Dynamic Proportional Share Scheduling in Hadoop

Thomas Sandholm and Kevin Lai

Social Computing Lab, Hewlett-Packard Labs, Palo Alto, CA 94304, USA
{thomas.e.sandholm,kevin.lai}@hp.com

Abstract. We present the Dynamic Priority (DP) parallel task scheduler for Hadoop. It allows users to control their allocated capacity by adjusting their spending over time. This simple mechanism allows the scheduler to make more efficient decisions about which jobs and users to prioritize and gives users the tool to optimize and customize their allocations to fit the importance and requirements of their jobs. Additionally, it gives users the incentive to scale back their jobs when demand is high, since the cost of running on a slot is then also more expensive. We envision our scheduler to be used by deadline or budget optimizing agents on behalf of users. We describe the design and implementation of the DP scheduler and experimental results. We show that our scheduler enforces service levels more accurately and also scales to more users with distinct service levels than existing schedulers.

Keywords: MapReduce, Dynamic Priority, Task Scheduling.

1 Introduction

Large compute clusters have become increasingly easier to program because of simplified parallel programming models such as MapReduce. At the same time, the costs for deploying and operating such clusters are significant enough that users have a strong incentive to share them. However, MapReduce was initially designed for small teams where resource contention can be resolved using FIFO scheduling or through social scheduling.

In this paper, we examine different task-scheduling methods for shared Hadoop (an open source implementation of MapReduce) clusters. As a result of our analysis of Hadoop scheduling, we have developed the Dynamic Priority (DP) scheduler, a novel scheduler that extends the existing FIFO and fair-share schedulers in Hadoop. This scheduler plug-in allows users to purchase and bid for capacity or quality of service levels dynamically. The capacity allotted, represented by Map and Reduce task slots, is proportional to the spending rate a user is willing to pay for a slot and inversely proportional to the aggregate spending rate of all existing users. When running a task on the alloted slot, that same spending rate is deducted from the user's budget.

This simple mechanism allows the DP scheduler to make more efficient decisions about which jobs and users to prioritize and gives users the ability to optimize and customize their allocations to fit the importance and requirements of their jobs. Additionally, it gives users the incentive to scale back their jobs when demand is high, since the cost of running on a slot is then also more expensive. We envision the DP scheduler to be used by deadline or budget optimizing agents on behalf of users. In comparison to existing schedulers, the DP implementation is simpler because it does not rely on heuristics, while still providing preemption and being work-conserving.

We present the design and implementation of the DP scheduler and experimental results. We show that our scheduler enforces service levels more accurately and also scales to more users with distinct service levels than existing schedulers. We also show how the dynamics of budgets and spending rates affect job completion time. The DP scheduler enables cost-driven scheduling across Hadoop clusters potentially operated from different sites and administrative domains.

This paper is organized as follows. In Section 2 we review the current Hadoop schedulers. We then describe the design and rationale behind our scheduler implementation in Section 3. In Section 4 and Section 5 we present and discuss a series of experiments used to evaluate our scheduler. Finally, we relate our work to previous work in Section 6 and conclude in Section 7.

2 Hadoop MapReduce

Apache Hadoop [1] is an open source version of the MapReduce parallel programming framework [2] and the Google Filesystem [3]. Historically it was developed for the same reasons Google developed their corresponding protocols, to index and analyze a huge number of Web pages. Data parallel programming or data-intensive scalable computing (DISC) [4] have since been deployed in a wide range of applications (e.g., OLAP, data mining, scientific computing, media processing, log analysis and data warehousing [5]). Hadoop runs on tens of thousands of nodes in production at Yahoo!, and Google uses their implementation heavily in a wide range of production services such as Google Earth [6].

The MapReduce model allows programmers to focus on designing the application workflow and how data are filtered and aggregated in the different stages of these workflows. The system takes care of common distributed systems tasks such as scheduling, input partitioning, failover, replication, and distributed sorting of intermediate results. The main benefits compared to other parallel programming models are the inherent data-local scheduling, and the ease of use, leading to increased developer productivity and application robustness.

In the seminal deployment at Google [2] the MapReduce architecture comprises one master and many workers. The input data is split and replicated in 64 MB blocks across the cluster. When a job executes, the input data is partitioned among parallel map tasks and assigned to slots on idle worker nodes by the master while considering data locality. Similarly, the master schedules reduce tasks on idle worker nodes that read the intermediate output from the map tasks. Between the map and the reduce phases of the execution the intermediate map data are shuffled across the reduce nodes and a distributed sort is performed. This ensures that all data with a given key are guaranteed to be redirected to the same reduce node, and in the reduce processing phase all keys are streamed in a sorted order. Re-execution of a failed task is supported where the master reschedules the task. To address the issue of a small number of tasks executing substantially slower than average and slowing down the overall job completion time, duplicate backup tasks are speculatively executed and the task that completes first is used whereas others are discarded.

2.1 Scheduling

In Hadoop all scheduling and allocation decisions are made on a task and node slot level for both the map and reduce phases. I.e., not all tasks of a job may be scheduled at once. The reason for not scheduling on a resource (node) level but on a slot level, is to allow different nodes of different capacity to offer varying numbers of slots and to increase the benefits of statistical multiplexing. The assumption is that even very complex jobs can be broken down into primitive tasks that may run in parallel on a commodity compute unit. The schedulers assume that each task in the same job takes roughly the same amount of time to complete given a slot. If this is not the case some heuristics may be applied like speculative scheduling.

All tasks are by default scheduled using a FIFO queue. Experience from large deployments at Yahoo! shows that this leads to inefficient allocations and the need for "social scheduling". The next generation scheduler in Hadoop, Hadoop on Demand (HOD), addressed this issue by setting up private MapReduce clusters on demand, managed by the Torque batch scheduling system. This approach failed in practice because it violated the data locality design of the original MapReduce scheduler, and it became too high of a maintenance burden to support and configure an additional scheduling system[1]. Creating small sub-clusters for processing individual users' tasks, as in the HOD case, violates locality because the processing nodes only cover a subset of the data nodes, and thus more data transfers are needed to stage in and out data to and from the compute nodes.

