Reducing latency in cloud processing/data stores via adaptive scheduling

***Abstract*—**
Cloud computing frameworks like Hadoop and Spark, often used in multi-user environments, have struggled to achieve a balance between the full utilization of cluster resources and fairness between users. In particular, data locality becomes a concern, as enforcing fairness policies may cause poor placement of tasks in relation to the data on which they operate. To combat this, the schedulers in many frameworks use a heuristic called delay scheduling, which involves waiting for a short, constant interval for data-local task slots to become free if none are available; however, a fixed delay interval is inefficient, as the ideal time to delay varies depending on input data size, network conditions, and other factors. We propose an adaptive solution (Dynamic Delay Scheduling), which uses a simple feedback metric from finished tasks to adapt the delay scheduling interval for subsequent tasks at runtime. We present a dynamic delay implementation in Spark, and show that it outperforms a fixed delay in TPC-H benchmarks. Our preliminary experiments suggest that job latency in batch-processing scheduling can be improved further, using additional feedback metrics and adaptive techniques.

I. Introduction

Cloud computing systems like Hadoop [4], and Spark [9] have been widely used in many companies such as Google, Yahoo, Facebook, and Microsoft. Their popularity brings forth new problems in the realm of task scheduling and management in these systems, particularly with the rise of the cloud as a platform to host these frameworks.

In this paper, we explore the problem of sharing a cluster between users while preserving the efficiency of systems like Hadoop and Spark — specifically, finding a trade-off between locality (i.e., the placement of computation near its input data) and fairness. Locality is important for task (and thus job) performance, as it reduces the amount of data that needs to travel across the network, while fairness is important for clusters to be useful to multiple users at load.

Figure 1 demonstrates these tradeoffs between data locality and fairness, in a simple four-machine cluster. We consider a scenario where a ”job” consists of many smaller tasks, and each node in the cluster has a finite number of ”slots” which provide resources for tasks. Every task has input data, located in some distributed file system in blocks across the cluster. When a slot becomes available (a slot representing the resources to run a single task), a scheduler that provides fair sharing needs to schedule tasks from the job that is farthest below its fair share first. In Figure 1, the job furthest below the fair share is clearly Job 1, so a task from Job 1 will be chosen for the empty slot; however, Job 1 has no input data on Machine 3, meaning that the task will need to

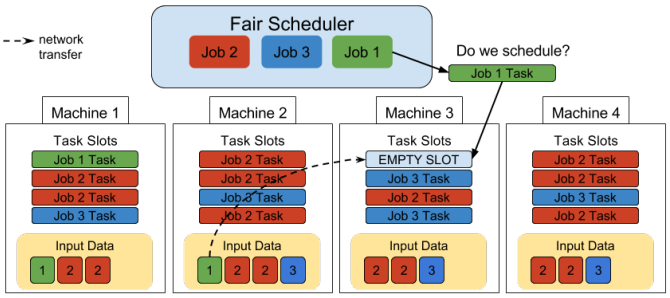

Fig. 1: A compute cluster, with a fair scheduler that must fill an empty slot

transfer a block of data over the network in order to proceed, incurring additional overhead and causing worse performance. The question becomes: what should a fair scheduler do in this situation? Should it sacrifice performance to enforce fairness, or choose another task with local data and remedy the fairness imbalance later?

The Hadoop Fair Scheduler (HFS) [7] is an influential example of a cluster framework scheduler that seeks to find this balance. The HFS was designed with two main goals: *Fair sharing*: dividing resources using max-min fair sharing [10] to achieve statistical multiplexing, and *Data locality*: placing computations (tasks) near their input data as often as possible, to maximize system throughput.

To achieve the first goal (fair sharing), a scheduler must reallocate resources between jobs when the number of jobs changes; 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, the HFS relaxes fairness slightly through a simple algorithm called *delay scheduling* [8], in which a job waits for a fixed amount of time for additional scheduling opportunities on nodes that have data for it, if none are available.

