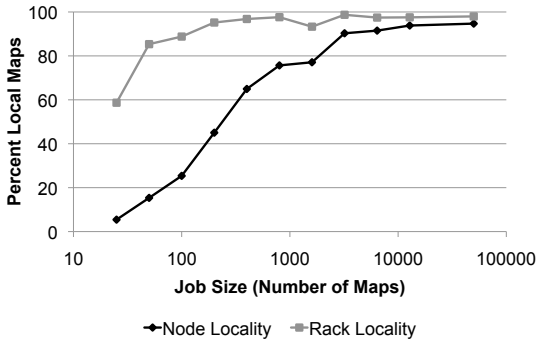


Figure 2: Data locality vs. job size in production at Facebook.

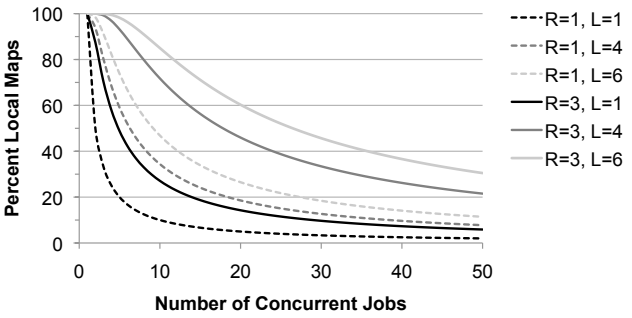


Figure 3: Expected effect of sticky slots on node locality under various values of file replication level (R) and slots per node (L).

3.3 Locality Problems with Naïve Fair Sharing

The main aspect of MapReduce that complicates scheduling is the need to place tasks near their input data. Locality increases throughput because network bandwidth in a large cluster is much lower than the total bandwidth of the cluster's disks [18]. Running on a node that contains the data (*node locality*) is most efficient, but when this is not possible, running on the same rack (*rack locality*) is faster than running off-rack. For now, we only consider node locality. We describe two locality problems that arise with naïve fair sharing: head-of-line scheduling and sticky slots.

3.3.1 Head-of-line Scheduling

The first locality problem occurs in small jobs (jobs that have small input files and hence have a small number of data blocks to read). The problem is that whenever a job reaches the head of the sorted list in Algorithm 1 (*i.e.* has the fewest running tasks), one of its tasks is launched on the next slot that becomes free, no matter which node this slot is on. If the head-of-line job is small, it is unlikely to have data on the node that is given to it. For example, a job with data on 10% of nodes will only achieve 10% locality.

We observed this head-of-line scheduling problem at Facebook in a version of HFS without delay scheduling. Figure 2 shows locality for jobs of different sizes (number of maps) running at Facebook in March 2009. (Recall that there is one map task per input block.) Each point represents a bin of job sizes. The first point is for jobs with 1 to 25 maps, which only achieve 5% node locality and 59% rack locality. Unfortunately, this behavior was problematic because most jobs at Facebook are small. In fact, 58% of Facebook's jobs fall into this first bin (1-25 maps). Small jobs are so common because both ad-hoc queries and periodic reporting jobs work on small data sets.

3.3.2 Sticky Slots

A second locality problem, *sticky slots*, happens even with large jobs if fair sharing is used. The problem is that there is a tendency for a job to be assigned the same slot repeatedly. For example, suppose that there are 10 jobs in a 100-node cluster with one slot per node, and that each job has 10 running tasks. Suppose job j finishes a task on node n. Node n now requests a new task. At this point, j has 9 running tasks while all the other jobs have 10. Therefore, Algorithm 1 assigns the slot on node n to job j again. Consequently, in steady state, *jobs never leave their original slots*. This leads to poor data locality because input files are striped across the cluster, so each job needs to run some tasks on each machine.

The impact of sticky slots depends on the number of jobs, the number of slots per slave (which we shall denote L), and the number of replicas per block in the file system (which we denote R). Suppose that job j's fractional share of the cluster is f. Then for any given block b, the probability that none of j's slots are on a node with a copy of b is $(1-f)^{RL}$: there are R replicas of b, each replica is on a node with L slots, and the probability that a slot does not belong to j is $1-f$. Therefore, j is expected to achieve at most $1-(1-f)^{RL}$ locality. We plot this bound on locality for different R and L and different numbers of concurrent jobs (with equal shares of the cluster) in Figure 3. Even with large R and L, locality falls below 80% for 15 jobs and below 50% for 30 jobs.

Interestingly, sticky slots do not occur in Hadoop due to a bug in how Hadoop counts running tasks. Hadoop tasks enter a "commit pending" state after finishing their work, where they request permission to rename their output to its final filename. The job object in the master counts a task in this state as running, whereas the slave object doesn't. Therefore another job can be given the task's slot. While this is bug (breaking fairness), it has limited impact on throughput and response time. Nonetheless, we explain sticky slots to warn other system designers of the problem. For example, sticky slots have been reported in Dryad [24]. In Section 5, we show that sticky slots lower throughput by 2x in a version of Hadoop without this bug.

3.4 Delay Scheduling

The problems we presented happen because following a strict queuing order forces a job with no local data to be scheduled. We address them through a simple technique called delay scheduling. When a node requests a task, if the head-of-line job cannot launch a local task, we skip it and look at subsequent jobs. However, if a job has been skipped long enough, we start allowing it to launch non-local tasks,

to avoid starvation. The key insight behind delay scheduling is that although the *first* slot we consider giving to a job is unlikely to have data for it, tasks finish so quickly that *some* slot with data for it will free up in the next few seconds.

In this section, we consider a simple version of delay scheduling where we allow a job to be skipped up to D times. Pseudocode for this algorithm is shown below:

Algorithm 2 Fair Sharing with Simple Delay Scheduling

```
Initialize j.skipcount to 0 for all jobs j.
when a heartbeat is received from node n:
  if n has a free slot then
    sort jobs in increasing order of number of running tasks
    for j in jobs do
      if j has unlaunched task t with data on n then
        launch t on n
        set j.skipcount = 0
      else if j has unlaunched task t then
        if j.skipcount ≥ D then
          launch t on n
        else
          set j.skipcount = j.skipcount + 1
        end if
      end if
    end for
  end if
```

Note that once a job has been skipped D times, we let it launch arbitrarily many non-local tasks without resetting its *skipcount*. However, if it ever manages to launch a local task again, we set its *skipcount* back to 0. We explain the rationale for this design in our analysis of delay scheduling.

3.5 Analysis of Delay Scheduling

In this section, we explore how the maximum skip count D in Algorithm 2 affects locality and response times, and how to set D to achieve a target level of locality. We find that:

1. Non-locality decreases exponentially with D.
2. The amount of waiting required to achieve a given level of locality is a fraction of the average task length and decreases linearly with the number of slots per node L.