To address some of these shortcomings, Hadoop recently added a scheduling plug-in framework with two additional schedulers that extend rather than replace the original FIFO scheduler. The additional schedulers implement alternative fair-share capacity algorithms where separate queues are maintained for separate pools (groups) of users, and each are given some service guarantee over time. The inter-queue priorities are set manually by the MapReduce cluster administrator. This reduces the need for social scheduling of individual jobs but there is still a manual or social process needed to determine the initial fair distribution of priorities across pools, and once this has been set all users and groups are limited by the task importance implied by the priority of their pool. There is no way for users to optimize the usage of their granted allocation across jobs of different importance, during different job stages, or to respond to run-time anomalies such

[1] `https://cwiki.apache.org/jira/browse/HADOOP-3421`

as failures or slow nodes. The potential allocation inefficiency arising from this static setup is the main target for our work.

Previously we studied scheduling of entire virtual-machine-hosted Hadoop clusters in [7]. The general problem addressed there was how to scale up and down a set of virtual machines running Hadoop workers to complete jobs more cost-effectively and faster, based on knowledge of job workflow resource requirements. This approach works well if each user works with a separate data set. However, in case of groups of people sharing large data sets, it becomes too much of an overhead to load the data into multiple virtual clusters, and if file system clusters are shared you face the same problem as with HOD of reduced data locality. Furthermore, Hadoop is very IO intensive both for file system access and Map/Reduce scheduling, so virtualization incurs a high overhead. To address these problems we, in this work, focus on the approach of allocating slots in the Hadoop scheduler for different queues dynamically. This approach works both in a virtual and physical cluster, and it incurs less overhead when sharing the cluster among a large number of users. Next we describe our scheduler design and implementation in more detail.

3 Design

The primary design goal of our Hadoop task scheduler is to allow capacity distribution across concurrent users to change dynamically based on user preferences. Traditional priority systems that try to guess user priority are too inaccurate [8], and unregulated user priorities assume trusted small groups of users. Our scheduler automates capacity allocation and redistribution in a regulated task slot resource market.

3.1 Mechanism

The core of our design is a proportional share resource allocation mechanism that allows users to purchase or be granted a *queue priority budget*. This budget may be used to set *spending rates* denoting the willingness to pay a certain amount of the budget per Hadoop map or reduce task slot per time unit. The time unit is configurable, and referred to as *allocation interval*. It is typically set to somewhere between 10 seconds and 1 minute. In each allocation interval the scheduler:

- aggregates all spending rates s from all current users to calculate the Hadoop cluster *price*, p,
- for all users, allocates $(s_i/p) \times c$ task slots (both mappers and reducers) to user i, where s_i, is the spending rate of user i, and c is the aggregate slot capacity of the cluster,
- for all users, deducts $s_i \times u_i$ from budget b where u_i, is the number of slots used by user i

Users consuming more resources will deplete their budget faster given the same spending rate. However, they are guaranteed to not pay more than the spending rate per allocated slot. Thus a user's *bid* represents her willingness to pay a certain rate per slot.

It may appear that this model is biased towards users with small jobs who would be able to outbid users with bigger jobs. However, in the Hadoop MapReduce task model users with big jobs can effortlessly scale down their jobs to run fewer concurrent tasks and thereby consume the same amount of resources per time unit as small jobs but instead run longer. Our model thus sets the right incentives for users to scale back resource consumption as much as their job deadlines or SLAs allow.

Because we only want to charge each user for the capacity they use and reallocate the unused capacity to other users, (and we want to make sure users actually pay for the spending rate they *bid*) we calculate the capacity allocation and the price to pay for slots for an allocation interval based on the spending rates in the interval directly preceding the interval when the slots are consumed. To avoid blocking new arriving users and having non-running users hold up resources, we only calculate an allocation for a user if either a job is pending or running for that user.

To adapt more quickly to user demand fluctuations and avoid head of queue blocking and starvation issues, we support preemption where task slots that have been allocated but are no longer paid for may be reclaimed and allocated to other users. This works well for most applications since Hadoop automatically puts preempted tasks back in the pending queue to be reallocated when demand, measured by user spending rates, allows.

The key feature of this mechanism is that it discourages free-riding and gaming by users. Users who claim a higher priority will have to pay for it, so they have an incentive to accurately reveal how important priority is to them. In addition, the variable pricing allows users with a low budget and low time-sensitivity to run during low demand periods. These users would otherwise not be able to run at all in a fixed pricing model. Conversely, at high demand periods, users have a disincentive to run, but resources will nonetheless be available (for a high price) for users that really need them.

The disadvantage is less capacity predictability and more variation in capacity allocated to an application. However, the Hadoop MapReduce scheduling framework allows jobs to be split up in finer grained tasks that can run and possibly fail and recover independently. So the only thing the end users would need to worry about is to get a good enough average capacity over some time to meet their deadlines.

This introduces the difficulty of making spending rate decisions to meet the SLA and deadline requirements. It is outside the scope of this paper and the target of future work to address this particular issue, but the mechanisms presented here opens the door for innovation in this area, by allowing much more fine grained control over resources for competing users in a multi-tenancy hosted Hadoop cluster.

Figure 1 depicts how our scheduler components fit into the Hadoop architecture. Alice is willing to pay \$4 per slot, Bob is willing to pay \$1.50, and Sam \$2.

Assuming that 15 slots are available to these three users in the global (logical) slot table, Alice will be allocated 8 slots, Bob 3 slots and Sam 4 slots. Exactly how these slots are mapped to physical nodes is not guaranteed. Whenever a slot becomes available the allocations are recalculated to determine who should get the new slot according to their granted share. Furthermore, local tasks are attempted first. If that fails, remote rack tasks are scheduled. There may be opportunities to delay scheduling of some jobs to achieve a higher ratio of data local tasks. However, in the current implementation we enforce the shares strictly in each time period. This is not overly restricting because Hadoop replicates all the data in at least three data blocks by default, which ensure many opportunities for data local scheduling. Packing a user on a single node versus distributing the job workload across nodes is another application specific trade-off that we may address in future implementations.

Possible starvation of low-priority (low-spending) tasks can be mitigated by using the standard approach in Hadoop of limiting the time each task is allowed to run on a node. Moreover, our new mechanism also allows administrators to set budgets for different users and let them individually decide whether the current price of preempting running tasks is within their budget or if they should wait until the current users run out of their budget. The fact that Hadoop uses task and slot level scheduling and allocation as opposed to job level scheduling also avoids many starvation scenarios.

