Preemptive ReduceTask Scheduling for Fair and Fast Job Completion

Yandong Wang[*] *Jian Tan*[†] *Weikuan Yu*[*] *Li Zhang*[†] *Xiaoqiao Meng*[†]

Auburn University[*] *IBM T.J Watson Research*[†]
{wangyd,wkyu}@auburn.edu {tanji,zhangli,xmeng}@us.ibm.com

Abstract

Hadoop MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks among data-intensive applications. However, under this scheme, Hadoop schedulers do not work effectively for both phases. We reveal that there exists a serious fairness issue among jobs of different sizes, leading to prolonged execution for small jobs, which are starving for reduce slots held by large jobs. To solve this fairness issue and ensure fast completion for all jobs, we propose the *Preemptive ReduceTask* mechanism and the *Fair Completion scheduler*. Preemptive ReduceTask is a mechanism that corrects the monopolizing behavior of long reduce tasks from large jobs. The Fair Completion Scheduler dynamically balances the execution of different jobs for fair and fast completion. Experimental results with a diverse collection of benchmarks and tests demonstrate that these techniques together speed up the average job execution by as much as 39.7%, and improve fairness by up to 66.7%.

1 Introduction

MapReduce [10] is a simple yet powerful programming model that is increasingly deployed at many data centers for the analysis of large volumes of unstructured data. Hadoop [1] is an open-source implementation of MapReduce. It divides a MapReduce job into two types of tasks, map tasks (MapTasks) and reduce tasks (ReduceTasks), and assigns tasks to multiple workers called TaskTrackers for parallel data processing.

To support many users and jobs (large batch jobs and small interactive queries), Hadoop MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks for data-intensive applications. The Hadoop Fair Scheduler (HFS) [4] and Hadoop Capacity Scheduler (HCS) [3] have focused on fairness among MapTasks. These schedulers strive to maximize the use of system capacity and ensure fairness among different jobs. However, they do not work effectively for both phases. What complicates the matter is the distinct execution behaviors of MapTasks and ReduceTasks. Unlike MapTasks which are launched one group after the other to process data splits, ReduceTasks have a different execution pattern. Once a ReduceTask is launched, it occupies the reduce slot until completion or failure.

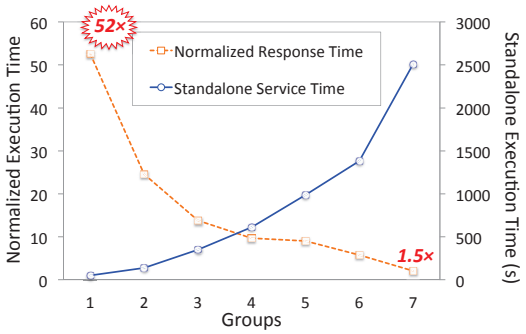

Figure 1: Unfair Execution among Different Size Jobs

We have examined the performance of Hadoop schedulers using a synthetic workload of jobs submitted to a shared MapReduce cluster. Jobs are divided into 7 groups based on their increasing data sizes; jobs in the same group are identical. They arrive according to a Poisson random process. Figure 1 shows the comparison of the *normalized execution time*, which is defined as the ratio between a job's actual execution time and its standalone execution time (the time when a job is running in the system alone). As shown in the figure, the standalone execution time of jobs in each group increases in proportion to their input data size. However, the completion of these jobs varies dramatically with HFS. Jobs in the smaller groups have much worse normalized execution times, indicating that they must wait very long (as much as 52× longer than the stand-alone execution time). Such scheduling behavior contradicts users' intuitive expectation that smaller jobs should be completed faster and turned around more quickly.

To address this fairness issue and ensure fast completion for jobs of various sizes, we design a combination of two techniques: the *Preemptive ReduceTask*

mechanism and the *Fair Completion Scheduler*. Preemptive ReduceTask is a solution to correct the monopolizing behavior of long ReduceTasks. By enabling a lightweight working-conserving option to preempt ReduceTasks, Preemptive ReduceTask offers a mechanism to dynamically change the allocation of reduce slots. On top of this preemptive mechanism, the Fair Completion Scheduler is designed to allocate and balance the reduce slots among jobs of different sizes. In summary, we make the following contributions on the scheduling of jobs in data centers for fair and fast job completion.

- We examine the unfairness issue of MapReduce jobs execution in detail and identify the key shortcomings of existing schedulers in balancing the allocation of reduce slots among jobs.
- We introduce the Preemptive ReduceTask mechanism for lightweight, work-conserving preemption, on top of which we design the Fair Completion Scheduler that improves both the fairness and execution of MapReduce jobs.
- We have conducted a systematic evaluation of Fair Completion Scheduler. Our results demonstrate that it can reduce the average execution time of workloads by up to 39.7% and improves the fairness by as much as 66.7%, when compared to HFS.

2 Background and Motivation

In this section, we first provide a brief overview of Hadoop job scheduling, then discuss the issues within existing schedulers.

2.1 Job Scheduling in Hadoop

In Hadoop, the JobTracker assigns available map and reduce slots separately to jobs in the queue, one slot per task. Figure 2 shows an example of scheduling three jobs (represented by shaped blocks in three colors) on a system with three reduce slots and five map slots. The scheduling policy is based on the Hadoop Fair Scheduler. A job when running alone can satisfy its needs with all reduce slots, but it has to share the slots when other jobs arrive. Once granted a slot, a ReduceTask has to fetch data produced by all MapTasks before it completes. In the figure, Job 1 first arrives by itself. It grabs 3 map slots and 2 reduce slots for itself and completes execution. Job 2 then takes the rest of map and reduce slots. When Job 2 needs more map or reduce slots, it has to share, because Job 3 has arrived.

