Practical Size-based Scheduling for MapReduce Workloads

Mario Pastorelli
EURECOM
Campus SophiaTech, France
pastorel@eurecom.fr

Antonio Barbuzzi
EURECOM
Campus SophiaTech, France
barbuzzi@eurecom.fr

Damiano Carra
University of Verona
Italy
damiano.carra@univr.it

Matteo Dell'Amico
EURECOM
Campus SophiaTech, France
dellamic@eurecom.fr

Pietro Michiardi
EURECOM
Campus SophiaTech, France
michiard@eurecom.fr

ABSTRACT

We present the Hadoop Fair Sojourn Protocol (HFSP) scheduler, which implements a size-based scheduling discipline for Hadoop. The benefits of size-based scheduling disciplines are well recognized in a variety of contexts (computer networks, operating systems, etc...), yet, their practical implementation for a system such as Hadoop raises a number of important challenges.

With HFSP, which is available as an open-source project, we address issues related to job size estimation, resource management and study the effects of a variety of preemption strategies. Although the architecture underlying HFSP is suitable for any size-based scheduling discipline, in this work we revisit and extend the Fair Sojourn Protocol, which solves problems related to job starvation that affect FIFO, Processor Sharing and a range of size-based disciplines.

Our experiments, in which we compare HFSP to standard Hadoop schedulers, pinpoint at a significant decrease in average job sojourn times – a metric that accounts for the total time a job spends in the system, including waiting and serving times – for realistic workloads that we generate according to production traces available in literature.

1. INTRODUCTION

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [17], has created the need to organize and manage the resources of clusters of computers that operate in a shared, multi-tenant environment. For example, within the same company, many users *share* the same cluster because this avoids redundancy (both in physical deployments and in data storage) and may represent enormous cost savings. Initially designed for few and very large batch processing jobs, data-intensive scalable computing frameworks such as MapReduce are nowadays used by many companies for production, recurrent and even experimental data analysis jobs. This is substantiated by recent studies [14, 28] that analyze a variety of production-level workloads (both in the industry and in academia): an important characteristic that emerges from such works is that there exists a stringent need for *interactivity.* The number of small jobs might be dominant in current workloads: these are preliminary data analysis tasks involving a human in the loop, which for example seeks at tuning algorithm parameters with a trial-and-error process, or even small jobs that are part of orchestration frameworks whose goal is to launch other jobs according to a workflow schedule.

In this work, we study the problem of job *scheduling*, that is how to allocate the resources of a cluster to a number of concurrent jobs submitted by the users, and focus on the open-source implementation of MapReduce, namely Hadoop [6]. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several alternatives [37, 12, 20, 23, 30, 36] have been proposed to enhance scheduling: in general, existing approaches aim at two key objectives, namely *fairness* among jobs and *performance* in terms of job execution time.

We propose to use job *sojourn time* as a performance metric, which accounts for the time a job spends in the system waiting to be served and its execution time. We thus proceed with the design and implementation of a new scheduling protocol that caters both to a fair and efficient utilization of cluster resources, while striving to achieve short sojourn times. As such, our approach satisfies *both* the interactivity requirements of small jobs, and the performance requirements of large batch jobs.

Our solution, Hadoop Fair Sojourn Protocol (HFSP), belongs to the category of size-based, preemptive scheduling disciplines. In addition to addressing the problem of scheduling jobs characterized by a complex structure in a multi-processor system, we propose an efficient method to implement size-based scheduling when job size cannot be known *a priori.*

HFSP allocates cluster resources such that job size information is inferred while the job makes progress toward its completion. The scheduling discipline benefits from preemption to achieve short job sojourn times; however, preemption is not currently implemented in Hadoop. As such, we introduce a new set of primitives that enables HFSP to interrupt and eventually resume running jobs, and show in which cases this approach is superior to the widely adopted technique of killing running tasks to make room for other jobs. The contribution of our work can be summarized as follows:

- We design and implement the system architecture of HFSP, including a (pluggable) component to estimate job sizes, a dynamic resource allocation mechanism that strives at efficient cluster utilization and a new set of low-level primitives that allows preemptive disciplines. HFSP is available as an open-source project.[1]

- HFSP uses a new scheduling discipline inspired by the Fair Sojourn Protocol [19]. Job scheduling operates in a multi-processor context and aims to minimize job sojourn times. One of the main consequences of the HFSP discipline is that small jobs, for which "interactivity" is important, do not wait for a long time before being awarded cluster resources. The HFSP scheduler is also beneficial to medium-large jobs which are granted a large fraction of cluster resources.