If there is no contention, i.e. there are enough slots available to run all tasks from all jobs submitted, the cost for excess resources essentially becomes free because of the work conserving principle of our scheduler. However, the

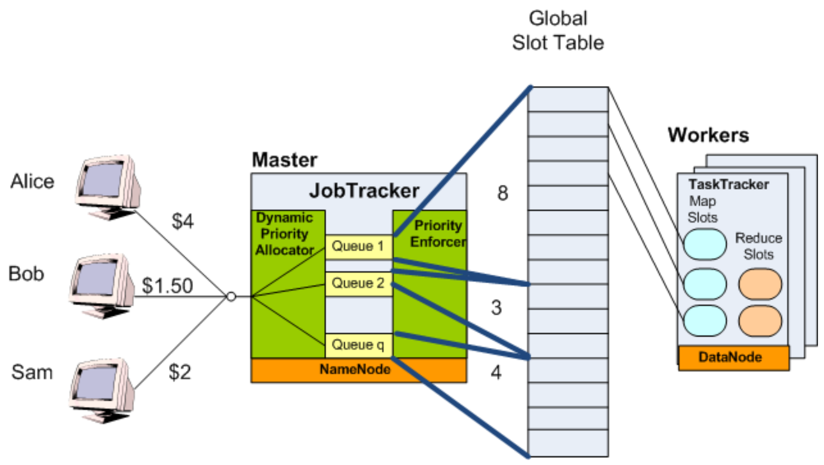


Fig. 1. Dynamic Priority Scheduler Architecture. This example shows how a max capacity of 15 Map slots gets allocated proportionally to three users. For example, Alice bids \$4 and gets $4/(4 + 1.5 + 2) * 15 = 8$ slots. The central scheduler comprises a Dynamic Priority Allocator and a Priority Enforcer component responsible for accounting and schedule enforcement respectively.

guarantees of maintaining these excess resources are reduced. To see why, consider new users deciding whether to submit jobs or not. If they see that the price is high they may wait to preempt currently running jobs, but if the resources are essentially given out for free they are likely to lay claim on as many resources they can immediately.

We note that the Dynamic Priority scheduler can easily be configured to mimic the behavior of the other schedulers. If no queues or users have any credits left the scheduler reduces to a FIFO scheduler. If all queues are configured with the same share (spending rate in our case) and the allocation interval is set to a very large value, the scheduler reduces to the behavior of the static fair-share schedulers.

3.2 Implementation

The *Dynamic Priority* scheduler is implemented as a scheduler plugin for the Hadoop JobTracker service. This allows DP to be a drop-in replacement of the default FIFO scheduler. The scheduler is split into two components: one for allocation, *Dynamic Priority Allocator*, and one for enforcement, *Priority Enforcer*.

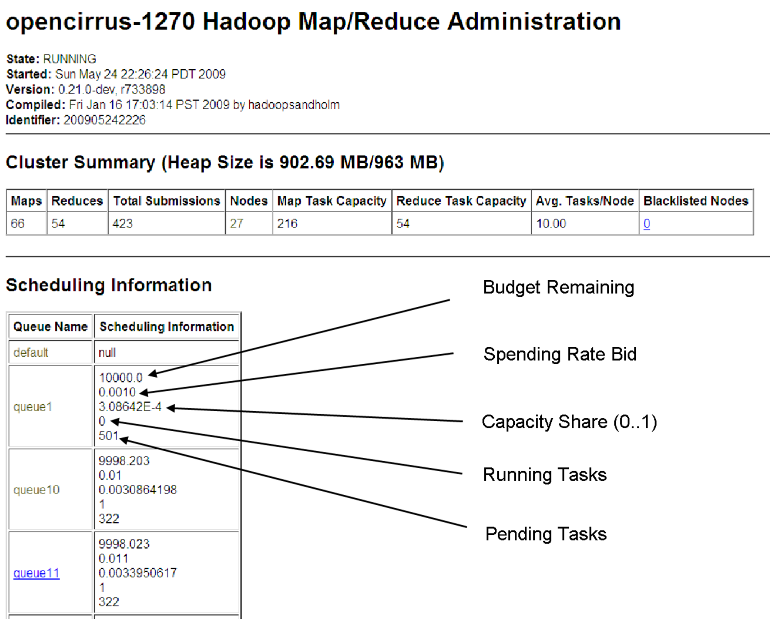
The *Dynamic Priority Allocator* implements dynamic slot allocation, budgeting and accounting, and provides a remote secure API to manage and monitor budgets and spending rates.

The *Priority Enforcer* component is responsible for enforcing the shares of resources calculated by the allocation component. It is responsible for picking pending tasks from jobs to be scheduled when mapper and reducer slots open up in Hadoop TaskTrackers. It thus implements the same functionality as the FIFO and fair-share schedulers. However, these schedulers were not designed to handle a large number of queues with constantly varying capacities that are determined on demand from user input. They do not enforce shares at the granularity and precision that our mechanism requires and do not support preemption to the extent that we require.

The budgets and spending rates are stored in a storage component that can be file-based or SQL-based. An XML REST Servlet controls the scheduler. The monitoring component plugs into the Hadoop JobTracker Web console. The Web console is depicted in Figure 2. The numbers displayed next to each queue

Table 1. REST XML API to Manage Scheduler Allocations

HTTP Options	**Description**	**Authz**
price	Gets current price	None
info=*queue*	Gets queue usage info	User
infos	Gets usage info for all queues	Admin
setSpending=*spending*&queue=*queue*	Set the spending rate for queue	User
addBudget=*budget*&queue=*queue*	Add budget to queue	Admin
addQueue=*queue*	Add queue	Admin
removeQueue=*queue*	Remove queue	Admin



opencirrus-1270 Hadoop Map/Reduce Administration

State: RUNNING
Started: Sun May 24 22:26:24 PDT 2009
Version: 0.21.0-dev, r733898
Compiled: Fri Jan 16 17:03:14 PST 2009 by hadoopsandholm
Identifier: 200905242226

Cluster Summary (Heap Size is 902.69 MB/963 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
66	54	423	27	216	54	10.00	0

Scheduling Information

Queue Name	Scheduling Information
default	null
queue1	10000.0 0.0010 3.08642E-4 0 501
queue10	9998.203 0.01 0.0030864198 1 322
queue11	9998.023 0.011 0.0033950617 1 322

Fig. 2. MapReduce Administration Monitor

represent from top to bottom: current budget, spending rate, resource share, slots used, and slots pending. The supported APIs are listed in Table 1 and an example XML response for authorized requests can be seen in Listing 1.