Each map output file has a partition for every ReduceTask, the current Hadoop scheduler greedily launches as many ReduceTasks as permitted for each job to maximize the chance of overlapping the shuffling of available intermediate data with the execution of future MapTasks. Hadoop also allows a configuration parameter

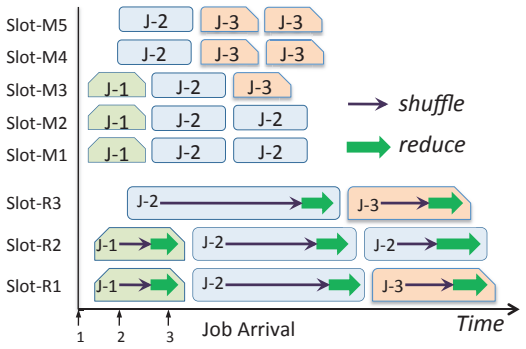

Figure 2: An Example of Scheduling Slots among Jobs

(*slowstart.completed.maps*) to delay the launch of ReduceTasks so that small jobs after large jobs can have chances to share the reduce slots.

2.2 Profiling of Unfair Slot Allocation

To closely examine the fairness issue between different jobs, we conduct an experiment on a cluster of 20 nodes. 40 map slots are created on 10 nodes, and 20 reduce slots on the other 10 nodes. 8 jobs are sequentially submitted into the cluster every 60 seconds. Job 3 is a large job that requires 20 ReduceTasks. Figure 3 shows the usage of map and reduce slots by 8 jobs. Map slots are shared among jobs over time as jobs arrive and leave, but reduce slots are all occupied by Job 3. As a result, Jobs 4-8 cannot get a share until Job 3 completes, even if they have successfully finished all their MapTasks. On average, Jobs 4-8 are significantly delayed compared to their stand-alone execution times. This reveals that Hadoop Fair Scheduler is not able to achieve fair normalized execution times for all jobs. A similar behavior was also reported by an IBM study [15]. Note that there exists a dramatic variance among the normalized execution time for different jobs in the same pool and in different tests (c.f. Figure 1 and Figure 3). More importantly, when the generation rate of intermediate data is low, even if long running ReduceTasks are occupying the slots, they do not efficiently utilize the resources, and ReduceTasks periodically enter into the idle state, causing severe resource underutilization. In this experiment, on average, during 87.6% of Job 3's ReduceTasks execution time, CPUs and disks are idle and waiting for the intermediate data, and network is highly underutilized.

2.3 Proposed Solutions

The monopolizing behavior of ReduceTasks has been documented earlier as a reason to cause small jobs starve for reduce slots [17, 15, 18]. Hadoop provides a slowstart configuration option that can delay the launch of ReduceTasks and mitigate this situation, but at the cost of slowing down the shuffle phase, thus it can significantly prolong the execution times of small jobs. Zaharia *et al.* [18]

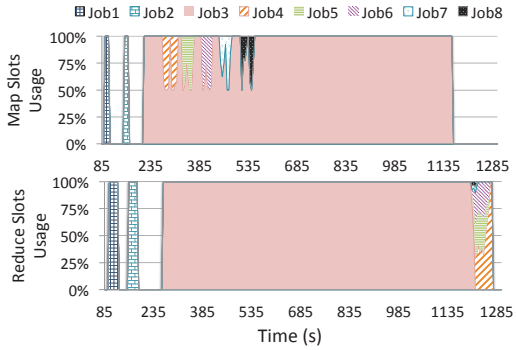

Figure 3: Run-Time Allocation Profile of Map and Reduce Slots.

proposed a copy-compute splitting mechanism, but it does not fully resolve this issue. Tan *et al.* [15] proposed a coupling scheduler to launch reducers gradually by coupling the progresses of map and reduce tasks in the same job. With this scheduler, a large job can spare reduce slots for other jobs when its map phase has not progressed much. But when a large job finishes its map phase, it still takes all available reduce slots and causes the starvation of small jobs. Like the slowstart option, the coupling scheduler delays and mitigates the monopolization of reduce slots by large jobs. But it does not solve the monopolization, instead let it progressively happen.

In this study, we examine the job fairness and efficiency issues in data centers and investigate the feasibility of lightweight task preemption and automated preemptive scheduling policy for fair and fast job completion under MapReduce context. Two techniques are designed accordingly to tackle these issues: the Preemptive ReduceTask mechanism and the Fair Completion Scheduler. Preemptive ReduceTask allows ReduceTasks to be preempted in a work-conserving manner (without losing previous I/O or computation work, or causing high overhead) during shuffle or reduce phases. The Fair Completion Scheduler builds on top of preemptive ReduceTask to automatically monitor job progresses and dynamically balance the usage of reduce slots, thereby speeding up the execution of small jobs and ensuring fairness among a large number of jobs on a shared Hadoop cluster.

3 Preemptive ReduceTask

A preemptive mechanism needs to be efficient and lightweight so that it can react fast enough to dynamic system workloads. But a ReduceTask often consumes the bulk of processing time due to its main responsibilities of fetching and merging intermediate data from all MapTasks and performing user-defined reduce computation on the merged data. In this section, we introduce our Preemptive ReduceTask mechanism that can preempt a ReduceTask at any time during its execution, with low overhead and negligible delay to the job progress.

3.1 Work-Conserving Self Preemption

Preemption is usually an OS utility to threads and processes running on a system. Operating systems such as Linux are equipped with a sophisticated thread/process table along with virtual memory to record the progresses of threads/processes and support lightweight preemption. However, there is no such utility in Hadoop to keep the ReduceTask around as a process after its preemption. Although Hadoop currently provides a *killing* based preemption mechanism, our results show that killing is a poor preemption option that can significantly delay the progress of entire job. A naive checkpoint/start mechanism is also not suitable because it dumps all memory of a ReduceTask (it can be several GB) to persistent storage and incurs very high costs. Instead we introduce a work-conserving self preemption mechanism. When requested, a ReduceTask will conserve its work and then preempt itself, i.e., exit and release reduce slot. Note that our preemptive ReduceTask keeps current APIs of Hadoop and HDFS [14] intact, all existing Hadoop applications can still function without any modification.