We assume that we have an M-node cluster with L slots per node, for a total of $S = ML$ slots. Also, at each time, let P_j denote the set of nodes on which job j has data left to process, which we call "preferred" nodes for job j, and let $p_j = \frac{|P_j|}{M}$ be the fraction of nodes that j prefers. To simplify the analysis, we assume that all tasks are of the same length T and that the sets P_j are uncorrelated (for example, either every job has a large input file and therefore has data on every node, or every job has a different input file).

We first consider *how much locality improves* depending on D. Suppose that job j is farthest below its fair share. Then j has probability p_j of having data on each slot that becomes free. If j waits for up to D slots before being allowed to launch non-local tasks, then the probability that it does *not* find a local task is $(1-p_j)^D$. This probability decreases exponentially with D. For example, a job with data on 10% of nodes ($p_j = 0.1$) has a 65% chance of launching a local task with $D = 10$, and a 99% chance with $D = 40$.

A second question is *how long a job waits* below its fair share to launch a local task. Because there are S slots in the cluster, a slot becomes free every $\frac{T}{S}$ seconds on average. Therefore, once a job j reaches the head of the queue, it will wait at most $\frac{D}{S}T$ seconds before being allowed to launch non-local tasks, provided that it stays at the head of the queue.[3] This wait will be much less than the average task length if S is large. In particular, waiting for a local task may cost less time than running a non-local task: in our experiments, local tasks ran up to 2x faster than non-local tasks. Note also that for a fixed number of nodes, the wait time decreases linearly with the number of slots per node.

We conclude with an approximate analysis of *how to set* D to achieve a desired level of locality.[4] Suppose that we wish to achieve locality greater than λ for jobs with N tasks on a cluster with M nodes, L slots per node and replication factor R. We will compute the expected locality for an N-task job j over its lifetime by averaging up the probabilities that it launches a local task when it has $N, N-1, \ldots, 1$ tasks left to launch. When j has K tasks left to launch, $p_j = 1-(1-\frac{K}{M})^R$, because the probability that a given node does *not* have a replica of one of j's input blocks is $(1-\frac{K}{M})^R$. Therefore, the probability that j launches a local task at this point is $1-(1-p_j)^D = 1-(1-\frac{K}{M})^{RD} \geq 1-e^{-RDK/M}$. Averaging this quantity over $K = 1$ to N, the expected locality for job j, given a skip count D, is at least:

$$\begin{aligned} l(D) &= \frac{1}{N}\sum_{K=1}^{N} 1-e^{-RDK/M} \\ &= 1-\frac{1}{N}\sum_{K=1}^{N} e^{-RDK/M} \\ &\geq 1-\frac{1}{N}\sum_{K=1}^{\infty} e^{-RDK/M} \\ &\geq 1-\frac{e^{-RD/M}}{N(1-e^{-RD/M})} \end{aligned}$$

Solving for $l(D) \geq \lambda$, we find that we need to set:

$$D \geq -\frac{M}{R}\ln\left(\frac{(1-\lambda)N}{1+(1-\lambda)N}\right) \qquad (2)$$

For example, for $\lambda = 0.95$, $N = 20$, and $R = 3$, we need $D \geq 0.23M$. Also, the maximum time a job waits for a local task is $\frac{D}{S}T = \frac{D}{LM}T = \frac{0.23}{L}T$. For example, if we have $L = 8$ slots per node, this wait is 2.8% of the average task length.

[3] Once a job reaches the head of the queue, it is likely to stay there, because the head-of-queue job is the one that has the smallest number of running tasks. The slots that the job lost to fall below its share must have been given to other jobs, so the other jobs are likely above their fair share.

[4] This analysis does not consider that a job can launch non-local tasks without waiting after it launches its first one. However, this only happens towards the end of a job, so it does not matter much in large jobs. On the flip side, the inequalities we use underestimate the locality for a given D.

3.5.1 Long Tasks and Hotspots

The preceding analysis assumed that all tasks were of the same length and that job's preferred location sets, P_j, were uncorrelated. Two factors can break these assumptions:

1. Some jobs may have long tasks. If all the slots on a node are filled with long tasks, the node may not free up quickly enough for other jobs to achieve locality.
2. Some nodes may be of interest to many jobs. We call these nodes *hotspots*. For example, multiple jobs may be trying to read the same small input file.

We note that both hotspots and nodes filled with long tasks are relatively long-lasting conditions. This is why, in Algorithm 2, we allow jobs to launch arbitrarily many non-local tasks if they have been skipped D times, until they launch a local task again. If D is set high enough that a job has a good chance of launching a local task on one of its preferred nodes when these nodes are not "blocked" by hotspots or long tasks, then once a job has been skipped D times, it is likely that the job's preferred nodes *are* indeed blocked, so we should not continue waiting.

How much long tasks and hotspots impact locality depends on the workload, the file replication level R, and the number of slots per node L. In general, unless long tasks and hotspots are very common, they will have little impact on locality. For example, if the fraction of slots running long tasks is φ, then the probability that all the nodes with replicas of a given block are filled with long tasks is φ^{RL}. On a cluster with $R = 3$ and $L = 6$, this is less than 2% as long as $\varphi < 0.8$. We have not seen significant problems with long tasks and hotspots in practice. Nonetheless, for workloads where these conditions are common, we propose two solutions:

Long Task Balancing: To lower the chance that a node fills with long tasks, we can spread long tasks throughout the cluster by changing the locality test in Algorithm 2 to prevent jobs with long tasks from launching tasks on nodes that are running a higher-than-average number of long tasks. Although we do not know which jobs have long tasks in advance, we can treat new jobs as long-task jobs, and mark them as short-task jobs if their tasks finish quickly.[5]

Hotspot Replication: Because distributed file systems like HDFS place blocks on random nodes, hotspots are only likely to occur if multiple jobs need to read the same data file, and that file is small enough that copies of its blocks are only present on a small fraction of nodes. In this case, no scheduling algorithm can achieve high locality without excessive queueing delays. Instead, it would be better to dynamically increase the replication level of small hot files.

3.6 Rack Locality

Networks in large clusters are typically organized in a multi-level hierarchy, where nodes are grouped into racks of 20-80 nodes at the lowest level, and one or more levels of aggregation switches connects the racks [24]. Usually, bandwidth per node within a rack is much higher than bandwidth per node between racks. Therefore, when a task cannot be placed on a node that contains its data, it is preferable to place it on a rack that contains the data.

This can be accomplished by extending Algorithm 2 to give each job two waiting periods. First, if the head-of-line job has been skipped at most D_1 times, it is only allowed to launch node-local tasks. Once a job has been skipped D_1 times, it enters a second "level" of delay scheduling, where it is only allowed to launch rack-local tasks. If the job is skipped D_2 times while at this level, it is allowed to launch non-local tasks. A nice consequence of our analysis is that D_2 can be much smaller than D_1: because there are much fewer racks than nodes, a job will not be skipped many times until it finds a slot in a rack that contains its data.

We have implemented this algorithm in HFS, and describe it in detail in Section 4.1. A similar algorithm can be used for networks with more than 2 levels of hierarchy.