The original delay scheduling work [8] reported that a small amount of waiting is enough to bring locality close to 100% (the chosen default value is three seconds). Since its inception, delay scheduling has been used in other cluster framekwork schedulers, such as Spark, as well as in resource managers, such as YARN [5] and Mesos [2].

Delay scheduling provides a simple solution for the scenario in Figure 1. Job 1 will simply be delayed for a fixed interval (and a task from Job 3 scheduled in its place) in the hopes that data-local slots will open up in the future; however, delay

scheduling assumes that data locality is always beneficial. In reality, the decision is not as simple. The importance of data locality depends on many factors, such as network conditions, input data size, disk i/o usage, and others. These conditions may change over time (or even during the execution of a job). As well, with clusters in the public cloud becoming so common, cluster conditions can change as simultaneous user load changes, or after instances are stopped and restarted, potentially in a different physical configuration. So, the decision of whether or not to delay (and for how long) can not be efficiently solved with a fixed, unchanging delay interval. We propose that an *adaptive* mechanism is needed, in order to properly judge and exploit these tradeoffs between fairness and locality, based on feedback from the system.

In this paper, we present Dynamic Delay Scheduling, a simple, low-state, adaptive solution for task scheduling in the cloud, which improves job latency over fixed-interval delay scheduling. Dynamic Delay Scheduling operates by using feedback from finished tasks in the system, which report their perceived overhead from remote data reads. Our scheduler then adapts the delay scheduling interval to be used to schedule subsequent tasks, using a running average of the overhead metrics reported as the interval itself. In this way, our adaptive solution minimizes completion time by finding the shifting balance between: 1) immediate fairness policy enforcement, and 2) waiting for data locality, based on the severity of network overhead, for increased performance. This paper contributes an outline of this adaptive solution. We have also implemented a simple prototype in Spark. Preliminary experiments demonstrate that an adaptive delay outperforms the current Spark distribution (which uses a fixed delay scheduling interval) in TPC-H workloads.

The remainder of this paper is structured in the following way. Section II presents background on fixed delay scheduling as a concept. Section III discusses how fixed delay scheduling is implemented/mapped to real systems. Section IV defines the terminology and design of Dynamic Delay Scheduling, our simple adaptive scheduling solution. Section V describes our lightweight implementation in Apache Spark. Section VI demonstrates our evaluation of Dynamic Delay Scheduling, using our implementation in Spark under several workloads. Finally, Section VII provides closing remarks, and well as some future direction for further applying adaptive algorithms to other facets of task scheduling.

II. Delay Scheduling Background

Delay scheduling is a heuristic designed to improve the data locality of tasks in the context of fairness policy enforcement. We consider a cluster computing setting, where each application (or job), when submitted, is divided into many tasks which can be executed in parallel. The main components involved are: 1) a set of N compute nodes, each with S slots (one slot can run one task at a time), 2) a set of J jobs, each with T tasks, and 3) a set of D blocks of data, distributed across the compute nodes, where each task in a job operates on one (specific) block.

To constitute fairness, each of the J jobs must have an equal share of the cluster's resources. If there are 2 jobs, then each gets half of the cluster, if there are 3, then a third, etc. To implement this, a fair scheduler must maintain a sorted queue of the remaining jobs, based on their current share of the cluster. The job with the least running tasks (the job farthest below its fair share of resources) is at the head of this queue. Let us call this job *J_priority*. Tasks from this job will take priority over the rest in the queue.

An observation that can be made is that, when a compute slot opens up on one of the N nodes, there might not be any data blocks on the node that the tasks of *J_priority* need. This is called *data locality*. A task can be considered to be data local when it is scheduled in a slot on a node colocated with its data. Fairness enforcement compromises data locality, because the next job in the scheduler's queue (based on fairness priority) might not have data in the next open slot in the cluster. Tasks that run non-locally must pull their input data in from across the network, causing an increase in completion time.