Listing 1. Example XML response for authorized requests

```
<QueueInfo>
  <host>myhost</host>
  <queue name="queue1">
    <budget>99972.0</budget>
    <spending>0.11</spending>
    <share>0.008979593</share>
    <used>1</used>
    <pending>43</pending>
  </queue>
</QueueInfo>
```

3.3 Security and Authentication

The existing Unix user and group based security model of Hadoop is too simple to support a full-fledged multi-tenancy resource market as described above. More specifically, relying on each user to pick queues and be trustworthy about their identity would defeat the accounting and budget enforcement mechanism. As a result, we implemented a lightweight symmetric key authentication and role-based authorization protocol modeled after AWS Query Authentication [9], and

OAuth. The advantage is that it is easy to use from any client and only requires the capability to construct HMAC/SHA1 signatures based on shared secret keys. The existing Hadoop command line clients were also extended to pass the signatures required to submit jobs to queues being paid for in job configuration parameters.

4 Evaluation

In this section, we describe experiments run to study the scalability and allocation dynamics of our scheduler. There are three sets of experiments. In the first set, we examine the correlation of spending rates, budgets and performance metrics. In the second set, we study how accurately and effectively service levels can be supported. Finally we measure how well the system adapts to changes in spending rates. Unless otherwise stated all users are given the same budgets in all experiments. We use the term *queue* interchangeably with the term *user* since all users are given a dedicated queue to submit their jobs on in all of our experiments. Our scheduler allows queues to be shared across users but it should be compared to sharing bank accounts or access to a PC account among users, i.e. sharing security credentials such as passwords, which is generally frowned upon.

4.1 Setup

We use two testbeds for our evaluation: a 30 node quad-core cluster (referred to as the *big* cluster) and a 5 node octo-core cluster (referred to as the *small* cluster). The *small* cluster runs on virtual machines, whereas the *big* cluster is installed directly on the hardware. More details of the clusters are shown in Table 2.

For both setups, we allocate one queue per user and run 2-80 users concurrently. All users run the same benchmark application, the Pi estimator from the Hadoop example code base. The Pi application was set up to be able to consume the entire cluster if run in isolation (i.e. number of job tasks were set to the number of slots available in the cluster), and thus ran slower when there was contention. The pi precision target was set to 450000000 for the small cluster and 500000000 for the big cluster to ensure that the application was both CPU and data intensive. The ability to fine-tune the CPU versus data intensity without having to provision a large amount of data was the main reason we chose the Pi application for our experiments. The fact that all Hadoop applications conform to the same general internal structure (MapReduce) allows us to treat the results more generally than with a typical parallel workload. To stress the system, all users are launched concurrently and submit a continuous stream of jobs. In the initial 2-user experiments we test the FIFO, Fairshare (fair-share scheduler developed at Facebook), and Capacity (fair-share scheduler developed at Yahoo!) schedulers and compare them to the Dynamic Priority scheduler that we developed. The Fairshare and Capacity schedulers were not able to handle the 10-80

Table 2. Experiment Cluster Setup

Cluster	Used in Graphs	Nodes	Cores (CPUs)	Physical/Virtual	OS	Disk
big	3-9	30	120(30)	Physical	CentOS 5	45TB
small	10-11	5	40(40)	Virtual	CentOS 5	250GB

queue and user workload reliably so they were excluded from the larger experiments. To switch between the schedulers during the experiment we restarted the JobTracker service resulting in a clean start since no running job information is persisted in the current version of the JobTracker. The stream of jobs from the clients is not affected either during a restart since the clients will just resubmit jobs when a job is done or fails.

4.2 Spending Rates, Budgets and Performance

In the first experiment, we start two concurrent user workloads. We give queue1 an initial budget of 1000 and queue2 10000 credits. The spending rate per Hadoop slot of queue1 is set to twice the rate of queue2. Since queue1 will then be allocated twice as many resources the total spending is expected to be 4 times that of queue2 in any allocation interval.

Figure 3 depicts the budget over time for the two users, and Figure 4 shows the completion time of their jobs over the same time period. Our scheduler is initially configured to run without preemption and queue1 will thus not see an immediate benefit in completion time.

We also see that the budget of queue1 runs out at time 05/15-22:00, at which point the allocation is given over to queue2, and the performance of queue1 degrades significantly. At time 05/16-14:00 the budgets of queue1 and queue2 are reset to 10000 and the scheduler is reconfigured to preempt. We now see that the queue1 completion time is around 3000s for each job in Figure 4 and the spending is about 26-27% more than queue2 (25% expected) as seen in Figure 3. We do not obtain exactly half the job completion time when getting twice the amount of resources but about 1.8. This is because we only control the slot capacity not other resources such as HDFS (distributed file system) IO and network bandwidth. We can also see that the higher spender (queue1) gets a very stable high performance, oscillating between 3000-3200s completion times compared to the low priority queue (queue2) which oscillates between 4500-5800s.

Now just looking at Figure 4 at time 05/18-00:00 we reconfigure the cluster to use the Capacity scheduler. The differentiation in obtained service level is far less although the capacity configuration is the same, twice as many slots for queue2. We attribute this to less aggressive preemption, and less granular control over allocations in this scheduler compared to ours. We can also see that the min/max range variation is greater for both queues with the capacity scheduler. Queue1 oscillates between 3000-3600s, and queue2 oscillates between 3600-5500s.

Taking the ratio of minimum performance to maximum performance we get a differentiation of about 1.5 to be compared to 1.8 for our scheduler. At time

05/19-00:00 we had a failed attempt to set up the Fairshare scheduler for this workload. We saw that all schedulers showed signs of a memory bloat with workload and would eventually run out of memory. This behavior was most apparent with the Fairshare scheduler which did not manage to complete a single job. We point out that this bug was not in any of the schedulers but in the jobtracker framework, so it just surfaces how different schedulers handle memory in general. So instead at time 05/19-18:00 we reconfigure the cluster with the standard FIFO scheduler. We can see that this scheduler does not offer any differentiation as expected, and the average performance level is above the queue1 level and below the queue2 level obtained with the other schedulers.

We note that the capacity scheduler was configured with 60min preemption. More frequent preemption caused problems with completing the tasks. Neither the fair-share nor the FIFO schedulers supported preemption in the versions tested[2]. However, both Capacity and Fairshare Queue/Pool capacity was configured the exact same way as with our scheduler, with the only difference that it was not able to change over time. The FIFO scheduler was not configured with any priorities, since no queue-based priorities could be set.