4. Hadoop Fair Scheduler Design

The simple fair scheduler described in Section 3, which gives each job an equal share of the cluster, is adequate for clusters with small numbers of users. However, to handle the needs of a larger organization such as Facebook, we wanted to address several shortcomings of this model:

1. Some users may be running more jobs than others; we want fair sharing at the level of users, not jobs.
2. Users want control over the scheduling of their own jobs. For example, a user who submits several batch jobs may want them to run in FIFO order to get their results sequentially.
3. Production jobs need to perform predictably even if the cluster is loaded with many long user tasks.

We address these problems in our design of the Hadoop Fair Scheduler (HFS). HFS uses a two-level scheduling hierarchy. At the top level, HFS allocates task slots across *pools* using weighted fair sharing. There is one pool per user, but organizations can also create special pools for particular workloads (e.g. production jobs). At the second level, each pool allocates its slots among jobs in the pool, using either FIFO with priorities or a second level of fair sharing. Figure 4 shows an example pool hierarchy. HFS can easily be generalized to support multi-level pool hierarchies, or policies other than FIFO and fair sharing within a pool.

We provide one feature beyond standard weighted fair sharing to support production jobs. Each pool can be given a *minimum share*, representing a minimum number of slots that the pool is guaranteed to be given as long as it contains jobs, even if the pool's fair share is less than this amount (e.g. because many users are running jobs). HFS always prioritizes meeting minimum shares over fair shares, and may kill tasks to meet minimum shares. Administrators are expected to set minimum shares for production jobs based

[5] Tasks within a job have similar lengths because they run the same function.

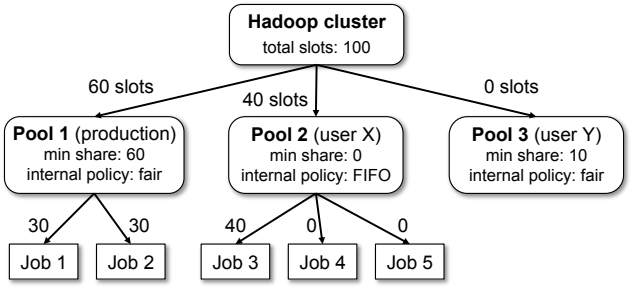


Figure 4: Example of allocations in HFS. Pools 1 and 3 have minimum shares of 60 and 10 slots. Because Pool 3 is not using its share, its slots are given to Pool 2. Each pool's internal scheduling policy (FIFO or fair sharing) splits up its slots among its jobs.

on the number of slots a job needs to meet a certain SLO (e.g. import logs every 15 minutes, or delete spam messages every hour). If the sum of all pools' minimum shares exceeds the number of slots in the cluster, HFS logs a warning and scales down the minimum shares equally until their sum is less than the total number of slots.

Finally, although HFS uses waiting to reassign resources most of the time, it also supports task killing. We added this support to prevent a buggy job with long tasks, or a greedy user, from holding onto a large share of the cluster. HFS uses two task killing timeouts. First, each pool has a *minimum share timeout*, T_{min}. If the pool does not receive its minimum share within T_{min} seconds of a job being submitted to it, we kill tasks to meet the pool's share. Second, there is a global *fair share timeout*, T_{fair}, used to kill tasks if a pool is being starved of its fair share. We expect administrators to set T_{min} for each production pool based on its SLO, and to set a larger value for T_{fair} based on the level of delay users can tolerate. When selecting tasks to kill, we pick the most recently launched tasks in pools that are above their fair share to minimize wasted work.

4.1 Task Assignment in HFS

Whenever a slot is free, HFS assigns a task to it through a two-step process: First, we create a sorted list of jobs according to our hierarchical scheduling policy. Second, we scan down this list to find a job to launch a task from, applying delay scheduling to skip jobs that do not have data on the node being assigned for a limited time. The same algorithm is applied independently for map slots and reduce slots, although we do not use delay scheduling for reduces because they usually need to read data from all nodes.

To create a sorted list of jobs, we use a recursive algorithm. First, we sort the pools, placing pools that are below their minimum share at the head of the list (breaking ties based on how far each pool is below its minimum share), and sorting the other pools by $\frac{\text{current share}}{\text{weight}}$ to achieve weighted fair sharing. Then, within each pool, we sort jobs based on the pool's internal policy (FIFO or fair sharing).

Our implementation of delay scheduling differs slightly from the simplified algorithm in Section 3.4 to take into account some practical considerations. First, rather than using a maximum skip count D to determine how long a job waits for a local task, we set a maximum *wait time* in seconds. This allows jobs to launch within a predictable time when a large number of slots in the cluster are filled with long tasks and slots free up at a slow rate. Second, to achieve rack locality when a job is unable to launch node-local tasks, we use two levels of delay scheduling – jobs wait W_1 seconds to find a node-local task, and then W_2 seconds to find a rack-local task. This algorithm is shown below:

Algorithm 3 Delay Scheduling in HFS

Maintain three variables for each job j, initialized as $j.level = 0$, $j.wait = 0$, and $j.skipped = false$.
when a heartbeat is received from node n:
  for each job j with $j.skipped = true$, increase $j.wait$ by the time since the last heartbeat and set $j.skipped = false$
  if n has a free slot **then**
    sort *jobs* using hierarchical scheduling policy
    for j in *jobs* **do**
      if j has a node-local task t on n **then**
        set $j.wait = 0$ and $j.level = 0$
        return t to n
      else if j has a rack-local task t on n and ($j.level \geq 1$ or $j.wait \geq W_1$) **then**
        set $j.wait = 0$ and $j.level = 1$
        return t to n
      else if $j.level = 2$ or ($j.level = 1$ and $j.wait \geq W_2$) or ($j.level = 0$ and $j.wait \geq W_1 + W_2$) **then**
        set $j.wait = 0$ and $j.level = 2$
        return any unlaunched task t in j to n
      else
        set $j.skipped = true$
      end if
    end for
  end if

Each job begins at a "locality level" of 0, where it can only launch node-local tasks. If it waits at least W_1 seconds, it goes to locality level 1 and may launch rack-local tasks. If it waits a further W_2 seconds, it goes to level 2 and may launch off-rack tasks. Finally, if a job ever launches a "more local" task than the level it is on, it goes back down to a previous level, as motivated in Section 3.5.1. The algorithm is straightforward to generalize to more locality levels for clusters with more than a two-level network hierarchy.

We expect administrators to set the wait times W_1 and W_2 based on the rate at which slots free up in their cluster and the desired level of locality, using the analysis in Section 3.5. For example, at Facebook, we see 27 map slots freeing per second when the cluster is under load, files are replicated $R = 3$ ways, and there are $M = 620$ machines. Therefore, setting $W_1 = 10$s would give each job roughly $D = 270$ scheduling opportunities before it is allowed to launch non-local tasks. This is enough to let jobs with $K = 1$ task achieve at least $1 - e^{-RDK/M} = 1 - e^{-3 \cdot 270 \cdot 1/620} = 73\%$ locality, and to let jobs with 10 tasks achieve 90% locality.