- We perform an extensive experiment campaign, where we compare the HFSP scheduler to the two main schedulers used in production-level Hadoop deployments, namely the FIFO and the FAIR schedulers. For the experiments, we use state-of-the-art workload suite generators that take as input realistic workload traces. Our results – that we obtain on a large cluster deployed in Amazon EC2 – show that the average job sojourn time achieved with HFSP is drastically reduced with respect to the other scheduler we examined. In addition, we show results that substantiate the claim of an efficient cluster resource utilization under heavy loads.

The remainder of the paper is organized as follows: in Section 2 we provide background information on a set of scheduling disciplines and on some details of Hadoop MapReduce. In Section 3 we describe in details the HFSP schedulers and its inner components. We evaluate the performance of our job scheduler in Section 4 and provide in Section 5 additional considerations. In Section 6 we discuss the related work, and we conclude in Section 7.

2. BACKGROUND

When comparing different scheduling disciplines, there are different performance metrics one can consider. In this work we focus on the *mean response time* – *i.e.* the total time spent in the system, given by the waiting and service time, called also *sojourn time* – for each job, and *fairness*. Next, we consider two disciplines that are relevant in our context: one that minimizes the mean response time and one that provides perfect fairness.

The optimal preemptive scheduling policy that minimizes the mean response time is the Shortest Remaining Processing Time (SRPT), where the job in service is the one with the smallest remaining processing time – this policy requires the job size to be known *a priori*. SRPT provides no guarantees on system fairness: as such, long jobs may starve. As opposed to minimizing the mean response time, the Processor Sharing[2] (PS) discipline is conceived to guarantee a fair share of system resources to be dedicated to each job: if N jobs need to be served, with PS each receives a $1/N$th fraction of the system resources. However, the mean response time achieved by PS is higher than that obtained with SRPT.

Friedman and Henderson [19] study a scheduling policy for a single-server queue model that strives to obtain both (near) optimal mean response times and fairness, called Fair Sojourn Protocol (FSP). Since our work is inspired by FSP, in the following we provide sufficient background to understand its properties.

2.1 How FSP Works

The main idea of FSP is to run jobs in series rather than concurrently. FSP computes the completion time for each job under a virtual PS discipline. The order at which jobs complete in PS is used as a reference to schedule jobs in series. In the basic single server-queue model, this means that at most one job is served at a time, and that such job may be preempted by a newly arrived job. An example is the best way to illustrate how FSP works.

Assume that there are three jobs, j_1, j_2 and j_3, each requiring *all the resources* available in the system. Such jobs arrive at time $t_1 = 0$s, $t_2 = 10$s and $t_3 = 15$s respectively; it takes 30 seconds to process job j_1, 10 seconds to process job j_2 and 10 seconds to process job j_3 (if all the resources are used, otherwise the time increases inversely proportionally to the available resources).

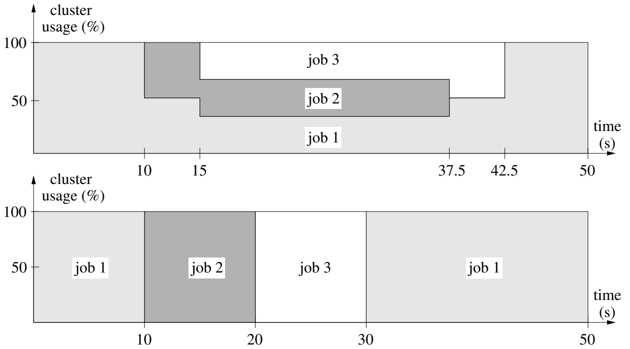


Figure 1: Comparison between PS (top) and FSP (bottom).