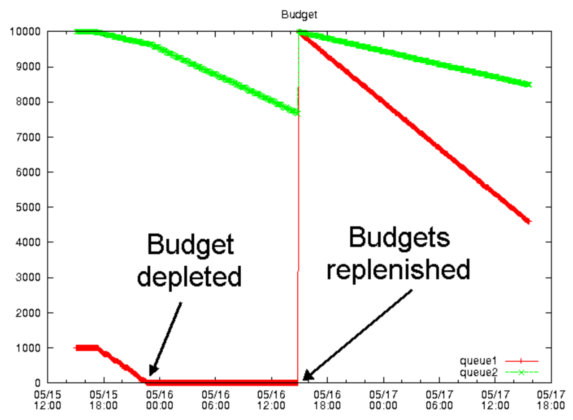


Fig. 3. *2-user budget dynamics example.* The graph shows how the budget (y-axis) evolves over time (x-axis as month/day and time). The slopes of the curves represent the spending rates of the users over time. Queue(user)1 uses twice the spending rate of queue(user)2. At the center of the graph the budgets of both users are reset to 10000 (time 05/16 14:00).

We stress that it is not simply an implementation artifact that the capacity and fair-share schedulers perform poorly in these tests. These schedulers were not designed for dynamic priorities nor for handling a large number of queues from the outset as our scheduler was[3].

[2] Hadoop 0.20-0.21 code base checked out around May 2009

[3] `http://issues.apache.org/jira/browse/HADOOP-4768`

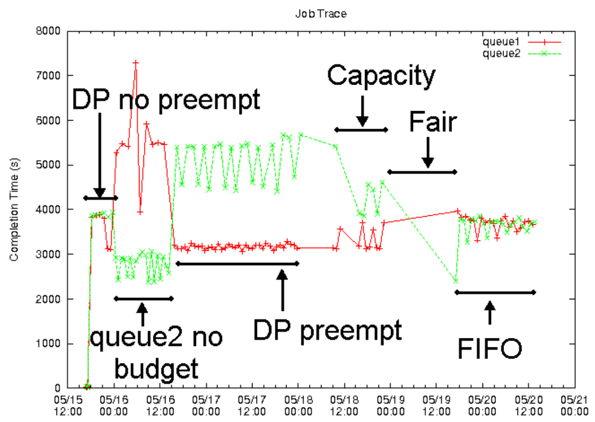


Fig. 4. *2-users service differentiation trace.* The graph shows the completion time over time for jobs submitted by the 2 users in the budget graph in Figure 3. The first half of the timeline corresponds directly to the timeline in the budget graph. The second half corresponds to experiments with the capacity, fairshare and FIFO schedulers. The first drop in completion time for queue1 is correlated with the budget running out. The key result is the clear separation of completion times between queue1 and queue2 seen in the first half compared to the second half of the graph.

4.3 Allocation Fidelity and Overhead

Now we look at how well we can preserve the differentiation of service levels with more users and queues. Figure 5 shows the completion times obtained for 10 queues when queue n is given a share of $n/\sum_{i=1}^{10} i$. We can see that all 10 service levels are enforced successfully. At time 05/21-10:00 we reconfigure the cluster with the FIFO scheduler. We note that there is a random distribution of service levels for the first job because there is no preemption. For other jobs the identical service level is given to all jobs. This experiment showcases that a dynamic non-stationary workload with users entering and leaving the system may result in random highly variable service levels even with the FIFO scheduler.

In Figure 6 we show the results of an experiment that ran our scheduler with preemption and 80 users first, then the FIFO scheduler and finally our scheduler without preemption. Still we see that the 10 service levels are maintained. We do not obtain more than 10 service levels with this application (Pi estimator). The number of service levels obtainable depends both on overhead and bottlenecks in the specific applications run but also on the overall scale of the cluster and the slots available. We also note here that the preempting version of our scheduler, in the left half of the graph, delivers somewhat more stable service level than the non-preemptive one (after time 05/24-22:00) but the differences are cosmetic. This experiment again shows that our scheduler shapes the workflow into the desired service levels quickly.

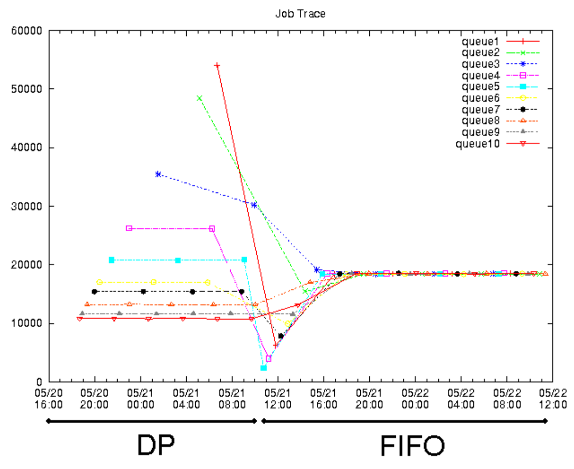


Fig. 5. *10-user service differentiation trace.* The graph shows completion time for jobs (y-axis) over time (x-axis). The first half of the graph shows how our scheduler separates the queues' performance compared to the second half when the FIFO scheduler was used. Half of the queues obtain better performance and the other half worse than the FIFO case.

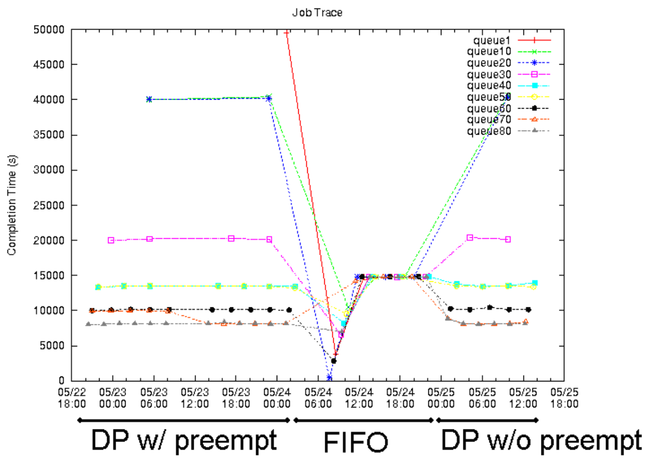


Fig. 6. *Sample of 80-user service differentiation trace.* The graph shows completion time (x-axis) over time (y-axis) using the same setup as in the 2-user graph in Figure 5, but with 80 users. For clarity only a sample of the users are shown. The results are very similar to the 2-user graph, which shows how our scheduler's ability to differentiate service levels scales well in number of queues/users.