Environment	Nodes	Hardware and Configuration
Amazon EC2	100	4 2GHz cores, 4 disks and 15 GB RAM per node. Appears to have 1 Gbps links. 4 map and 2 reduce slots per node.
Private Cluster	100	8 cores and 4 disks per node. 1 Gbps Ethernet. 4 racks. 6 map and 4 reduce slots per node.

Table 1: Experimental environments used in evaluation.

5. Evaluation

We have evaluated delay scheduling and HFS through a set of macrobenchmarks based on the Facebook workload, microbenchmarks designed to test hierarchical scheduling and stress delay scheduling, a sensitivity analysis, and an experiment measuring scheduler overhead.

We ran experiments in two environments: Amazon's Elastic Compute Cloud (EC2) [1] and a 100-node private cluster. On EC2, we used "extra-large" VMs, which appear to occupy a whole physical nodes. Both environments are atypical of large MapReduce installations because they have fairly high bisection bandwidth; the private cluster spanned only 4 racks, and while topology information is not provided by EC2, tests revealed that nodes were able to send 1 Gbps to each other. Therefore, our experiments understate potential performance gains from locality. We ran a modified version of Hadoop 0.20, configured with a block size of 128 MB because this improved performance (Facebook uses this setting in production). We set the number of task slots per node in each cluster based on hardware capabilities. Table 1 lists the hardware and slot counts in each environment.

5.1 Macrobenchmarks

To evaluate delay scheduling and HFS on a multi-user workload, we ran a set macrobenchmarks based on the workload at Facebook on EC2. We generated a submission schedule for 100 jobs by sampling job inter-arrival times and input sizes from the distribution seen at Facebook over a week in October 2009. We ran this job submission schedule with three workloads based on the Hive benchmark [5] (which is itself based on Pavlo et al's benchmark comparing MapReduce to parallel databases [26]): an IO-heavy workload, in which all jobs were IO-bound; a CPU-heavy workload, in which all jobs were CPU-bound; and a mixed workload, which included all the jobs in the benchmark. For each workload, we compared response times and data locality under FIFO scheduling, naïve fair sharing, and fair sharing with delay scheduling. We now describe our experimental methodology in detail, before presenting our results.

We began by generating a common job submission schedule that was shared by all the experiments. We chose to use the same schedule across experiments so that elements of "luck," such as a small job submitted after a large one, happened the same number of times in all the experiments. However, the schedule was long enough (100 jobs) to contain a variety of behaviors. To generate the schedule, we first sampled job inter-arrival times at random from the Facebook trace. This distribution of inter-arrival times was roughly exponential with a mean of 14 seconds, making the total submission schedule 24 minutes long.

Bin	# Maps	% Jobs at Facebook	# Maps in Benchmark	# Jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3–20	14%	10	14
4	21–60	9%	50	8
5	61–150	6%	100	6
6	151–300	6%	200	6
7	301–500	4%	400	4
8	501–1500	4%	800	4
9	> 1501	3%	4800	4

Table 2: Distribution of job sizes (in terms of number of map tasks) at Facebook and in our macrobenchmarks.

We also generated job input sizes based on the Facebook workload, by looking at the distribution of number of map tasks per job at Facebook and creating datasets with the correct sizes (because there is one map task per 128 MB input block). We quantized the job sizes into nine bins, listed in Table 2, to make it possible to compare jobs in the same bin within and across experiments. We note that most jobs at Facebook are small, but the last bin, for jobs with more than 1501 maps, contains some very large jobs: the 98$^{\text{th}}$ percentile job size is 3065 map tasks, the 99$^{\text{th}}$ percentile is 3846 maps, the 99.5$^{\text{th}}$ percentile is 6232 maps, and the largest job in the week we looked at had over 25,000 maps.[6] We chose 4800 maps as our representative for this bin to pose a reasonable load while remaining manageable for our EC2 cluster.

We used our submission schedule for three workloads (IO-heavy, CPU-heavy, and mixed), to evaluate the impact of our algorithms for organizations with varying job characteristics. We chose the actual jobs to run in each case from the Hive benchmark [5], which contains Hive versions of four queries from Pavlo et al's MapReduce benchmark [26]: text search, a simple filtering selection, an aggregation, and a join that gets translated into multiple MapReduce steps.

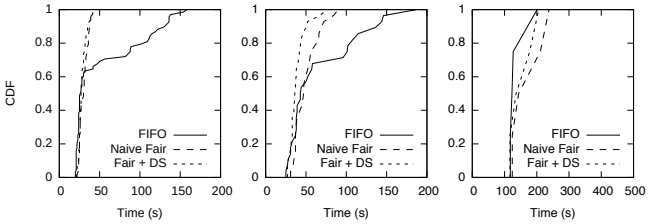
Finally, we ran each workload under three schedulers: FIFO (Hadoop's default scheduler), naïve fair sharing (*i.e.* Algorithm 1), and fair sharing with 5-second delay scheduling. For simplicity, we submitted each job as a separate user, so that jobs were entitled to equal shares of the cluster.

5.1.1 Results for IO-Heavy Workload

To evaluate our algorithms on an IO-heavy workload, we picked the text search job out of the Hive benchmark, which scans through a data set and prints out only the records that contain a certain pattern. Only 0.01% of the records contain the pattern, so the job is almost entirely bound by disk IO.

Our results are shown in Figures 5, 6, and 7. First, Figure 5 shows a CDF of job running times for various ranges of

[6] Many of the smallest jobs are actually periodic jobs that run several times per hour to import external data into the cluster and generate reports.



(a) Bins 1-3 (b) Bins 4-8 (c) Bin 9

Figure 5: CDFs of running times of jobs in various bin ranges in the IO-heavy workload. Fair sharing greatly improves performance for small jobs, at the cost of slowing the largest jobs. Delay scheduling further improves performance, especially for medium-sized jobs.

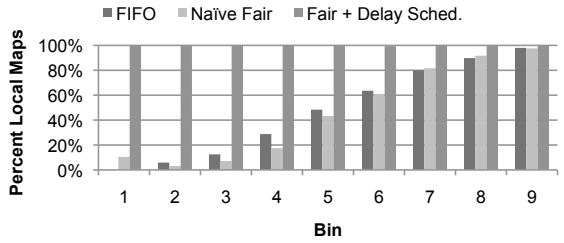


Figure 6: Data locality for each bin in the IO-heavy workload.