Figure 1 (top) represents the system utilization over time under the PS discipline: when job j_2 arrives, the server is shared between j_1 and j_2, and, when job j_3 arrives, the server is shared among the three jobs. The job completion order is j_2, j_3 and j_1. The bottom part of the figure shows how the workload described above is scheduled under the FSP discipline. When job j_2 arrives, since it would finish before job j_1 in case of PS, it preempts job j_1. When job j_3 arrives, it does not preempt job j_2, since it would finish after it in case of PS; when job j_2 finishes, job j_3 is scheduled since it would finish before job j_1 in case of PS. The FSP discipline ensures each job receives a fair amount of system resources, as when PS scheduling is used. At the same time, under FSP, the mean job completion time is considerably smaller than under PS. Next, using a simple example, we anticipate a more elaborate setup that underlies our work, whereas in Section 3.1 we detail all the hidden intricacies of a multi-processor version of FSP.

[1] `https://bitbucket.org/bigfootproject/hfsp-project`

[2] In this work, we compare our proposal to the FAIR scheduler, which implements the concept of PS, albeit with additional features to cope with a multi-processor system.

Assume that jobs j_1, j_2 and j_3 require 100%, 55% and 35% of the system resources respectively. The arrival times are $t_1 = 0$s, $t_2 = 10$s and $t_3 = 13$s and the processing time (if the required share of system resources is given to each job) is 30 seconds for job j_1, 10 seconds for job j_2 and 10 seconds for job j_3.

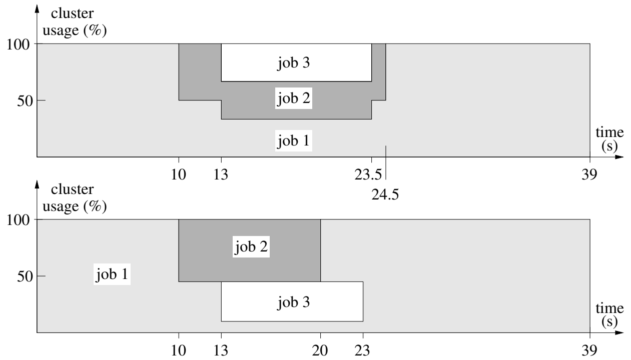


Figure 2: Comparison between PS (top) and an ideal multi-processor FSP (bottom), with jobs that do not require the full cluster.

Figure 2 compares PS (top) to an ideal, *multi-processor* version of FSP (bottom). With ideal FSP, job j_2 would preempt job j_1; since j_2 requires only 55% of the system, the remaining 45% can still be used by j_1. When job j_3 arrives, it would preempt job j_1 (but not job j_2), but it is sufficient to allocate 35% of the system to serve it, leaving 10% of the server to job j_1. As shown in the Figure, the mean job completion time under multi-processor FSP is smaller than that achieved by PS, and system resources are allocated such that no job is "mistreated."

2.2 Hadoop MapReduce

MapReduce, popularized by Google with their work in [17] and by Hadoop [6], is both a programming model and an execution framework. In MapReduce, a job consists of three phases and accepts as input a dataset, appropriately partitioned and stored in a distributed file system. In the first phase, called MAP, a user-defined function is applied in parallel to input partitions to produce intermediate data stored on the local file system of each machine of the cluster; intermediate data is sorted and partitioned when written to disk. Afterwards, a REDUCE phase is scheduled. It comprises a SHUFFLE sub-phase, where intermediate data is "routed" to the *reducers*. Subsequently, intermediate data from multiple mappers is sorted and aggregated to produce output data which is written back on the distributed file system.

Hadoop Scheduling. In Hadoop, the JOBTRACKER takes care of coordinating TASKTRACKER nodes, which can be thought of as worker machines. The scheduler, which is the subject of this work, resides in the JOBTRACKER. The role of the scheduler in MapReduce is to allocate TASKTRACKER resources to running tasks: MAP and REDUCE tasks are granted independent *slots* on each machine. The number of MAP and REDUCE slots on each TASKTRACKER is a configurable parameter, which depends on the cluster in which Hadoop is deployed, and on the characteristics (e.g., the number of CPU cores) of each server in the cluster.

When a *single* job is submitted to the cluster, the scheduler simply assigns as many MAP tasks as the number of available slots in the cluster. Note that the total number of MAP tasks is equal to the number of partitions of the input data. The scheduler tries to assign MAP tasks to slots available on machines in which the underlying storage layer holds the input intended to be processed, a concept called *data locality*. Also, the scheduler may need to wait for a portion of MAP tasks to finish before scheduling subsequent mappers, that is, the MAP phase may execute in multiple "waves", especially when processing very large data. Similarly, REDUCE tasks are scheduled once intermediate data, output from mappers, is available.[3] When *multiple* jobs are submitted to the cluster, the scheduler decides how to allocate available task slots across jobs.