Table 3. Distance to Ideal Line (in seconds) from Average Queue Completion Time with Approximate 95% Confidence Bounds

Scheduler	**Queue1**	**Queue2**
Capacity	1000 ± 150	600 ± 250
DynPrio	800 ± 20	300 ± 200

We now study the performance fidelity of the granted allocation more carefully. There is obviously some trade-offs in throughput of the system and the level of preemption enforced since a killed Hadoop task (note not a job) must be restarted from the beginning. Figure 7 shows the fidelity versus overhead for the two-user experiment. The ideal line depicts the performance expected if queue1 runs its jobs twice as fast as queue2, but the average across the queues is the same as for the FIFO case (e.g. optimal fidelity and maximum throughput). Our dynamic priority scheduler running with preemption comes closest to meeting this ideal, but we can also see that we can improve the throughput and move closer to the FIFO line if preemption is not turned on. Improved closeness to ideal here is seen by observing that both the queue1 point and the queue2 point in the graph for the 60s preempt dynprio line are closer to the respective ideal line points (see also Table 3).

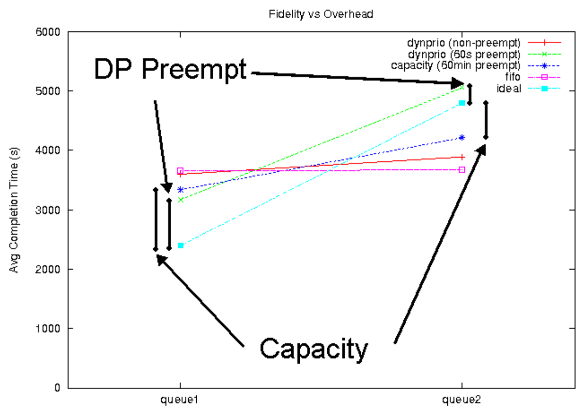


Fig. 7. *2-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling. The fairshare scheduler was not included in the results due to reliability issues. However, it behaves similarly to the capacity share scheduler. The ideal line represents the performance that should have been observed for the two queues if adhering to the configured capacities while obtaining the same throughput as with the FIFO scheduler. When comparing the slopes of the dynprio preempt line and the capacity scheduler line with the ideal line we see that the slope of the dynprio line is a closer match (one of the goals of our scheduler).

One could argue that the capacity scheduler achieves the least degradation across both users while still achieving some differentiation and should therefore be preferred. This may be the case in fair-share scheduled systems where users do not pay for their usage. But in a cloud computing scenario where queue1 actually paid twice as much as queue2 it may no longer hold true. We focus more on differentiating service-levels that are as close as possible to the capacity you pay for as opposed to achieving some overall fair outcome in our scheduler.

Figure 8 shows the corresponding graph for the 10 user experiment. We can see that the extremes (highest and lowest service levels) are far away from the ideal line whereas service levels 3 through 9 mimic the ideal scenario well. We also show an ideal adjusted line that has the same service level as the dynamic priority scheduler for the maximum service level but the same degradation in service levels as the ideal line. We can see that only service levels 1 and 2 fall outside of the ideal and ideal adjusted lines, which indicates that our scheduler is a bit biased against users with low spending rates. The same behavior can be seen in the 80-user experiment depicted in Figure 9. Here we note that the users are heavily discretized in groups of about 10-15. This is most likely due to the MapReduce workload chosen which only uses 10 reducers, and thus limits the reduce phase throughput to 10 service levels.

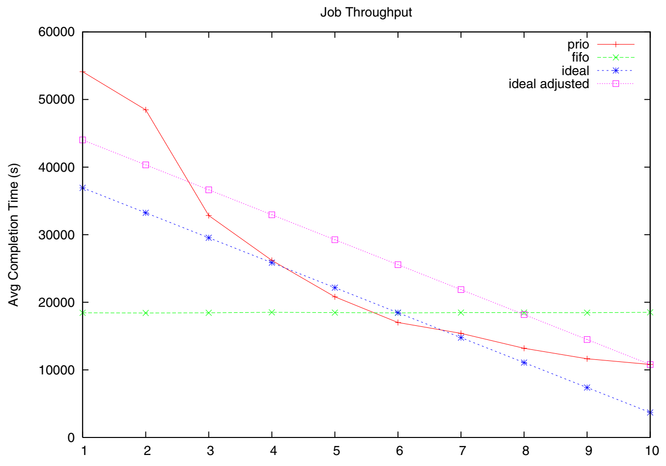


Fig. 8. *10-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling for the experiment with 10 users (user 1-10 denoted on x-axis). The ideal adjusted line corresponds to the ideal (no overhead and perfect differentiation) line with the same minimal completion time as observed in the experiments. Only users 1 and 2 (with the lowest slot capacity) deviate significantly from the ideal lines.

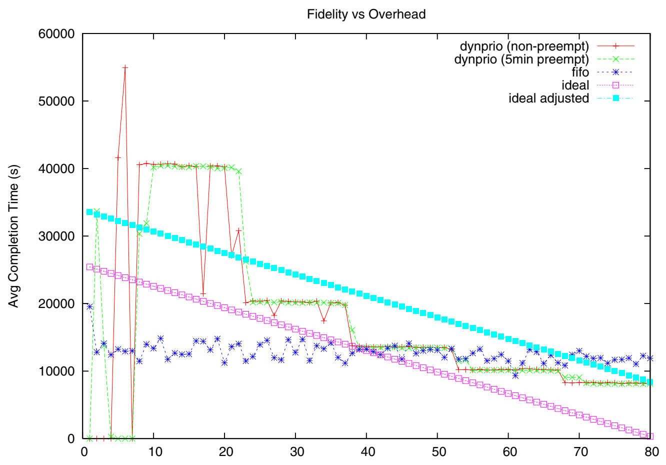


Fig. 9. *80-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling for the experiment with 80 users (user 1-80 denoted on x-axis). As in the 10-user graph the top 80 percent of the users (with highest spending rates and capacity) obtain completion times within the ideal lines (right side of graph).

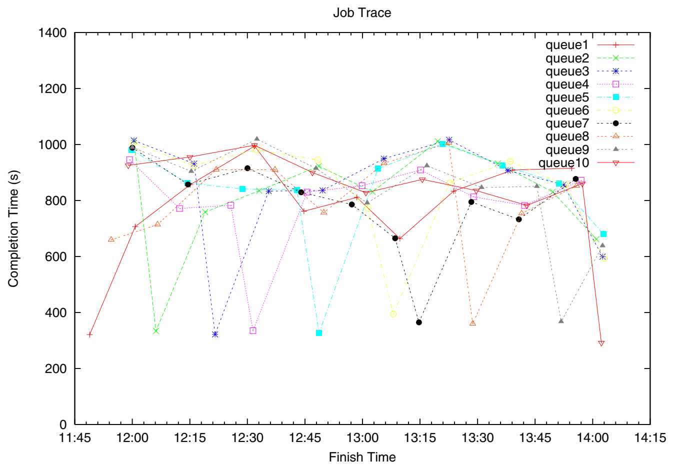


Fig. 10. *Dynamic priority adjustment with 10 users, with 60s preemption.* The graph shows the completion times of jobs for queues/users who increased their spending rates for their x'th job, where x is the queue number. All boosted jobs obtained a significant decrease in completion time, showing the agility and dynamic nature of our scheduler.