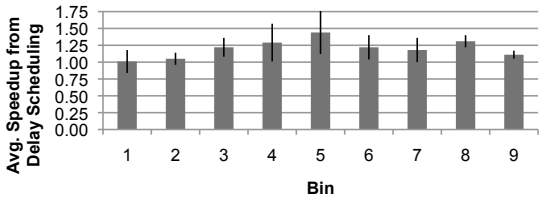
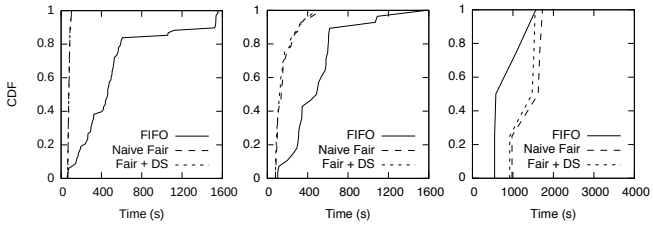


Figure 7: Average speedup of delay scheduling over naïve fair sharing for jobs in each bin in the IO-heavy workload. The black lines show standard deviations.

bins. We see that about 30% of the smaller jobs are significantly slowed down under FIFO scheduling, because they must wait for a larger job to finish. Switching to fair sharing resolves this problem, letting all the small jobs perform nearly equally well no matter when they are launched. The job with the greatest improvement runs 5x faster under fair sharing than FIFO. On the other hand, fair sharing slows down the largest jobs (in bin 9), because it lets other jobs run while they are active. The greatest slowdown, of about 1.7x, happens to two jobs from bin 9 that overlap in time. This is expected behavior for fair sharing: predictability and response times for small jobs are improved at the expense of moderately slowing down larger jobs.

Second, we observe that adding delay scheduling to fair sharing improves performance overall. As shown in Figure 6, delay scheduling brings the data locality to 99-100% for all bins, whereas bins with small jobs have low data locality under both Hadoop's default scheduler and naïve fair sharing. The effect on response time is more nuanced, and to illustrate it clearly, we have plotted Figure 7, which shows the average speedup experienced by jobs in each bin when moving from naïve fair sharing to delay scheduling. We see that delay scheduling has a negligible effect on the smallest jobs; this is partly because much of the lifetime of small jobs is setup, and partly because the cluster is actually underloaded most of the time, and small jobs are launched roughly uniformly throughout time, so most of them do not face network contention. Delay scheduling also has less of an effect on the largest jobs; these have many input blocks on every node, so they achieve high locality even with Hadoop's greedy default algorithm. However, significant speedups are seen for medium-sized jobs, with jobs in bin 5 (100 maps) running on average 44% faster with delay scheduling.



(a) Bins 1-3 (b) Bins 4-8 (c) Bin 9

Figure 8: CDFs of running times of jobs in various bin ranges in the CPU-heavy workload. Fair sharing speeds up smaller jobs while slowing down the largest ones, but delay scheduling has little effect because the workload is CPU-bound.

5.1.2 Results for CPU-Heavy Workload

To create a CPU-heavy workload, we modified the text search job in the Hive benchmark to run a compute-intensive user defined function (UDF) on each input record, but still output only 0.01% of records. This made the jobs 2-7 times slower (the effect was more pronounced for large jobs, because much of the lifetime of smaller jobs is Hadoop job setup overhead). We observed data locality levels very close to those in the IO-heavy workload, so we do not plot them here. However, we have plotted job response time CDFs in Figure 8. We note two behaviors: First, fair sharing improves response times of small jobs as before, but its effect is much larger (speeding some jobs as much as 20x), because the cluster is more heavily loaded (we are running on the same data but with more expensive jobs). Second, delay scheduling has a negligible effect, because the workload is CPU-bound, but it also does not hurt performance.

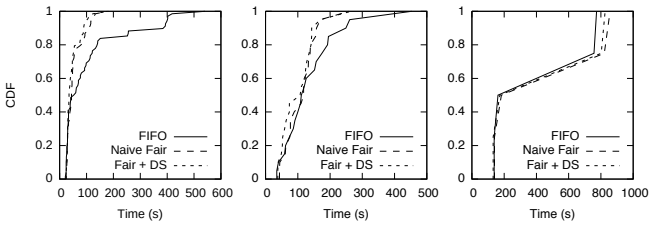
5.1.3 Results for Mixed Workload

We generated a mixed workload by running all four of the jobs in the Hive benchmark. Apart from the text search job used in the IO-heavy workload, this includes:

- A simple select that is also IO-intensive (selecting pages with a certain PageRank).
- An aggregation job that is communication-intensive (computing ad revenue for each IP address in a dataset).
- A complex join query that translates into a series of four jobs (identifying the user that generated the most revenue and the average PageRank of their pages).

Bin	Job Type	Map Tasks	Reduce Tasks	# Jobs Run
1	select	1	NA	38
2	text search	2	NA	16
3	aggregation	10	3	14
4	select	50	NA	8
5	text search	100	NA	6
6	aggregation	200	50	6
7	select	400	NA	4
8	aggregation	800	180	4
9	join	2400	360	2
10	text search	4800	NA	2

Table 3: Job types and sizes for each bin in our mixed workload.



(a) Bins 1-3 (b) Bins 4-8 (c) Bins 9-10

Figure 9: CDFs of running times of jobs in various bin ranges in the mixed workload.

For this experiment, we split bin 9 into two smaller bins, one of which contained 2 join jobs (which translate into a large 2400-map job followed by three smaller jobs each) and another of which contained two 4800-map jobs as before. We list the job we used as a representative in each bin in Table 3. Unlike our first two workloads, which had map-only jobs, this workload also contained jobs with reduce tasks, so we also list the number of reduce tasks per job.

We plot CDFs of job response times in each bin in Figure 9. As in the previous experiments, fair sharing significantly improves the response time for smaller jobs, while slightly slowing larger jobs. Because the aggregation jobs take longer than map-only jobs (due to having a communication-heavy reduce phase), we have also plotted the speedups achieved by each bin separately in Figure 10. The dark bars show speedups for naïve fair sharing over FIFO, while the light bars show speedups for fair sharing with delay scheduling over FIFO. The smaller map-only jobs (bins 1 and 2) achieve significant speedups from fair sharing. Bin 3 does not achieve a speedup as high as in other experiments because the jobs are longer (the median one is about 100 seconds, while the median in bins 1 and 2 is 32 seconds with delay scheduling). However, in all but the largest bins, jobs benefit from both fair sharing and delay scheduling. We also see that the benefits from delay scheduling are larger for the bins with IO-intensive jobs (1, 2, 4, 5, 7 and 10) than for bins where there are also reduce tasks (and hence a smaller fraction of the job running time is spent reading input).

5.2 Microbenchmarks

We ran several microbenchmarks to test HFS in a more controlled manner, and to stress-test delay scheduling in situations where locality is difficult to achieve. For these experiments, we used a "scan" included in Hadoop's GridMix benchmark. This is a synthetic job in which each map outputs 0.5% of its input records, similar to the text search job in the macrobenchmarks. As such, it is a suitable workload for stress-testing data locality because it is IO-bound. The scan job normally has one reduce task that counts the results, but we also ran some experiments with no reduces (saving map outputs as the job's output) to emulate pure filtering jobs.