The default scheduler in Hadoop implements a FIFO policy: the whole cluster is dedicated to individual jobs in sequence; optionally, it is possible to define priorities associated to jobs. In practice, the FIFO scheduler works as follows: it assigns tasks (MAP or REDUCE) in response to heartbeats sent by each individual TASKTRACKER, which reports the number of free MAP and REDUCE slots available for new tasks. Task assignment is accomplished by scanning through all jobs that are waiting to be scheduled, in order of priority and job submission time. The goal is to find a job with a pending task of the required type (MAP or REDUCE). In particular, for MAP tasks, once the scheduler chooses a job, it will select greedily the more suitable task to achieve *data locality*. In this work we also consider the Hadoop Fair Scheduler, which we call FAIR. FAIR groups jobs into "pools" (generally corresponding to users or groups of users)and assigns each pool a guaranteed minimum share of cluster resources, which are split up among the jobs in each pool. In case of excess capacity (because the cluster is over dimensioned with respect to its workload, or because the workload is lightweight), FAIR splits it evenly between jobs. When a slot on a machine is free and needs to be assigned a task, FAIR proceeds as follows: if there is any job below its minimum share, it schedules a task of that particular job. Otherwise, FAIR schedules a task that belongs to the job that has received less resources, based on the notion of "deficit."

3. HADOOP FAIR SOJOURN PROTOCOL

HFSP is a concrete implementation of the multi-processor FSP introduced in Section 2.1. The abstract idea is simple: prioritize jobs according to the completion time they would have using processor sharing, and always use preemption to allocate resources to the highest-priority job. The size of new jobs is at first estimated roughly based on their number of tasks; by observing the running time of the first few tasks, that estimate is then updated. Implementing HFSP in practice, though, raises a variety of issues; in the following we detail how we tackled them.

3.1 The Job Scheduler

HFSP runs on a multi-processor system, managing jobs that may require a different number of MAP and REDUCE

[3] Precisely, a configuration parameter indicates the fraction of mappers that are required to finish before reducers are awarded an execution slot.

tasks: unlike FSP and like the multi-processor variant illustrated in Figure 2 on the preceding page, a job might need only a fraction of the resources of a cluster, and therefore two or more jobs may be scheduled at the same time.

Job Dependencies. In MapReduce, a job is composed by a Map phase followed optionally by a Reduce phase. We estimate job size by observing the time needed to compute the first few "training" tasks of each phase; for this reason we cannot estimate the length of the Reduce phase when scheduling Map tasks. Because of this, for the purpose of scheduling choices we consider Map and Reduce phases as two separate jobs. For ease of exposition, we will thus refer to both Map and Reduce phases as "jobs" in the remainder of Section 3. As we experimentally show in Section 4, the good properties of size-based scheduling ensure shorter mean response time for both the Map and the Reduce phase, resulting of course in better response times overall.

Scheduling Policy. HFSP works by estimating the size of each job using the training module described in Section 3.2; this size is then used to compute the completion time of each job in a simulated processor-sharing scheduler (see Section 3.3). When jobs arrive, all cluster resources are assigned to fulfill demands of jobs ranked by increasing simulated completion time. If a job ends up having more running tasks than assigned, excess slots are released by using the preemption mechanisms described in Section 3.4.

Training Phase. Initially, the training module provides a rough estimate for the size of new jobs (Section 3.2.1). This estimate is then updated after the first s "sample" tasks of a job are executed (Sections 3.2.2 and 3.2.3). To guarantee that job size estimates quickly converge to more accurate values, the first s tasks of each jobs are prioritized, and the above mechanism is amended so that a configurable number of Map and Reduce "training" slots are always available, if needed, for the sample tasks. This number is limited in order to avoid starvation for "regular" jobs in case of a bursty arrival of a large number of jobs.

Data locality. In order to minimize data transfer latencies and network congestion, it is important to make sure that Map tasks work on data available locally. For this reason, HFSP uses the *delay scheduling* strategy [37], which postpones scheduling tasks operating on non-local data for a fixed amount of attempts; in those cases, tasks of jobs with lower priority are scheduled instead.