During the shuffle phase, a ReduceTask fetches all the segments that belong to it from all intermediate map outputs. According to the sizes of the segments, ReduceTask stores them either to local disks or in memory. Meanwhile, multiple merging threads merge fetched segments into larger segments and store them to the persistent storage. During the reduce phase, a ReduceTask organizes all the segments in a *Minimum Priority Queue* (MPQ, which has a heap structure), in which the segment that has the minimum first <key,value> pair is positioned at the head of MPQ. As the reduce phase progresses, <key,value> pairs are continually popped out from the MPQ and supplied to the reduce function.

3.1.1 Preemption during Shuffle Phase

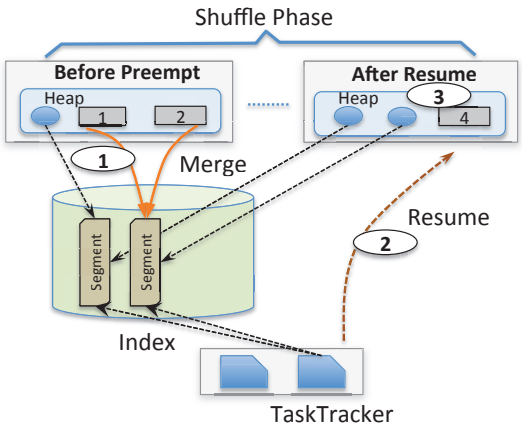

Figure 4: Preemption during Shuffle Phase

Figure 4 shows our design of work-conserving preemption when a ReduceTask is in the shuffle phase. Be-

fore preemption, a ReduceTask has a mixture of one on-disk segment and two in-memory segments, organized in a heap. Preserving the state of shuffle phase is to keep track of the shuffling status of all segments. Upon receiving a preemption request, this ReduceTask merges the in-memory segments and flushes the results to the disks (Step 1) while leaves on-disk segments untouched. The parent TaskTracker maintains an index record on the locations of fetched segments, one per preempted ReduceTask. Then the ReduceTask preempts itself and releases the slot. When the ReduceTask is later resumed (Step 2), it retrieves the index record from the parent TaskTracker, then restores the heap structure before the preemption. After that, this ReduceTask continues to fetch the rest segments from remaining map outputs (Step 3).

3.1.2 Preemption during Reduce Phase

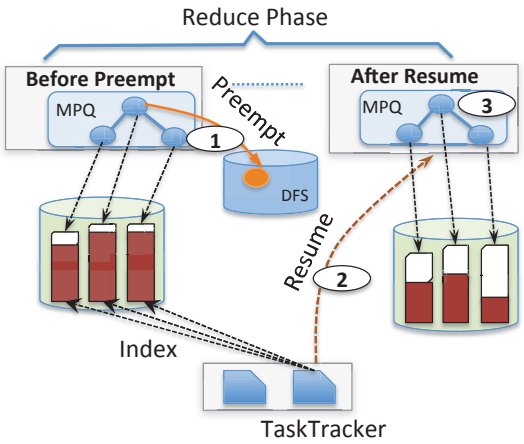

Figure 5: Preemption during Shuffle Phase

To conserve the work before preemption in the reduce phase, a ReduceTask needs to store the current results to HDFS besides recording the positions of input segments in the MPQ. In other words, ReduceTask needs to preserve the state of reduce computation at the end of each intermediate <key,val> pair, and remember the index of the last intermediate <key,val> pair at the time of preemption. Figure 5 shows our strategy for work-conserving preemption during the reduce phase. A ReduceTask is drawing <key,val> pairs from the MPQ that consists of three segments. When it receives a preemption request, it stops the reduce computation at the boundaries of <key,val> pairs (Step 1). Available results for previous <key,val> pairs are stored to HDFS. The parent TaskTracker again helps in this process by storing an index record for a preempted ReduceTask, and later provides it for preempted ReduceTask to resume its execution (Step 2). After resumption, the ReduceTask restores the MPQ again and proceeds further from the next <key,val> pair without any loss or repetition of reduce computation and intermediate data reshuffling (Step 3). During this process, to allow multiple preempted/resumed ReduceTasks to write to HDFS, we let the TaskTracker maintain the output streams to HDFS, therefore they can be shared by many ReduceTasks. Only the last ReduceTask closes the stream. In addition, Task migration is also possible for a preempted ReduceTask but it requires data to be re-fetched over the network.

4 Fair Completion Scheduler

Algorithm 1 FCS: Selecting ReduceTask to Preempt

1: $L_{running}$: {a list of running jobs of decreasing remaining work.}
2: J_i: {a job requesting new reduce slots.}
3: Demand(J_i) $\leftarrow$ {J_i's demand for reduce slots.}
4: **if** *Available_reduce_slots* $<$ Demand(J_i) **then**
5: $m \leftarrow$ Demand(J_i) - *Available_reduce_slots*
6: **for all** $j \in L_{running} \wedge$ IsPreemptable(J_j) $\wedge (m > 0)$ **do**
7: **if** $(J_j.T_{rs} > J_i.T_{rs}) \vee ((J_j.T_{rs} == J_i.T_{rs}) \wedge (J_j.R_{left} > J_i.R_{left})$ **then**
8: $RL_n \leftarrow$ {J_j's list of running ReduceTasks}
9: **for all** $r \in RL_n \wedge (m > 0)$ **do**
10: preempt r
11: $m \leftarrow m - 1$
12: **end for**
13: **end if**
14: **end for**
15: **end if**

To efficiently balance the reduce slots among a large number of jobs of different sizes, we introduce a novel preemptive ReduceTask scheduling policy based on the remaining ReduceTasks workload of all the jobs. MapTasks are scheduled under independent scheduling policies, such as max-min fair sharing, or FLEX [17]. Because of its benefits in achieving fairness for jobs of different sizes (c.f. Section 5), we refer to it as Fair Completion Scheduler (FCS).