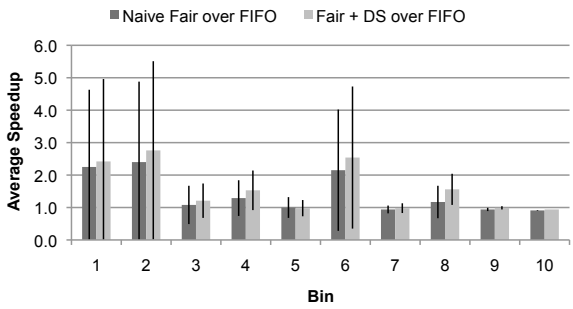


Figure 10: Average speedup of naïve fair sharing and fair sharing with delay scheduling over FIFO for jobs in each bin in the mixed workload. The black lines show standard deviations.

5.2.1 Hierarchical Scheduling

To evaluate the hierarchical scheduling policy in HFS and measure how quickly resources are given to new jobs, we set up three pools on the EC2 cluster. Pools 1 and 2 used fair sharing as their internal policy, while pool 3 used FIFO. We then submitted a sequence of jobs to test both sharing between pools and scheduling within a pool. Figure 11 shows a timeline of the experiment. Delay scheduling (with $W_1 = 5$s) was also enabled, and all jobs achieved 99-100% locality.

We used two types of filter jobs: two long jobs with long tasks (12000 map tasks that each took 25s on average) and four jobs with short tasks (800 map tasks that each took 12s on average). To make the first type of jobs have longer tasks, we set their filtering rate to 50% instead of 0.5%.

We began by submitting a long-task job to pool 1 at time 0. This job was given tasks on all the nodes in the cluster. Then, at time 57s, we submitted a second long-task job in pool 2. This job reached its fair share (half the cluster) in 17 seconds. Then, at time 118s, we submitted three short-task jobs to pool 3. The pool ran acquired 33% of the slots in the cluster in 12 seconds and scheduled its jobs in FIFO order, so that as soon as the first job finished tasks, slots were given to the second job. Once pool 3's jobs finished, the cluster returned to being shared equally between pools 1 and 2. Finally, at time 494s, we submitted a second job in pool 1. Because pool 1 was configured to perform fair sharing, it split up its slots between its jobs, giving them 25% of the slots each, while pool 2's share remained 50%.

Note that the graph in Figure 11 shows a "bump" in the share of pool 2 twenty seconds after it starts running jobs, and a smaller bump when pool 3 starts. These bumps occur

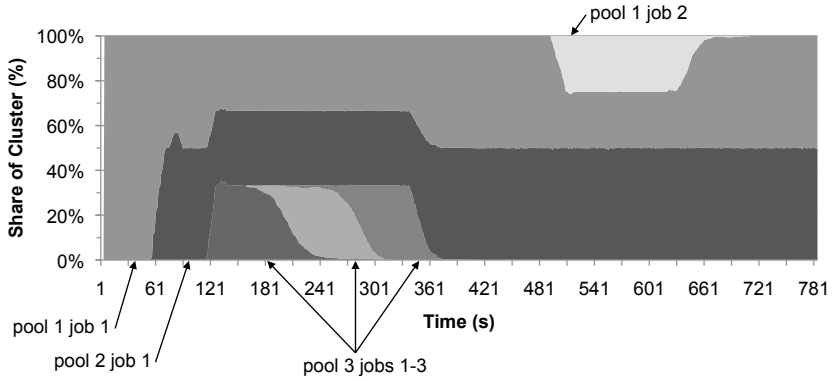


Figure 11: Stacked chart showing the percent of map slots in the cluster given to each job as a function of time in our hierarchical scheduling experiment. Pools 1 and 2 use fair sharing internally, while pool 3 uses FIFO. The job submission scheduled is explained in the text.

Job Size	Node / Rack Locality Without Delay Sched.	Node / Rack Locality With Delay Sched.
3 maps	2% / 50%	75% / 96%
10 maps	37% / 98%	99% / 100%
100 maps	84% / 99%	94% / 99%

Table 4: Node and rack locality in small-jobs stress test workload. Results were similar for FIFO and fair sharing.

because of the "commit pending" bug in Hadoop discussed in Section 3.3.2. Hadoop tasks enter a "commit pending" phase after they finish running the user's map function when they are still reported as running but a second task can be launched in their slot. However, during this time, the job object in the Hadoop master counts the task as running, while the slave object doesn't. Normally, a small percent of tasks from each jobs are in the "commit pending" state, so the bug doesn't affect fairness. However, when pool 2's first job is submitted, none of its tasks finish until about 20 seconds pass, so it holds onto a greater share of the cluster than 50%. (We calculated each job's share as the percent of running tasks that belong to it when we plotted Figure 11.)

5.2.2 Delay Scheduling with Small Jobs

To test the effect of delay scheduling on locality and throughput in a small job workload where head-of-line scheduling poses a problem, we ran workloads consisting of filter jobs with 3, 10 or 100 map tasks on the private cluster. For each workload, we picked the number of jobs based on the job size so as to have the experiment take 10-20 minutes. We compared fair sharing and FIFO with and without delay scheduling ($W_1 = W_2 = 15$s). FIFO performed the same as fair sharing, so we only show one set of numbers for both.

Figure 12 shows normalized running times of the workload, while Table 4 shows locality achieved by each scheduler. Delay scheduling increased throughput by 1.2x for 3-map jobs, 1.7x for 10-map jobs, and 1.3x for 100-map jobs, and raised data locality to at least 75% and rack locality to at least 94%. The throughput gain is higher for 10-map jobs than for 100-map jobs because locality with 100-map jobs is fairly good even without delay scheduling. The gain for the

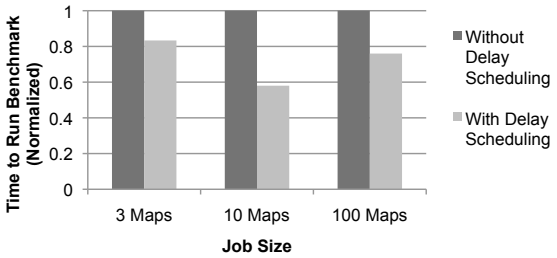


Figure 12: Performance of small-jobs stress test with and without delay scheduling. Results were similar for FIFO and fair sharing.

3-map jobs was low because, at small job sizes, job initialization becomes a bottleneck in Hadoop. Interestingly, the gains with 10 and 100 maps were due to moving from rack-local to node-local tasks; rack locality was good even without delay scheduling because our cluster had only 4 racks.

5.2.3 Delay Scheduling with Sticky Slots

As explained in Section 3.3, sticky slots do not normally occur in Hadoop due to an accounting bug. We tested a version of Hadoop with this bug fixed to quantify the effect of sticky slots. We ran this test on EC2. We generated a large 180-GB data set (2 GB per node), submitted between 5 and 50 concurrent scan jobs on it, and measured the time to finish all jobs and the locality achieved. Figures 14 and 13 show the results with and without delay scheduling (with $W_1 = 10$s). Without delay scheduling, locality was lower the more concurrent jobs there were – from 92% with 5 jobs down to 27% for 50 jobs. Delay scheduling raised locality to 99-100% in all cases. This led to an increase in throughput of 1.1x for 10 jobs, 1.6x for 20 jobs, and 2x for 50 jobs.