3.2 Job Size Estimation

Size-based scheduling strategies require knowing the size of jobs. In Hadoop, we do not have a perfect knowledge of the size of a job until it is finished; however, we can at first estimate job size based on characteristics known *a priori* such as the number of tasks; after the first sample tasks have executed, the estimation can be updated based on their running time.

This estimation component has been designed to result in minimized response time rather than coming up with perfectly accurate estimates of job length; this is the reason why sample jobs should not be too many (our default is $s = 5$), and they are scheduled quickly. We stress that the computation performed by the sample tasks is *not* thrown away: the results computed by sample tasks will be used in the rest of the job exactly as the other regular tasks.

3.2.1 Initial Estimation

When a new job arrives in the system, a first rough estimate is used; after obtaining enough information from the sample tasks, this estimate will be updated as described in the following sections. In Hadoop, the number of Map and Reduce tasks each job needs is known *a priori*. In turn, each Map task processes data residing on a single, fixed-size, HDFS block. Our first approximation will be therefore directly proportional to the number of tasks per job.

The size of a Map (resp. Reduce) job with k tasks is, at first, estimated as $\xi \cdot k \cdot l$, where l is the average size of past Map (resp. Reduce) tasks, and $\xi \in [1, \infty]$ is a tunable parameter that represents the propensity that the system has to schedule jobs of unknown size. At the extreme $\xi = 1$, new jobs are scheduled quite aggressively based on the initial estimate, with the possible drawback of scheduling particularly large jobs too early. At the extreme $\xi = \infty$, conversely, non-sample tasks are only scheduled if all jobs which are not in the training phase are satisfied. This avoids resorting to preemption for jobs that turn out to be larger than at first envisioned, but this choice may penalize small jobs that could be scheduled quickly as they arrive in the scheduler, increasing the response time by adding so-called "waves" (i.e., tasks scheduled after a first set of tasks is already completed).

We note that these problems do not generally apply to particularly small jobs having s or less tasks, since all tasks are tagged as sample and in most cases scheduled quickly.

3.2.2 Map Phase Size

It has been observed [37, 15], across a variety of jobs, that Map task execution times are generally stable and short, with execution times around one minute. For this reason, it is possible to perform job size estimation when the s sample tasks are completed. Our estimation builds an approximate cumulative distribution function (CDF) with the data points obtained by measuring, for each sample task j of job i, the execution time $\sigma_{i,j}$; via regression analysis (in particular, least-square fitting), that approximate CDF is then fitted to a target parametric distribution. The estimated Map phase size is finally computed as k times the expected value of the resulting fitted distribution, where k is the number of Map tasks for the estimated job.

Data Locality. A Map sample task could perform worse than normal due to network latencies if operating on non-local data. However, since the sample tasks are between the first to be scheduled, there is a larger choice of blocks to process, making the need of operating on remote data less likely. In combination with the delay scheduling strategy described in Section 3.1, we found that data locality issues on sample tasks, as a result, are negligible.

3.2.3 Reduce Phase Size

The Reduce phase can be broken down in two parts: Shuffle time – needed to move and merge data from mappers to reducers, and the execution time of the Reduce function, which can only start when the Shuffle phase has completed.

Size of SHUFFLE. In general, as soon as a REDUCE task is scheduled, it starts pulling data from the output of mappers; once data from all mappers is available, a *global sort* is performed by merging all the mappers' output. Since each mapper output is already locally sorted, a simple $O(n)$ merge step is sufficient. Network transfer and sort time are essentially proportional to the size of the data to move, and the SHUFFLE performance depends mostly on the amount of data and the characteristics of the cluster it is run on, rather than on the type of data that is moved.

For this reason, an approximate duration of the SHUFFLE phase can be computed as follows. *For each* of the s sample REDUCE tasks of a job that enters the training module, we measure the time required for their SHUFFLE phase. This is given by the difference between the moment a task executes the REDUCE function, and the moment the same task was scheduled in the training module. The estimated SHUFFLE time of the entire REDUCE phase is then the *weighted average* of the individual SHUFFLE times of the sample tasks multiplied by the total number of REDUCE tasks of the job, where the weights are the normalized input data size to each sample task.