4.4 Adaptability of Service Levels

We run the final experiments on our *small* cluster and investigate how well we can dynamically adjust the service levels. 10 users all run 10 Pi estimator jobs in sequence and concurrent with all the other users. User n is given a 4x boost

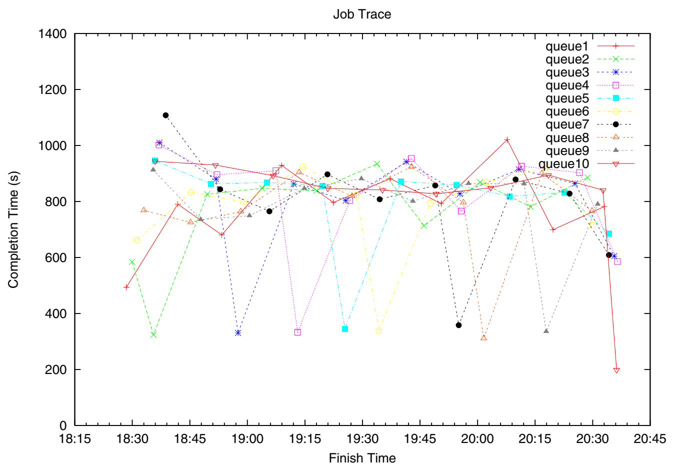


Fig. 11. *Dynamic priority adjustment with 10 users, without preemption.* The graph shows the completion times of jobs for queues/users who increased their spending rates for their x'th job, where x is the queue number when no preemption was used. The graph looks almost identical to the preemption graph with only slight deviations for the boosted jobs from user 1 and user 10.

in spending rate for job n. In Figure 10 we can see that a 3x performance boost is obtained consistently for all users and jobs regardless of when during the job sequence the boost kicks in. The valleys hover around completion times of 300s, whereas the average of non-valley jobs lies around 900s. Figure 11 shows the same experiment but with preemption turned off. We can then see that the service levels of the first jobs are random but all other jobs follow the same pattern as in the preemption case. This shows that we are able to converge quickly to a stable state even without preemption. The overhead of preemption, calculated based on the difference in average job completion time between the two experiments was less than 2.6%.

5 Discussion

Some issues merit additional discussion: preemption, dynamic adjustment, and currency management.

Whether preemption should be offered or not depends on the types of workloads expected. For CPU bound, embarrassingly parallel applications that benefit from holding a slot for a longer duration of time, preemption may be necessary to avoid starvation effects. On the contrary, for data bound applications that stream small amounts of input at a time into Map and Reduce tasks that complete within a couple of minutes, preemption may not add much value. We saw that using preemption incurred a small (2.6%) overhead in throughput, but allowed the system to adhere to service levels more quickly and accurately.

Note that preemption in the Hadoop context is somewhat different from the traditional CPU/scheduling type of preemption. Hadoop preemptions do not suspend and then resume the task but rather kills the task and forces it to start over again. It thus causes a throughput penalty. Care must hence be taken to kill the jobs that will degrade the throughput the least while still ensuring that starvation and unfairness effects are minimized.

One feature of the Dynamic Priority scheduler (DP) is that it allows users to change the priority of jobs during a run. However, it does not require it. Users who prefer not to monitor their jobs can let them run as initially configured. The opportunity to change priorities is most useful to handle unexpected situations like server failures, increases in load by other users, and the inability of users to predict their own job runtimes. In the latter case, DP allows users to adjust their spending rates so that the actual running time of their jobs fits their deadlines.

Since the DP introduces a currency into the system, it requires the system administrator to manage the overall economy of the system. The basic goal is to keep a stable exchange rate between currency and computational work. Users need to be able to expect that 1 credit will generally get 1 server hour (for example). Of course, prices will fluctuate, but the average should remain stable. Admins can do this by setting a total income rate per hour for the system which is equal to the number of servers. The admin then distributes this income among the users. For example, a cluster of 200 servers would have an income of 4800 credits per day which can be allocated for users. This total is fixed, regardless of the number of users, so the admin should reserve some amount for new users. As the admin adds new servers, the total can increase.

If prices start increasing significantly, this indicates that the system is under-provisioned with respect to its load. The admin should consider adding more servers and/or moving some users to another system. Conversely, if prices collapse, then the system is over-provisioned and the admin can add users and/or remove servers.

The admin must be careful with the inevitable demands to increase the income rate for some users. If some users actually have more important jobs than other users, then the admin should increase the income rate of the important users while decreasing the rate for other users such that the total income rate is the same. Otherwise, the system will enter an inflationary spiral that is difficult to break out of.

6 Related Work

Parallel job scheduling is a well-investigated field both in theory and in practice with applications beyond computational resource management [10]. Theoretical studies commonly assume embarrassingly-parallel jobs which has lead to much of the innovation in the field to be driven by simulations and experiments [11]. The most commonly deployed scheduling regime is First-Come-First-Served (FCFS) or variations thereof. FCFS suffers from head of queue blocking and starvation issues. Two popular variations to address these issues are backfilling [12] and gang scheduling [13] [14]. Many heuristics and variations have been proposed to improve throughput, e.g. Shortest-Job-First (SJF), or fairness, e.g. Fair-Share Scheduling. Many of these classical scheduling algorithms focus on improving systems metrics such as utilization and average response time. Some of these systems may however be very inefficient in terms of serving the most important task at the best time from an end-user point of view. The reason for this is that priorities are either assigned by the system, or are only valid across jobs for the same user, as exemplified by the Maui scheduler [15].

Proportional share and Lottery scheduling were proposed in [8] to give users more direct and dynamic control over capacity allocations for different types of tasks over time. In previous work this technique has been applied to both cluster node [16] and VM resource scheduling [17]. To our knowledge our work is the first applying the proportional share mechanism to MapReduce slot scheduling for computational clusters.