As a preemptive scheduler, FCS must be equipped with two algorithms: one to automatically select a ReduceTask to preempt and the other to select a ReduceTask to launch. We first describe the selection policy for preemption. To select a suitable ReduceTask and achieve fair execution, we need to evaluate the run-time progress of jobs. However, the relative progress and the remaining processing time of ReduceTasks are not available before they start. We choose the following approximations to estimate the progress.

Remaining shuffle time: This is estimated as T_{rs} through the function: $T_{rs} = (\frac{M_{left}}{M_{rate}}) \times T_{mavg}$, where M_{left} stands for the number of remaining MapTasks, M_{rate} is the average rate in completing MapTasks, and T_{mavg} is

the average execution time of MapTasks that have completed or in progress. As a job makes progress in its execution, we dynamically update M_{rate} accordingly.

Remaining reduce data: This is estimated as R_{left} through the function: $R_{left} = R_{total} - R_{done}$, where R_{total} stands for the total intermediate data to reduce, and R_{done} the data that has been reduced. The latter is available during the progress of reduce phase, and the former is available when the reduce phase starts.

Execution Slackness: This is estimated as E_{slack} through the function: $E_{slack} = \frac{T_{total}}{T_{est}}$, where T_{total} is a ReduceTask's total execution time since its beginning and T_{est} is its estimated execution time based on its progress without preemption. We calculate it as $T_{est} = \frac{T_{svc}}{C_{pctg}}$, where T_{svc} is the actual execution time excluding preemption and C_{pctg} is the percentage of completed work.

FCS is designed with policies to balance reduce slots between small jobs and large jobs. It compares a job j that has the largest amount of remaining work to a job i requesting reduce slots, as shown in Line 7 of Algorithm 1. Job j's ReduceTasks are preempted if it has more work than Job i (Line 10). Essentially, this allows small jobs to preempt large jobs, solving the monopolizing behavior of long-running jobs and reducing the delay of small jobs. On the other hand, we monitor the *execution slackness* of a ReduceTask since its beginning. If its execution slackness has reached a configurable upper-bound (5 by default), a ReduceTask will not be preemptable, i.e. $IsPreemptable$ returns false. This enables large jobs with an option to escape preemption–keeping their reduce slots–and avoid starvation. Note that the execution slackness is a calculated number at run-time, which offers a better choice than a static parameter, for example, the number of times a ReduceTask can be preempted. Its sole purpose is to guarantee that a long job would not get seriously delayed because of frequent preemptions by other jobs. Besides taking into account of execution slackness, we avoid preempting a newly launched ReduceTask or a ReduceTask whose progress has gone over 70% to avoid overhead.

Then we describe briefly the policy for selecting a ReduceTask to launch, which is shown as Algorithm 2. In making this selection, FCS favors the jobs with the least amount of remaining work as shown in Line 2 of Algorithm 2. Jobs are firstly sorted according to their T_{rs} values, when two T_{rs} values are equal, they are sorted according to R_{left}. In addition, it takes the data locality into account, trying to launch a preempted ReduceTask on the same node that it has executed before (Line 4). A preempted ReduceTask that cannot achieve data locality will be delayed (Line 16). However, if a preempted ReduceTask has been delayed for too long because it is not able to resume on its previous node (Line 11), then FCS migrates it to another node that has available reduce

Algorithm 2 FCS: Selecting ReduceTask to Launch

```
1:  {Receiving a heartbeat from node n with an empty slot.}
2:  L_rem: {a sorted list of jobs of increasing remaining work.}
3:  for all j ∈ L_rem do
4:    if (Task r is j's reduce task either preempted from n or never launched) then
5:      r.migration = 0
6:      launch r on n
7:      return
8:    end if
9:    T_prt ← {j's preempted ReduceTasks (oldest first)}
10:   for all r ∈ T_prt do
11:     if r.migration >D then
12:       migrate r to n
13:       r.migration = 0
14:       return
15:     end if
16:     r.migration += 1
17:   end for
18: end for
```

slots (Line 12). In this algorithm, D is an approximation of $-M \times ln(\frac{1-L}{1+(1-L)})$, a similar parameter employed in the delay scheduling [19], where M is the number of nodes in the cluster and L is the expected data locality. For example, on a cluster of 20 nodes, with the expected data locality $L = 0.95$, then $D \approx 61$. With this algorithm, we fit the same delay scheduling policy (and its parameter D) nicely into FCS, and delay the launching of a ReduceTask for a future possibility to resume it on the node it was preempted, i.e., better locality. This parameter allows us to consider the tradeoff between the need of resuming ReduceTask for data locality and the need of migrating ReduceTasks for free slot utilization. In Section 5.2.1, we show that careful tunning of D can indeed lead to a good tradeoff between these two factors.

5 Evaluation Results

This section presents a systematic performance evaluation of Fair Completion scheduler (FCS) using a diverse sets of workloads, including *Map-heavy* workload, *Reduce-heavy* workload. Furthermore, we conduct stress tests through *Gridmix2* [2]. We compare the performance of FCS to the Hadoop Fair Scheduler (HFS) and Hadoop Capacity Scheduler (HCS). Several versions of Hadoop are available. Particularly, YARN as a successor of Hadoop provides a new framework for task management. However, through code examination and perform evaluation, we have found that YARN adopts the same task schedulers, thus facing the same fairness issues as

Hadoop. In addition, YARN is still not yet ready for large-scale stable execution. Therefore, our evaluation is based on the stable version Hadoop 1.0.4.