Execution Time. The execution time of the REDUCE phase is evaluated analogously to the MAP phase described in Section 3.2.2. However, REDUCE tasks can be orders of magnitude longer than MAP tasks, therefore we aim at providing an estimate of the duration of the sample tasks before their completion. In particular, we set a timeout Δ (60s by default). If a sample task j of job i is not completed by the timeout, its estimated execution time will be $\tilde{\sigma}_{i,j} = \frac{\Delta}{p_{i,j}}$, where $p_{i,j}$ is the *progress* done during the execution stage. The progress of a task is computed as the fraction of data processed by a REDUCE task over the total amount of its input data.

Once we obtain the size (or an estimation of it) for each sample task, we compute the total execution time using the same procedure described in Section 3.2.2, using regression analysis to fit measured data to a parametric distribution, and multiplying the expected value of the resulting distribution by the number of REDUCE tasks in the job. The final estimate of the whole REDUCE phase is obtained by adding the estimated SHUFFLE time to this estimated execution time.

3.3 Virtual Cluster

The estimated job size is expressed in a "serialized form", that is the sum of runtimes of each of its tasks. This is useful because the physical configuration of the cluster does not influence the estimated size; moreover, in case of failures, the number of available TASKTRACKERs varies, but the size of jobs does not change. The job scheduler, though, requires an estimated completion time that depends on the physical configuration of the real cluster. We obtain that by simulating a processor sharing discipline applied on a *virtual cluster* having the same number of machines and the same configuration of slots per machine as the real cluster. The projected finish time obtained in the virtual cluster is then fed to the job scheduler, which will use it to perform its scheduling choices.

Resource Allocation. Virtual cluster resources need to be allocated following the principle of a fair queuing discipline. Since jobs may require less than their fair share, in HFSP, resource allocation in the virtual cluster uses a *max-min fairness* discipline. Max-min fairness is achieved using a round-robin mechanism that starts allocating virtual cluster resources to small jobs (in terms of their number of tasks). As such, small jobs are implicitly given priority in the virtual cluster, which reinforces the idea of scheduling small jobs as soon as possible.

Job Aging. The HFSP algorithm keeps track of, in the virtual cluster, the amount of work done by each job in the system. Each job arrival or task completion triggers a call to the job aging function, which uses the time difference between two consecutive events as a basis to distribute progress among each job currently scheduled in the virtual cluster. In practice, each running task in the virtual cluster makes a progress corresponding to the above time interval. Hence, the "serialized" representation of the remaining amount of work for the job is updated by subtracting the sum of the progress of all its running tasks in the virtual cluster.

3.4 Job Preemption

The HFSP scheduler uses preemption: a new job can suspend tasks of a running job, which are then resumed when resources become available. However, traditional preemption primitives are not readily available in Hadoop. The most commonly used technique to implement preemption in Hadoop is "killing" tasks or entire jobs. Clearly, this is not optimal, because it wastes work, including CPU and I/O. Alternatively, it is possible to WAIT for a running task to complete, as done by Zaharia *et al.* [37]. If the runtime of the task is small, then the waiting time is limited, which makes WAIT appealing. In fact, we suspend jobs using the WAIT primitive for MAP tasks which are generally small. However, for tasks with long runtime, the delay introduced by this approach may be too high.

Eager Preemption. Since REDUCE tasks may be orders of magnitude longer than MAP tasks, to preempt REDUCE jobs we adopt a more traditional approach, which we name *eager preemption*: tasks are suspended in favor of other jobs, and resumed only when their job is later awarded resources. Eager preemption requires implementing SUSPEND and RESUME primitives: we delegate them to the operating system (OS). MAP and REDUCE tasks are launched by the TASKTRACKER as child Java Virtual Machines (JVMs); child JVMs are effectively processes, which we suspend and resume using standard POSIX signals: `SIGSTOP` and `SIGCONT`. When HFSP suspends a task, the underlying OS proceeds with its materialization on the secondary storage (in the swap partition), *if and when* its memory is needed by another process. We note that our implementation introduces a new set of states associated to an Hadoop task, plus the related messages for the JOBTRACKER and TASKTRACKER to communicate state changes and their synchronization.