Our scheduling approach is closely related to and inspired by economic schedulers, whereby you bid for resources on a market and receive allocations based on various auction mechanisms [18,19,20,17,21,22,23,24]. We do not preclude nor require that our scheduler budgets are tied to a real currency. Furthermore, we do not assume that there are competing users who should be given different shares of the resources. Giving all users the same budget initially but allowing them to spend this budget at different rates is a valid use case of our scheduler. Many game theory inspired agent scheduling algorithms such as the Best Response algorithm in [25], could be implemented on top of our scheduler for Hadoop jobs. Meta-scheduling across Hadoop clusters in different organization is also simplified by exposing different demand-based prices for running jobs in a cluster.

Other work to improve the FIFO and fair share scheduling in Hadoop includes the LATE scheduler [26]. The main purpose of the LATE scheduler is to predict Hadoop job progress more accurately and to take overhead into account when launching speculative tasks. In [27] the work on the LATE scheduler is extended by two new techniques, delay scheduling and copy-compute splitting, designed to improve data locality and avoid reduce slot bottlenecks respectively. These techniques are complimentary to our work. In theory both of these issues are orthogonal to our scheduling mechanism since they tackle separate problems (not incentives and accountability which are at the core of our work). In practise, the delayed scheduling technique would require some changes in how slots are allocated in our scheduler, but since we only charge for slots that are actually used, the general accounting mechanism would stay the same.

MapReduce scheduling has also been explored beyond the traditional data center domain, such as for Cell [28], GPUs [29], and shared memory architectures [30]. Our general proportional share MapReduce slot algorithm presented in this paper could thus potentially also be employed in these other domains.

7 Conclusion

Our experimental results demonstrate that our scheduler scales better than the existing Hadoop schedulers in the number of queues. Having more queues allows providers to provide more service levels. The fair-share scheduler could not even handle the experimental workload for two concurrent queues, whereas the capacity scheduler was not able to handle the workload with ten queues. The Dynamic Priority scheduler handles up to 80 queues efficiently, which was only limited by the memory capacity of the experiment client node.

This enhanced scalability is due to the light-weight design of DP. In contrast to the other schedulers, it does not incur the overhead of heuristics for inferring fair priorities over time. Instead, DP users directly decide priorities, so it only has to maintain the budget currently remaining. As of this writing, the capacity scheduler contains 140KB of non-test source code, the fair-share scheduler 130KB, and DP 55KB.

Furthermore, we have shown that DP adapts service levels dynamically and quickly even during heavy load, adheres to them more accurately. This was shown by having 10 users with a stream of 10 15min jobs all boost their single high priority jobs accurately without overhead or notable randomness.

DP also solves the problems of lost data locality and virtualization overhead that we encountered in our previous work on virtualized MapReduce [7]. The downside is that we lose some control over tasks that are long-running, and the isolation properties cannot be enforced as strictly. However, an advantage is that it becomes easier to provision commonly used software and data sets in shared test-beds.

Future work includes leveraging the dynamic capacity control in our scheduler to adaptively change the allocations to meet higher level SLA goals such as deadlines.

References

1. White, T.: Hadoop: The Definitive Guide. O'Reilly, Sebastopol (2009)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Symposium on Operating System Design and Implementation (2004)
3. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. In: ACM Symposium on Operating Systems Principles (2003)
4. Bryant, R.E.: Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon University (2007)
5. `http://wiki.apache.org/hadoop/PoweredBy` (2009)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Symposium on Operating System Design and Implementation (2006)
7. Sandholm, T., Lai, K.: Mapreduce optimization using regulated dynamic prioritization. In: SIGMETRICS 2009: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 299–310. ACM, New York (2009)
8. Waldspurger, C.A.: Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. Technical Report MIT/LCS/TR-667 (1995)
9. Amazon elastic compute cloud (2008), `http://aws.amazon.com/ec2` (retrieved March 6, 2008)

10. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems, 3rd edn. Springer Science, Heidelberg (2008)
11. Frachtenberg, E., Schwiegelsohn, U.: New Challenges of Parallel Job Scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 1–23. Springer, Heidelberg (2008)
12. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, pp. 295–303. Springer, Heidelberg (1995)
13. Ousterhout, J.K.: Scheduling techniques for concurrent systems. In: 3rd International Conference on Distributed Computing Systems, pp. 22–30 (1982)
14. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling - a status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
15. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: 7th International Workshop on Job Scheduling Strategies for Parallel Processing, pp. 87–102 (2001)
16. Chun, B.N., Culler, D.E.: Market-based proportional resource sharing for clusters. Technical Report CSD-1092, University of California at Berkeley, Computer Science Division (2000)
17. Lai, K., Rasmusson, L., Adar, E., Sorkin, S., Zhang, L., Huberman, B.A.: Tycoon: an implemention of a distributed market-based resource allocation system. Multiagent and Grid Systems 1, 169–182 (2005)
18. Ernemann, C., Yahyapour, R.: Applying economic scheduling methods to grid environments. In: Grid Resource Management: State of the Art and Future Trends, pp. 491–506 (2004)
19. Piro, R.M., Guarise, A., Werbrouck, A.: An economy-based accounting infrastructure for the datagrid. In: GRID 2003: Proceedings of the 4th International Workshop on Grid Computing, Washington, DC, USA, p. 202. IEEE Computer Society, Los Alamitos (2003)
20. Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., Stornetta, W.S.: Spawn: A Distributed Computational Economy. Software Engineering 18, 103–117 (1992)
21. Chun, B.N., Culler, D.E.: User-centric performance analysis of market-based cluster batch schedulers. In: Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (2002)
22. Sandholm, T., Lai, K., Clearwater, S.: Admission control in a computational market. In: CCGrid 2008: Proceedings of the 8th International Symposium on Cluster Computing and the Grid (2008)
23. Wolski, R., Plank, J.S., Bryan, T., Brevik, J.: G-commerce: Market formulations controlling resource allocation on the computational grid. In: IPDPS 2001: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), Washington, DC, USA, p. 10046.2. IEEE Computer Society, Los Alamitos (2001)
24. Buyya, R., Murshed, M., Abramson, D., Venugopal, S.: Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimisation Algorithm. Software: Practice and Experience (SPE) Journal 35, 491–512 (2005)
25. Feldman, M., Lai, K., Zhang, L.: A price-anticipating resource allocation mechanism for distributed shared clusters. In: Proceedings of the ACM Conference on Electronic Commerce (2005)

26. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: OSDI 2008: 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
27. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, Electrical Engineering and Computer Sciences University of California at Berkeley (2009)
28. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. Parallel and Distributed Processing Symposium, International, 1–12 (2009)
29. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: PACT 2008: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 260–269. ACM, New York (2008)
30. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: HPCA 2007: IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24 (2007)