5.1 Experimental Environment

Cluster Setup: Experiments are conducted in a cluster of 46 nodes. One node is dedicated as both the NameNode of HDFS and the JobTracker of the Hadoop. Each node is equipped with four 2.67GHz hex-core Intel Xeon X5650 CPUs, 24GB memory, and two 500GB Western Digital SATA hard drives.

Hadoop Setup: We configure 8 map slots and 4 reduce slots per node, based on the number of cores and memory available on each node. We assign 1024MB heap memory to each map and reduce task, respectively. The HDFS block size is set to suggested 128MB [19] to balance the parallelism and performance for MapTasks.

Benchmarks: We employ the well-known *GridMix2* and *Tarazu* benchmarks [6] to demonstrate that FCS is suitable for various types of workloads.

Tarazu benchmarks represent typical jobs in production clusters. Meanwhile, Different benchmarks emphasize different workload characteristics. Map-heavy jobs generate a small amount of intermediate data, thus resulting in lighter ReduceTasks compared to the relatively heavier MapTasks. This group includes *Wordcount*, *TermVector*, *InvertedIndex* and *Kmeans*. On the other hand, Reduce-heavy jobs generate a large amount of intermediate data, thus causing heavy network shuffling and reduce computation at the ReduceTasks. This group includes *TeraSort*, *SelfJoin*, *SequenceCount*, and *RankedInvertedIndex*. it is worth mentioning that we configure the submission of GridMix2 jobs as a Poisson random process with a configurable arriving interval.

Evaluation Metrics: A number of performance metrics used in our presentation are listed as follows.

- ***Average execution time***: This is the plain average of execution time among a group of jobs, reflecting the efficiency of schedulers to a system.
- ***Maximum slowdown***: We refer to *slowdown* as the normalized execution time, which is defined earlier. *Maximum slowdown* is then the biggest slowdown among a group of jobs. This reflects the fairness to jobs of different characteristics.
- ***ReduceTask wait time***: It is defined as the time spent by a ReduceTask in waiting for reduce slots after the same job's MapTasks (i.e. the entire map phase) have all completed. If the ReduceTask gets a slot before that, then the wait time is 0. This aims to reflect the delay experienced by ReduceTasks.
- ***Average preemption times***: This is the average number of preemptions experienced by a group of jobs with similar job sizes. This quantifies the distribution and frequency of preemptions to jobs of different groups that differ in job sizes.

5.2 Evaluating Design Choices of FCS

The design of FCS includes a couple of important design choices such as the threshold parameter that allows task migration to resume a preempted ReduceTask, and the choice of Preemptive ReduceTask instead of killing as the preemption mechanism. In this section, we conduct tests to evaluate these design choices and elaborate their importance.

5.2.1 Opportunistic ReduceTask Migration

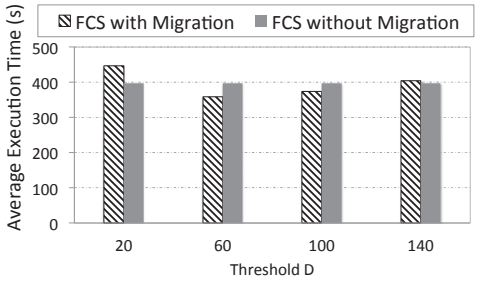

Figure 6: Effectiveness of ReduceTask Migration

As mentioned in section 4, FCS is designed with an opportunistic parameter *D* that controls the tradeoff between keeping ReduceTasks on their original node for data locality and migration ReduceTasks to other available slots for resource utilization. A very large *D* allows a ReduceTask to be delayed many times and become sticky to their original nodes, achieving better data locality for the resumed ReduceTask but at the cost of underutilization of other reduce slots. In contrast, a very small *D* leads to better resource utilization but also incurs more data movement. In this section, we assess the impact of *D* by executing a pool of Gridmix2 jobs. Also, job submission is configured to follow a Poisson random process with an average inter-arrival time of 30 seconds.

In the experiment, we increase *D* from 20 to 140, and compare the performance results of FCS with migration to that of FCS without migration. As shown in Figure 6, FCS with migration can lead to the best average execution time when *D* equals 60, with an improvement of 9.6%. Neither a small D of 20 or a large D of 140 can achieve a good balance between data locality and resource utilization. This experiment confirms that opportunistic task migration as controlled by *D* can lead to good system performance. In the rest of the section, we use 60 as the value for *D*.

5.2.2 Benefits of Preemptive ReduceTask

We investigate the efficiency of FCS when preemption is enabled with either the Preemptive ReduceTask or the killing-based approach. We use three GridMix2 workloads of different numbers of jobs (80 for Workload-

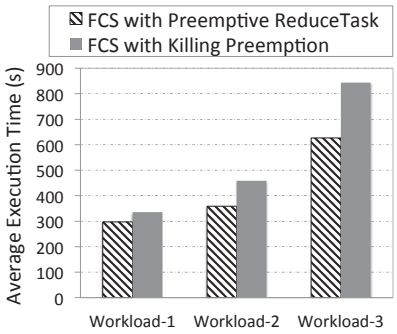

Figure 7: Benefit of Preemptive ReduceTask

1, 130 for Workload-2 and 180 for Workload-3). Figure 7 shows the results. Compared to FCS with killing-based preemption, FCS with Preemptive ReduceTask effectively reduces the average execution time by 11.3%, 21.8% and 25.7% for three workloads, respectively. This demonstrates that FCS performs more efficiently with Preemptive ReduceTask than with the killing approach. In the rest of this paper, we focus on further evaluation of FCS with the Preemptive ReduceTask mechanism.