Traditional fixed delay scheduling operates in the following way: if *J_priority* does not have any data blocks it needs in the slot currently being considered, it will be temporarily skipped, and a task from a job further down the queue (a lower priority job) will be scheduled instead. *J_priority* will remain at the head of the queue (because it is still farthest below its fair share). *J_priority* will thus be the highest priority job for the next slot that opens up, which hopefully will have data it needs. After a finite number of failed scheduling attempts (skips), *J_priority* will be scheduled in the next open slot regardless of locality. The maximum number of skips is usually determined by the number of scheduling attempts that occur within a certain interval of time (a few seconds). We can call this interval of time the *delay interval*.

The argument behind why fixed delay scheduling is acceptable is a probabilistic one. Assuming that slots free up frequently, the chance that an open slot will not overlap with a job's input data decreases exponentially. This chance is further decreased by block replication in distributed file systems (like HDFS), each of which can serve data for a task; however, this probabilistic approach relies on both a) a regular distribution of open slots across the cluster, and b) a constant slot opening frequency. If the frequency were to change (for example, if network conditions were to worsen and currently running tasks were to take longer to complete), then the chance of achieving locality during the same fixed interval decreases. As well, variance in job profiles can lead to irregular distributions of slot openings (some tasks take longer than others, some jobs have data on only a subset of machines, etc.).

III. Fixed Delay Scheduling in Cluster Frameworks

In real systems, fixed delay scheduling is usually implemented using a preset delay interval (as a unit of time, in milliseconds), and timestamps, the first of which is taken the first time a job is considered for scheduling (whether a task is actually scheduled or not). This timestamp is checked

whenever a slot opens. If the slot is on a machine without input data for a job, then tasks from that job may only be scheduled there if the time between the current time and the timestamp is greater than whatever the fixed delay interval is. If the slot *does* have input data for the job, then tasks can be scheduled no matter the timestamp. Regardless of how/why a task was scheduled for a job, once a task is scheduled then the timestamp is reset, and the process begins again. Very little extra state is required (just one additional field for each job currently in progress). For the purposes of this paper, we will be focusing on how fixed delay scheduling operates in Apache Spark (detailed below), but the principles and implementations are the same in other systems, such as Hadoop, YARN, and Mesos.

Apache Spark [9] is a general execution engine for distributed "Big Data" processing, similar to Hadoop, that breaks jobs into stages of tasks, and uses fixed delay scheduling to improve data locality when launching these tasks. Spark provides many advantages, such as an abstraction for representing distributed datasets (Resilient Distributed Datasets, or RDDs). These RDDs, which represent data either on disk (in Hadoop Distributed File System, or HDFS) or in memory, are divided into partitions across worker processes called executors in the cluster. Each executor has a finite number of slots (usually equal to the number of CPU cores available) with which to run tasks. The Spark scheduler, which resides in a centralized driver process separate from the executors, uses fixed delay scheduling to provide greater data locality when scheduling tasks into slots that become open. The fixed delay interval (the maximum waiting time) is set statically in a configuration file.

Spark (and the other systems mentioned above) also provide hierarchical fixed delay scheduling. That is, a delay interval can be specified for each node-local, rack-local, or even datacenter-local scheduling. If the delay interval expires for a job, rather than being scheduled, it will fall back into the next tier, and delay there (for perhaps a different period of time than the original interval) for slots with the relaxed locality condition of that tier (slots within the same rack as input data, for example).

IV. Dynamic Delay Scheduling

A. Beyond Constant Delays

The frameworks described above use a fixed delay interval, which remains the same throughout an entire job. Our observation is that there is a calculable tipping point after which waiting/skipping is no longer beneficial. Generally, we can define the total running time of a task t as:

$$t\_total = t\_compute + t\_netOH + sch\_delay$$

where *t_compute* is the local computation time of the task, *t_netOH* is the network latency incurred from data transfer (this is zero if the task is scheduled locally), and *sch_delay* is the amount of time t is delayed (skipped) to wait for locality, from the moment t is first considered for scheduling.