5.3 Sensitivity Analysis

We measured the effect of the wait time in delay scheduling on data locality through a series of experiments in the EC2 environment. We ran experiments with two small job sizes: 4 maps and 12 maps, to measure how well delay scheduling mitigates head-of-line scheduling. We ran 200 jobs in each experiment, with 50 jobs active at any time. We varied the node locality wait time, W_1, from 0 seconds to 10

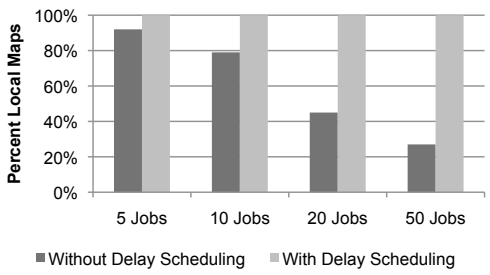


Figure 13: Node locality in sticky slots stress test. As the number of concurrent jobs grows, locality falls because of sticky slots.

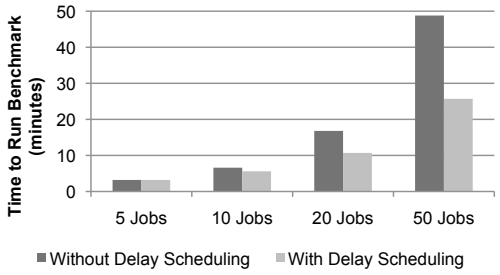


Figure 14: Finish times in sticky slots stress test. When delay scheduling is not used, performance decreases as the number of jobs increases because data locality decreases. In contrast, finish times with delay scheduling grow linearly with the number of jobs.

seconds. There was no rack locality because we do not have information about racks on EC2; however, rack locality will generally be much higher than node locality because there are more slots per rack. Figure 15 shows the results. We see that without delay scheduling, both 4-map jobs and 12-map jobs have poor locality (5% and 11%). Setting W_1 as low as 1 second improves locality to 68% and 80% respectively. Increasing the delay to 5s achieves nearly perfect locality. Finally, with a 10s delay, we got 100% locality for the 4-map jobs and 99.8% locality for the 12-map jobs.

5.4 Scheduler Overhead

In our 100-node experiments, HFS did not add any noticeable scheduling overhead. To measure the performance of HFS under a much heavier load, we used mock objects to simulate a cluster with 2500 nodes and 4 slots per node (2 map and 2 reduce), running a 100 jobs with 1000 map and 1000 reduce tasks each that were placed into 20 pools. Under this workload, HFS was able to schedule 3200 tasks per second on a 2.66 GHz Intel Core 2 Duo. This is several times more than is needed to manage cluster of this size running reasonably-sized tasks (*e.g.*, if the average task length is 10s, there will only be 1000 tasks finishing per second).

6. Discussion

Underlying our work is a classic tradeoff between utilization and fairness. In provisioning a cluster computing infrastructure, there is a spectrum between having a separate cluster per user, which provides great fairness but poor utilization, and having a single FIFO cluster, which provides great utilization but no fairness. Our work enables a sweet spot on this spectrum – multiplexing a cluster efficiently while giving each user response times comparable to a private cluster through fair sharing.

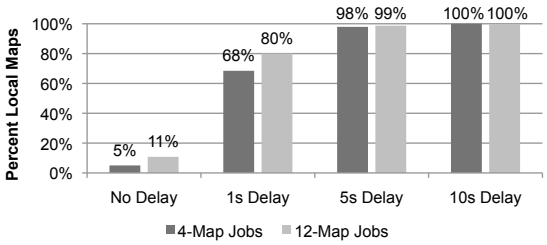


Figure 15: Effect of delay scheduling's wait time W_1 on node locality for small jobs with 4 and 12 map tasks. Even delays as low as 1 second raise locality from 5% to 68% for 4-map jobs.

To implement fair sharing, we had to consider two other tradeoffs between utilization and fairness: first, whether to kill tasks or wait for them to finish when new jobs are submitted, and second, how to achieve data locality. We have proposed a simple strategy called delay scheduling that achieves both fairness and locality by waiting for tasks to finish. Two key aspects of the cluster environment enable delay scheduling to perform well: first, *most tasks are short* compared to jobs, and second, there are *multiple locations* in which a task can run to read a given data block, because systems like Hadoop support multiple task slots per node.

Delay scheduling performs well in environments where these two conditions hold, which include the Hadoop environments at Yahoo! and Facebook. Delay scheduling will *not* be effective if a large fraction of tasks is much longer than the average job, or if there are few slots per node. However, as cluster technology evolves, we believe that both of these factors will improve. First, making tasks short improves fault tolerance [18], so as clusters grow, we expect more developers to split their work into short tasks. Second, due to multicore, cluster nodes are becoming "bigger" and can thus support more tasks at once. In the same spirit, organizations are putting more disks per node – for example, Google used 12 disks per node in its petabyte sort benchmark [9]. Lastly, 10 Gbps Ethernet will greatly increase network bandwidth within a rack, and may allow rack-local tasks to perform as well as node-local tasks. This would increase the number of locations from which a task can efficiently access its input block by an order of magnitude.

Because delay scheduling only involves being able to skip jobs in a sorted order that captures "who should be scheduled next," we believe that it can be used in a variety of environments beyond Hadoop and HFS. We now discuss several ways in which delay scheduling can be generalized.

Scheduling Policies other than Fair Sharing: Delay scheduling can be applied to any queuing policy that produces a sorted list of jobs. For example, in Section 5.2.2, we showed that it can also double throughput under FIFO.

Scheduling Preferences other than Data Locality: Some jobs may prefer to run multiple tasks in the *same* location rather than running each task near its input block. For example, some Hadoop jobs have a large data file that is shared by all tasks and is dynamically copied onto nodes that run the job's tasks using a feature called the distributed cache [4, 12]. In this situation, the locality test in our algorithm can be changed to prefer running tasks on nodes that have the cached file. To allow a cluster to be shared between jobs that want to reuse their slots and jobs that want to read data spread throughout the cluster, we can distribute tasks from the former throughout the cluster using the load balancing mechanism proposed for long tasks in Section 3.5.1.

Load Management Mechanisms other than Slots: Tasks in a cluster may have heterogeneous resource requirements. To improve utilization, the number of tasks supported on a node could be varied dynamically based on its load rather than being fixed as in Hadoop. As long as each job has a roughly equal chance of being scheduled on each node, delay scheduling will be able to achieve data locality.