5.3 Results for Map-heavy Workload

Table 1: Job Composition of Map-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	WordCount	10	1	50
2	TermVector	20	2	40
3	InvertedIndex	50	4	30
4	TermVector	100	8	20
5	Kmeans	500	10	10
6	TermVector	1000	20	8
7	Kmeans	5000	20	6
8	InvertedIndex	10000	60	4
9	TermVector	15000	120	2
10	InvertedIndex	20000	180	1
			Total Jobs	171

Table 2: Performance of Map-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	247	359	1061

We now present the evaluation results on Map-heavy workload. The workload composition is shown in Table 1, featuring two basic characteristics. First, as shown in empirical trace studies [9, 12], realistic workloads exhibit a heavy-tailed distribution for job sizes, accordingly the number of MapTasks among jobs. Second, to capture the effect that jobs arrive to the MapReduce cluster according to a random process, i.e., their arrival interval follows a Poisson random process with an average inter-arrival time of 30 seconds. For ease of presentation, we sort the jobs according to their input sizes and the requested number of tasks, then divide them into 10

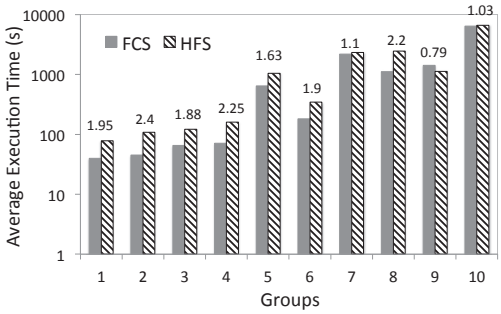

Figure 8: Average Execution Times of Jobs in Different Groups of Map-heavy Workload

different groups of increasing sizes. This categorization helps the interpretation of scheduling effects on jobs of different sizes.

Table 2 shows the average execution time for all jobs in Map-heavy workload with different schedulers. Both FCS and HFS significantly outperform HCS, which groups jobs into a small number of job queues, within each queue, HCS adopts FIFO scheduling policy that is known to bias against small jobs and cause long average execution times. Thus we focus on the comparisons between FCS with HFS in the rest performance tests on Map-heavy workload. Overall, FCS speeds up the average execution time by 31% compared to HFS.

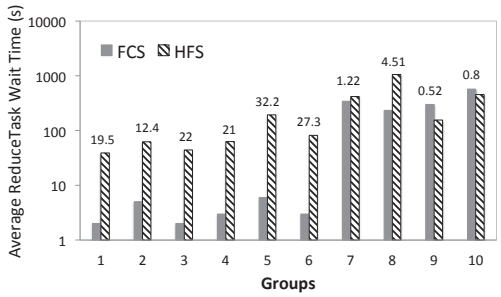

Figure 9: Average ReduceTask Wait Times of Jobs in Different Groups of Map-heavy Workload.

To shed light on how FCS treats jobs of different sizes, we examine the average execution times for the 10 different job groups inside workload. Figure 8 shows that FCS effectively reduces the average execution time for the first 8 groups compared to HFS, achieving up to 2.4× speedup for jobs in group 2. Only jobs in Group 9 are negatively affected by FCS, at an average ratio of 0.79. Such performance results match the design goal of FCS, i.e., trading long running large jobs for fast completion of small jobs.

FCS improves system performance by mitigating the starvation of small jobs. It prioritizes jobs whose shuffle phases are about to complete, thus reducing the ReduceTask wait times. Figure 9 shows the average Reduc-

eTask wait time for all jobs in 10 groups. As we can see, the average wait time is dramatically cut down for the first eight groups by as much as 32.2× for group 5. Only for the last two groups, the wait times are stretched slightly, indicating that FCS yields reduce slots for the small jobs.

To further obtain insights on how FCS has triggered preemptions to different jobs, we record the preemptions experienced by all ReduceTasks. Figure 10 shows the distribution of preemptions to different groups of jobs in the workload. As shown in the figure, no preemptions have happened to Groups 1-4. Groups 5-10 have experienced a small number of preemptions. This demonstrates that FCS can be effective in delivering fair and fast completion without imposing excessive preemptions.

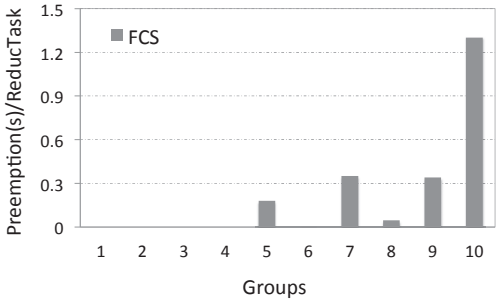

Figure 10: Preemption Frequency

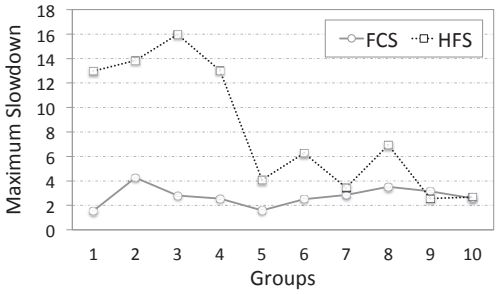

Figure 11: Fairness of Map-heavy Workload

We have measured the maximum slowdown of all jobs to evaluate the fairness of schedulers to different jobs. As shown in Figure 11, FCS efficiently improves the fairness by up to 66.7%, compared to HFS, and achieves nearly uniform maximum slowdown across 10 groups. In contrast, HFS causes serious unfairness to small jobs. In the worse case, a job in Group 3 is slowed down by as much as 16 times.

Taken together, these results confirm the benefits we expect from the design of FCS. They adequately demonstrate the strengths of FCS for Map-heavy workload.

5.4 Results for Reduce-heavy Workload

Map-heavy workload represents jobs that generate small amount of intermediate data. In this section, we continue our evaluation with Reduce-heavy workload, in which jobs generate a large amount of intermediate data, resulting in long running ReduceTasks. The ratio of intermediate data size to input size of those jobs is from 1 : 1 to 3 : 1. The job composition in the workload is listed in Table 3. We adopt the same distributions for job sizes and their arrival times as described in section 5.3.