Ideally, we want to minimize the task completion time of all tasks within a given job, while also providing some guarantee as to the maximum latency introduced by the scheduler itself through delay scheduling. The goal of delay scheduling is to remove *t_netOH* by scheduling a task locally, but if the task is delayed longer than the delay interval, then it would have actually been more efficient in hindsight (in terms of task completion time) to immediately schedule the task in the first open slot that was considered. Conversely, if the delay interval is too short, then the task could potentially miss scheduling opportunities with locality, which would result in a shorter completion time than scheduling without locality.

Dynamic Delay Scheduling improves upon fixed delay scheduling by dynamically adapting the maximum *sch_delay* equal to *t_netOH* on a per-task basis. This allows for the longest waiting time for data-local slots while limiting the worst case task running time to *t_compute + (2 * t_netOH)*. In this way, the delay interval will adapt to the severity of the network overhead. If *t_netOH* is low, then locality is less of a concern and the scheduler can shorten the time it delays. If *t_netOH* becomes high, then the scheduler increases the time that it waits, as locality is more important in poorer network conditions; however, this means that the network overhead incurred by running tasks needs to be monitored and reported to the scheduler over time, in order to adapt to changing network conditions and potentially heterogeneous data block sizes.

B. An Adaptive Solution Using Task Feedback

The network overhead incurred by missing data locality depends on many factors, including network traffic, the distance of a task from its data, and the input data size. These factors vary from job to job, and some (like network traffic) can change during the job execution itself. In order to adapt the delay scheduling interval to changing conditions, we have designed a simple feedback mechanism, in which each task reports the network overhead it experienced upon completion (*t_netOH*, for a task t). This metric is sent to a centralized fair scheduler, which then changes its delay scheduling interval using a running average of feedback from completed tasks. This new delay scheduling interval is then used when scheduling subsequent tasks of the same type.

The feedback metric itself (*t_netOH*) can be any time measurement which reflects the network overhead incurred by a task. In the case of Dynamic Delay Scheduling, we are considering tasks which read data from distributed file systems, so the *t_netOH* for each task is the amount of time spent reading data blocks remotely; however, any arbitrary type of task, which has some way of measuring its own overhead, could set this as well and benefit from the adaptation.

Algorithm 1 describes our adaptive solution in detail, in a cluster with N nodes and J jobs. When tasks complete, the delay interval will be changed to reflect recent overhead measurements. The delay interval adapts relatively quickly (the average of the last few readings), since feedback can only be received when tasks complete. For workloads in

Algorithm 1: Dynamic Delay Scheduling

```
Initialization:
delay_interval ← default (3 seconds)

Upon receiving task t completion event:
delay_interval ← (delay_interval + t.netOH) / 2

Scheduling Procedure:
for each open slot s on a node n in N do
    sort J in increasing order of currently running tasks
    for j in J do
        if j has just launched a task then
            j.wait_begin = current_time
        end
        if a task t in j has data on s's node n then
            launch t in s
        else
            if a task t in j is unlaunched then
                if current_time - j.wait_begin > delay_interval then
                    launch t in slot s
                else
                    continue
                end
            end
        end
    end
end
```

which tasks complete very often, the window for the average can be broadened. During scheduling, this new interval is used for delay scheduling (and this interval can change even while a job is currently waiting, as tasks complete). It is worth noting that there is nothing preventing this adaptive solution from also being hierarchical, as many frameworks desire/implement separate delay intervals for node-local or rack-local scheduling.

V. Implementation in Spark

We have implemented Dynamic Delay Scheduling in Spark, for tasks which read data from HDFS as input. The *t_netOH* for each task is implemented as a new field in each TaskContext, called *netOH*. To compute the overhead, we have inserted a wrapper for the HDFS RecordReader, which sums reading time. When a task completes, it will set *netOH* to this sum, and then send it back to the Spark scheduler via the event bus that drives the system. While we have focused our implementation on "map-like" tasks which read from HDFS (which have predictable behavior), any arbitrary task could monitor its own overhead and set this field before closing.