Distributed Scheduling Decisions: We have also implemented delay scheduling in Nexus [22], a two-level cluster scheduler that allows multiple instances of Hadoop, or of other cluster computing frameworks, to coexist on a shared cluster. In Nexus, a master process from each framework registers with the Nexus master to receive slots on the cluster. The Nexus master schedules slots by making "slot offers" to the appropriate framework (using fair sharing), but frameworks are allowed to reject an offer to wait for a slot with better data locality. We have seen locality improvements similar to those in Section 5 when running multiple instances of Hadoop on Nexus with delay scheduling. The fact that high data locality can be achieved in a distributed fashion provides significant practical benefits: first, multiple isolated instances of Hadoop can be run to ensure that experimental jobs do not crash the instance that runs production jobs; second, multiple versions of Hadoop can coexist; and lastly, organizations can use multiple cluster computing frameworks and pick the best one for each application.

7. Related Work

Scheduling for Data-Intensive Cluster Applications: The closest work we know of to our own is Quincy [24], a fair scheduler for Dryad. Quincy also tackles the conflict between locality and fairness in scheduling, but uses a very different mechanism from HFS. Each time a scheduling decision needs to be made, Quincy represents the scheduling problem as an optimization problem, in which tasks must be matched to nodes and different assignments have different costs based on locality and fairness. Min-cost flow is used to solve this problem. Quincy then kills some of the running tasks and launches new tasks to place the cluster in the configuration returned by the flow solver.

While killing tasks may be the most effective way to reassign resources in some situations, it wastes computation. Our work shows that waiting for suitable slots to free up can also be effective in a diverse real-world workload. One of the main differences between our environment and Quincy's is that Hadoop has multiple task slots per node, while the system in [24] only ran one task per node. The probability that all slots with local copies of a data block are filled by long tasks (necessitating killing) decreases exponentially with the number of slots per node, as shown in Section 3.5.1. Another important difference is that task lengths are much shorter than job lengths in typical Hadoop workloads.

At first sight, it may appear that Quincy uses more information about the cluster than HFS, and hence should make better scheduling decisions. However, HFS also uses information that is *not* used by Quincy: delay scheduling is based on knowledge about the *rate* at which slots free up. Instead of making scheduling decisions based on point snapshots of the state of the cluster, we take into account the fact that many tasks will finish in the near future.

Finally, delay scheduling is simpler than the optimization approach in Quincy, which makes it easy to use with scheduling policies other than fair sharing, as we do in HFS.

High Performance Computing (HPC): Batch schedulers for HPC clusters, like Torque [13], support job priority and resource-consumption-aware scheduling. However, HPC jobs run on a fixed number of machines, so it is not possible to change jobs' allocations over time as we do in Hadoop to achieve fair sharing. HPC jobs are also usually CPU or communication bound, so there is less need for data locality.

Grids: Grid schedulers like Condor [28] support locality constraints, but usually at the level of geographic sites, because the jobs are more compute-intensive than MapReduce. Recent work also proposes replicating data across sites on demand [17]. Similarly, in BAD-FS [16], a workload scheduler manages distribution of data across a wide-area network to dedicated storage servers in each cluster. Our work instead focuses on task placement in a local-area cluster where data is stored on the same nodes that run jobs.

Parallel Databases: Like MapReduce, parallel databases run data-intensive workloads on a cluster. However, database queries are usually executed as long-running processes rather than short tasks like Hadoop's, reducing the opportunity for fine-grained sharing. Much like in HPC schedulers, queries must wait in a queue to run [6], and a single "monster query" can take up the entire system [11]. Reservations can be used used to avoid starving interactive queries when a batch query is running [6], but this leads to underutilization when there are no interactive queries. In contrast, our Hadoop scheduler can assign all resources to a batch job and reassign slots rapidly when interactive jobs are launched.

Fair Sharing: A plethora of fair sharing algorithms have been developed in the networking and OS domains [7, 19,

25, 30]. Many of these schedulers have been extend to the hierarchical setting [15, 20, 27, 29]. While these algorithms are sophisticated and scalable, they do not deal with data locality, as they share only one resource.

8. Conclusion

As data-intensive cluster computing systems like MapReduce and Dryad grow in popularity, there is a strong need to share clusters between users. To multiplex clusters efficiently, a scheduler must take into account both fairness and data locality. We have shown that strictly enforcing fairness leads to a loss of locality. However, it is possible to achieve nearly 100% locality by relaxing fairness slightly, using a simple algorithm called delay scheduling. We have implemented delay scheduling in HFS, a fair scheduler for Hadoop, and shown that it can improve response times for small jobs by 5x in a multi-user workload, and can double throughput in an IO-heavy workload. HFS is open source and included in Hadoop: an older version without delay scheduling is in Hadoop 0.20, and a version with all the features described in this paper will appear in Hadoop 0.21.

9. Acknowledgements

We thank the Hadoop teams at Yahoo! and Facebook for the many informative discussions that guided this work. We are also grateful to our shepherd, Jim Larus, whose input greatly improved this paper. In addition, Joe Hellerstein and Hans Zeller referred us to related work in database systems. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley RAD Lab sponsors: Sun Microsystems, Google, Microsoft, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Apache Hive. http://hadoop.apache.org/hive.
[4] Hadoop Map/Reduce tutorial. http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.
[5] Hive performance benchmarks. http://issues.apache.org/jira/browse/HIVE-396.
[6] HP Neoview Workload Management Services Guide. http://www.docs.hp.com/en/544806-001/Neoview_WMS_Guide_R2.3.pdf.
[7] Max-Min Fairness (Wikipedia). http://en.wikipedia.org/wiki/Max-min_fairness.
[8] NSF Cluster Exploratory (CluE) Program Solicitation. http://nsf.gov/pubs/2008/nsf08560/nsf08560.htm.
[9] Official Google Blog: Sorting 1PB with MapReduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
[10] Open Cirrus. http://www.opencirrus.org.
[11] Personal communication with Hans Zeller of HP.
[12] Personal communication with Owen O'Malley of the Yahoo! Hadoop team.
[13] TORQUE Resource Manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php.
[14] Yahoo! Launches New Program to Advance Open-Source Software for Internet Computing. http://research.yahoo.com/node/1879.
[15] J. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *IEEE INFOCOM'96*, pages 120–128, 1996.
[16] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *NSDI'04*, 2004.
[17] A. Chervenak, E. Deelman, M. Livny, M.-H. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi. Data Placement for Scientific Applications in Distributed Environments. In *Proc. 8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, September 2007.
[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.
[19] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Journal of Internetworking Research and Experience*, pages 3–26, 1990.
[20] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.
[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. SOSP 2003*, pages 29–43, 2003.
[22] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *Workshop on Hot Topics in Cloud Computing (HotCloud) 2009*, 2009.
[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*, pages 59–72, 2007.
[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP 2009*, 2009.
[25] J. Nieh and M. S. Lam. A SMART scheduler for multimedia applications. *ACM TOCS*, 21(2):117–163, 2003.
[26] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, 2009.
[27] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM'97*, pages 162–173, Sept. 1997.
[28] 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.
[29] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, MIT, Laboratory of Computer Science, 1995. MIT/LCS/TR-667.
[30] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. OSDI 94*, 1994.