Table 3: Job Composition of Reduce-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	TeraSort	10	2	50
2	SelfJoin	20	4	40
3	SequenceCount	50	8	30
4	TeraSort	100	16	20
5	SelfJoin	500	32	10
6	RankInvertedIdx	1000	64	8
7	TeraSort	5000	128	6
8	SequenceCount	10000	256	4
9	TeraSort	15000	512	2
10	SequenceCount	20000	1024	1
			Total Jobs	171

Table 4: Performance of Reduce-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	978	1364	8829

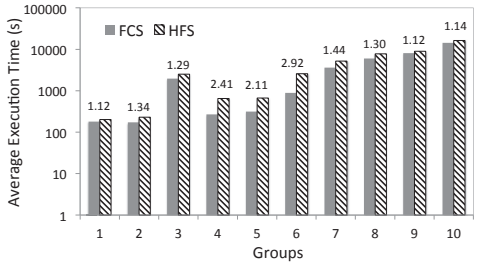

Figure 12: Average Execution Times of Jobs in Different Groups of Reduce-heavy Workload

We conduct the same set of experiments for Reduce-heavy workload as done for the Map-heavy workload to demonstrate that FCS can schedule different workloads effectively. Many results exhibit similar performance to those in Map-heavy workload. Thus, for succinctness, we avoid redundant description, omit some figures, and only highlight the differences. Table 4 shows the overall performance under three schedulers. FCS speeds up the average execution time of the workload by 28% when compared to the HFS, and HCS still performs worse than the other two.

Figure 12 illustrates that FCS speeds up the average execution times of 10 different groups in the workload. This differs from the Map-heavy workload, in which 8 out of 10 groups achieves obvious acceleration. In addition, we observe that FCS improves the completion rate not only for small and medium jobs, but effectively

for the large jobs in the last three groups as well. This is because in Reduce-heavy workload, map phases of large jobs run much longer to generate intermediate data than their counterparts in Map-heavy workload. When small jobs arrive, they preempt long running ReduceTasks from jobs that have not progressed much in the reduce phase. As a result, such preemptions impose little performance impact on the execution of these ReduceTasks in large jobs, because those long running ReduceTasks periodically enter into the idle mode to wait for the availability of intermediate data. On the contrary, these preemptions are greatly beneficial to small jobs that can efficiently utilize the reduce slots to accelerate their execution. Moreover, as small jobs quickly leave the cluster, resource contention is gradually ameliorated. Therefore, long running large jobs obtain resources during the reduce phases and achieve faster job completion.

Similar to the Map-heavy workload, significantly shortened ReduceTask wait time contributes to the fast job completion of Reduce-heavy workload. Figure 13 compares the average ReduceTask wait times between FCS and HFS. For Groups 9 and 10, FCS leads to a slightly longer delay, up to 15%. For Groups 3 and 8, FCS and HFS are comparable. Group 1 has zero wait time in both cases. FCS drastically reduces the wait time down to 0 for Groups 2 and 4, and effectively cuts down the average ReduceTask wait time for Groups 5,6 and 7, ranging from 9.8× to 84.5×. Interestingly, even FCS delays the launch of long running ReduceTasks in Group 9 and 10, their job execution is not affected. In such scenario, more intermediate data is buffered. Once launched, ReduceTasks spend more of their execution on data fetching. Faster completion of all jobs in all groups directly leads to better fairness. Figure 14 shows that FCS efficiently improves the fairness by 35.2% on average when compared to the HFS for the Reduce-heavy workload. Note that FCS still maintains low preemption frequency for different groups of jobs, in particular for large jobs. Because it bears strong resemblance to that of Map-heavy workload, we omit the preemption frequency result here.

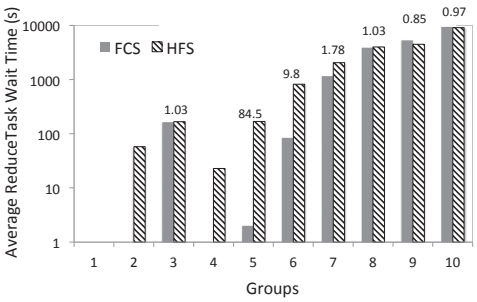

Figure 13: Average ReduceTask Wait Times of Jobs in Different Groups of Reduce-heavy Workload

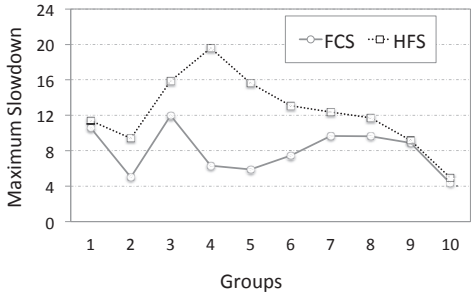

Figure 14: Fairness of Reduce-heavy Workload

5.5 Scalability

The workloads submitted to a production cluster vary substantially over different periods of time. Thus, the capability of efficiently scheduling a large number of randomly arriving jobs is critical for Hadoop schedulers, especially when the system is heavily loaded. To investigate the scalability of FCS, we employ Gridmix to assess the performance of Hadoop when the system is under stress. We vary the number of GridMix jobs from 60 to 300 and maintain the same distribution of job size throughout different tests.

The experimental results are shown in Figure 15. Compared to HFS, FCS consistently reduces the average execution times across different experiments. On average, FCS reduces the average execution time by 39.7%. More importantly, FCS shows stable performance improvement when the number of jobs in the workload increases. The improvement ratio rises from 10% to 28% when the number of jobs increases from 60 to 300. In the workload with 60 jobs, small jobs are dominant with very few large jobs arriving very late. In such a scenario, the demands for reduce slots from small small jobs can be satisfied in time, leading to shortened ReduceTasks waiting time. As a result, it leaves less optimization spaces for FCS to improve. Furthermore, during the tests, when the number of jobs increases, no noticeable scheduling overhead in terms of CPU utilization is observed in the JobTracker.