The Spark Scheduler, when it sees a TaskCompleted event, will pull off that task's *netOH* and average it into a global *delay_interval* that is used for all tasks in the same "stage" (stages are groups of tasks which do the same thing). Then, when a slot opens up in the cluster, the most recent *delay_interval* will be used in place of the delay interval fetched from the spark-defaults.conf file.

Overall, the implementation consists of about 100 lines of Scala code, which modify the already-existing fixed delay scheduling mechanisms as well as store a small bit of extra state. The implementation is simple enough to be adapted to other frameworks that use delay scheduling as well, as most systems (such as Hadoop) also have events/messages that inform the scheduler with information about finished tasks.

VI. Evaluation

In order to test the efficiency of Dynamic Delay Scheduling, we have designed several tests, on both a small and a large scale, using our implementation in Spark. The goal of these experiments is to demonstrate that an adaptive delay interval provides decreased job latency in a variety of cluster setups and workloads.

Our small-scale tests were performed on a four node local cluster, with four task slots (four 1.6 GHz cores for Spark tasks) per machine. All four nodes were connected through a gigabit ethernet switch. The workload chosen for the small-scale test was an integer sort, on 1GB of input data spread across HDFS. To provide data locality concerns, input data was placed on two nodes, so as to ensure both data-local and non-local slots for task placement. Our large-scale tests were performed on a 16 node cluster on Amazon EC2, using m3.xlarge instances. The workload chosen for the large scale test was a TPC-H benchmark (#7), which performs a query reading from multiple databases in HDFS, of size ranging from several megabytes to 80 gigabytes (for a total of 120GB of input data). The benchmark is designed to simulate a business query determining the value of goods shipped between two countries. We chose this particular benchmark because it consists of a mixture of large and small stages which read from these tables in parallel. This causes contention for resources in the cluster, creating situations in which locality and fairness clash. This benchmark also represents a workload in which fairness is likely to be desired. Without fair sharing, stages which read from small tables would be significantly delayed by longer-running stages operating on larger datasets. All numbers, from both small and large scale, are the average of 3 runs, with error bars shown.

The results of our small-scale tests are shown in Figure 2. We performed the sort with the default fixed delay interval (fixed 3 seconds), with dynamic delay scheduling, and with no delay for comparison. The default of 3 seconds causes a slight increase in completion time (versus no delay). Given that network conditions were very good between the nodes in the cluster (all connected to the same switch), having a fixed delay value of 3 seconds causes increased overhead, because the network overhead of missing locality is far shorter than 3 seconds. Our adaptive solution performs slightly better, because the delay interval shrinks based on the feedback from the first round of tasks, although the job is still too small for there to be the kind of resource contention that would result in larger speedups. This experiment simply demonstrates that an incorrect fixed delay interval can actually hurt the performance of a job, rather than improve it.

The results of our large-scale TPC-H tests are shown in Figure 3, with the same three delay setups (fixed 3 seconds, our dynamic solution, and no delay). Even though the default value still improves completion time versus no delay, the dynamic solution outperforms the fixed delay by just under 15%. Since