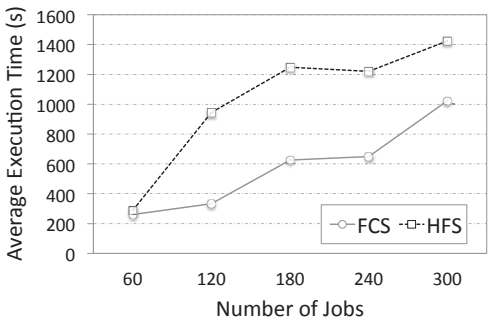

Figure 15: Scalability Evaluation with GridMix2

6 Related Work

Many MapReduce schedulers have been proposed over the past few years trying to maximize the resource utilization in the shared MapReduce clusters. Zaharia *et al.* introduced delay scheduling [19] that speculatively postpones the scheduling of the head-of-line tasks and ameliorate the locality degradation in the default Hadoop Fair scheduler [4]. In addition, Zaharia also proposed Longest Approximate Time to End (LATE) [20] scheduling policy to mitigate the deficiency of Hadoop scheduler in coping with the heterogeneity across virtual machines in a cloud environment. But neither of these two scheduling policies supports task preemption for jobs in the same pool, thus unable to correct the monopolizing behavior of long-running ReduceTasks.

Mantri [7] was designed to mitigate the impact of outliers in MapReduce cluster, it monitors task execution with real-time remaining work estimation, and accordingly take measures such as restarting outliers, placing tasks with network awareness and conserving valuable work from the tasks. But Mantri does not identify the resource monopolizing issue among large number of concurrent jobs caused by long-running ReduceTasks and does not provide lightweight preemption solution. Ahmad [6] proposed communication-aware placement and scheduling of MapTasks and predictive load-balancing for ReduceTasks as part of Tarazu to reduce the network traffic of Hadoop on heterogeneous clusters. But it also does not address the fairness and monopolization issues. Isard *et al.* [11] introduced the Quincy scheduler, which adopts min-cost flow algorithm to achieve a balance between fairness and data locality for the Dryad. But their use of killing as preemption mechanism can cause significant resource waste.

Verma [16] introduced ARIA to allocate appropriate amount of resources to MapReduce job so that it can meet SLO. Based on ARIA, Zhang *et al.* [21] further studied the estimation of required resources for completing a Pig program to meet SLO. Lama [13] proposed AROMA to automatically determine the system configuration for Hadoop jobs to achieve quality of service goal. FLEX [17] aims to optimize different given scheduling metrics based on a performance model between slots and job execution time. However, none of above four work considers the resource contention issue (reduce slot contention) among continuously incoming jobs in shared MapReduce clusters.

In [8], Ananthanarayanan proposed Amoeba which supports lightweight elastic tasks that can release the slots without losing previous I/O and computation. This bears strong similarity to our preemptive ReduceTask. However, it imposes many constraints such as safe points on task processing so that tasks can be interfered without losing previous work. However no overhead measurement is reported in the article. In addition, no corresponding scheduling policy is designed to leverage the benefits provided by elastic task.

Recently, YARN [5] has been proposed by Yahoo! as the next generation MapReduce. It separates the JobTracker into ResourceManager and ApplicationManager, and removes task slot concept. Instead, it adopts resource container concept that encapsulates the general resources, such as memory, CPU and disk I/O into the schedulable unit (current YARN only supports memory). But our initial evaluation discovers that monopolization behavior of long-running ReduceTasks still exist in such framework as long as schedulers greedily allocate as many resources as permitted to one job. Therefore, our Preemptive ReduceTasks and Fair Completion Scheduler can be very beneficial in the new framework. In future, we plan to incorporate our techniques into the YARN.

7 Conclusion

In this paper, we have revealed that there exists a serious fairness issue for the current MapReduce schedulers due to the lack of a lightweight preemption mechanism for ReduceTasks. Accordingly, we have designed and implemented the Preemptive ReduceTask as a work-conserving preemption mechanism, on top of which we have designed the Fair Completion Scheduler. The introduction of the new preemption mechanism and the novel ReduceTask scheduling policy have solved the fairness issue to small jobs, resulting in improved resource utilization and fast average job completion for all jobs. Our design of Fair Completion Scheduler, compared to the Hadoop Fair Scheduler and Capacity Scheduler, can reduce the average job execution time by up to 39.7% and 88.9%, respectively. Furthermore, the Fair Completion Scheduler improves the fairness among different jobs by up to 66.7%, compared to the Hadoop Fair Scheduler.

Acknowledgments

This work is funded in part by a National Science Foundation award CNS-1059376 and by an Alabama Governor's Innovation Award.

References

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Gridmix2. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[3] Hadoop Capacity Scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.

[4] Hadoop Fair Scheduler. http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html.

[5] Next Generation Hadoop MapReduce. http://hadoop.apache.org/docs/current/index.html.

[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 61–74, New York, NY, USA, 2012. ACM.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceeding OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, Canada, October, 2010. ACM.

[8] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 24:1–24:7, New York, NY, USA, 2012. ACM.

[9] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive query processing in big data systems: A cross industry study of mapreduce workloads. Technical Report UCB/EECS-2012-37, EECS Department, University of California, Berkeley, Apr 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Sixth Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, Dec. 2004.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[12] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010.

[13] P. Lama and X. Zhou. Aroma: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[15] J. Tan, X. Meng, and L. Zhang. Delay tails in mapreduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, New York, NY, USA, 2012. ACM.

[16] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011.

[17] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. balmin. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[21] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.