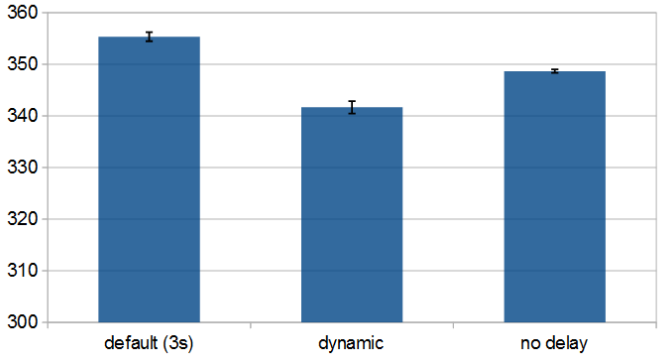

Fig. 2: 1GB Spark Sort - Job Completion Time (s) vs. Delay Type

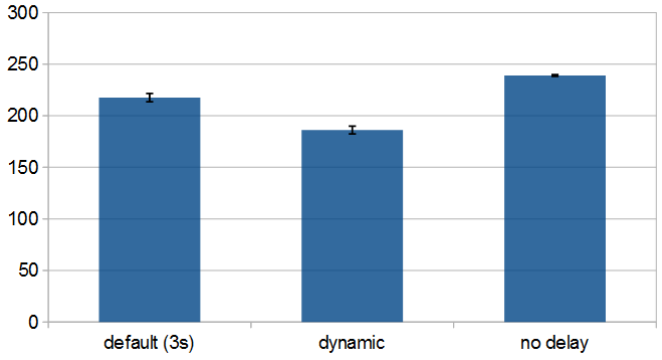

Fig. 3: 120GB Spark TPC-H Query - Job Completion Time (s) vs. Delay Type

the dynamic solution eliminates extraneous waiting while also still providing an appropriate time to wait for data locality, we see an improvement over the fixed delay. The improvement is larger than the small-scale for several reasons. First, with this test being performed in the cloud, network guarantees are not as strong, creating the need for adaptability if network conditions change. Second, with this being a large workload with hundreds of tasks, there is greater contention for slots amongst all running stages, creating more opportunity for delay scheduling to be applied.

VII. Conclusion and Related Work

Delay scheduling has long been accepted as a *de facto* solution for achieving higher data locality (and assumedly better performance) in cluster framework schedulers that enforce fairness. We show that delay scheduling, using a fixed delay interval, is inefficient, and can be improved by adapting the delay interval on-the-fly, based on the overhead experienced by completed tasks. An adaptive solution allows delay scheduling to find a proper balance between fairness and performance, based on changing network conditions, job profiles, and cluster usage. We present Dynamic Delay Scheduling as a simple proof of concept that adaptive scheduling in these systems can provide benefits with minimal state.

Regarding future work, there are other areas where adaptivity looks promising in the realm of delay scheduling. For example, the frequency with which tasks complete is major contributor to the probability that delay scheduling will actually provide locality, and could potentially be used as another adaptive improvement. Other feedback metrics (such as task completion time) hold promise for improvements/heuristics as well. Finally, avenues exist for optimally creating schedules of tasks (rather than the greedy scheduling currently used), assuming enough accurate feedback can be gained from running (or about to run) jobs efficiently.

There is much related work in the realm of cloud task scheduling. Google Borg [6] is a cluster manager that features robust fault tolerance and user isolation using containers. Borg features a centralized scheduler which assigns tasks to machines based on fine-grained resource requirements, but Borg does not consider data locality (aside from scheduling tasks on machines that already have packages/programs that are needed). Microsoft Apollo [1] is a scheduling framework which uses distributed, loosely synchronized schedulers to opportunistically schedule tasks, correcting conflicts should they arise. Apollo features a service which advertises load to schedulers to facilitate better scheduling decisions, but provides only probabilistic fairness guarantees; we hope to show that simple adaptive feedback metrics are enough to improve centralized scheduling in the presence of strict fairness enforcement. Sparrow [3] is a distributed task scheduler, which focuses on a particular set of high-frequency, interactive workloads. Sparrow features distributed, stateless scheduling and randomized load balancing of tasks; however, its completely distributed nature makes enforcing fairness policies very difficult, and its benefits wane as workloads stray from the interactive scale. Finally, some schedulers (like Hadoop YARN's [5] Fair Scheduler) have begun implementing fixed delay scheduling as a skipcount proportional to the size (in number of nodes) of the compute cluster. This approach adapts the interval to the size of the cluster, but not to the usage or load of the cluster in real time.

References

[1] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[2] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[3] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[6] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[7] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[8] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, 2010.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. pages 15—-28, 2012.

[10] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 29–42, 2008.