Task Suspension. As discussed in Section 3.1, the job scheduler allocates cluster resources to jobs that finish first, as computed in the virtual cluster. A new job arriving in the system may induce – depending on its size – the job scheduler to SUSPEND a running job. In practice, the job scheduler suspends tasks, rather than jobs: task suspension works as follows. Upon reception of a heart-beat from a TaskTracker, the job scheduler verifies whether a job tagged for suspension occupies resources. If this is the case, it suspends enough tasks of that job to provide the needed resources to the job that should take its place.

It may happen that not all the tasks of a job have to be suspended: in that case, tasks that have been launched last are the ones which get suspended, for two practical reasons: i) "oldest" tasks are the most likely ones to finish first, freeing resources to other tasks; ii) "young" tasks are likely to have smaller memory footprints, resulting in lower overhead should they be materialized to the swap partition.

Resume *Locality.* When a suspended job is resumed, we take care to avoid redundant work by not restarting from scratch tasks in a suspended state. TaskTrackers with free slots that host suspended tasks for a resumed job will Resume those tasks (oldest tasks get resumed first, because of the reasons just outlined for suspension). On the other hand, TaskTrackers with free slots but no suspended tasks will only schedule new tasks that have not been previously scheduled. In other words, suspended tasks will only be rescheduled as soon as a slot frees up in the TaskTracker they are suspended on.

Maximum Number of Suspended Jobs. Suspending tasks has a cost in terms of storage space requirements. If many tasks on a single machine are suspended, process data could use a large fraction of the RAM available on a machine and eventually could also deplete the swap space. This is an extreme case that only arises with particular workloads (several jobs arriving in decreasing size); we address it by defining a threshold on the number of tasks that can be suspended. When too many tasks are suspended, HFSP switches to the Wait-based preemption technique, until conditions are met for reverting to eager preemption.

Setting a limit on the number of suspended processes effectively implies limiting also the total amount of memory allocated to suspended processes, since in Hadoop the amount of memory allocated per Reduce task cannot exceed a configurable threshold (the default is 1 GB) [2].

Side effects. Eager preemption should be used with care in case of MapReduce jobs that operate on "external" resources, e.g. that heavily use Hadoop *streaming* or *pipes*. Our implementation can be easily extended to provide API support to inhibit Suspend and Resume primitives for such particular workloads.

4. EXPERIMENTS

This Section focuses on a comparative analysis between FAIR and HFSP schedulers. For the sake of readability – due to extremely large sojourn times – we omit the default Hadoop FIFO scheduler from our figures.

Next, we specify the experimental setup and present a series of results, organized in macro and micro benchmarks. Macro benchmarks illustrate the global performance of the schedulers we study in this work, in terms of job sojourn times. Micro benchmarks, instead, focus on the peculiarities of HFSP.

4.1 Experimental Setup

In this work we use both a cluster deployed on Amazon EC2 [1] – which we label the Amazon Cluster – and the standard Hadoop emulator, Mumak [5]. The Amazon Cluster is configured as follows: we deploy 100 "`m1.xlarge`" EC2 instances, each with four 2 GHz cores (eight virtual cores), 4 disks that provide roughly 1.6 TB of space, and 15 GB of RAM.[4] In our experiments – with the Amazon Cluster and with Mumak – the HDFS block size is set to 128 MB and a replication factor of 3, while the main Hadoop configuration parameters are as follows: we set 4 Map slots and 2 Reduce slots per node.

Workloads. Generating realistic MapReduce workloads is a difficult task, that has only recently received some attention. In this work, we use SWIM [8], a standard benchmarking suite that comprises workload and data generation tools, as described in the literature [15, 13, 14]. A workload expresses in a concise manner *i)* job inter-arrival times, *ii)* a number of Map and Reduce tasks per job, and *iii)* job characteristics, including the ratio between output and input data for Map tasks. For our experiments, we use two workloads synthesized from traces collected at Facebook and publicly available on [8] – that we label `FB2009` and `FB2010` – as done in previous works [37, 15]. Table 4 describes the workloads we use, as generated by SWIM for a cluster of 100 nodes.

The `FB2009` workload comprises 100 unique jobs, and is dominated by small jobs, i.e., jobs that have a small number of Map tasks and no Reduce tasks. The job inter-arrival time is a random variable with an exponential distribution and a mean of 13 seconds, making the total submission schedule 22 minutes long. Experiments with the `FB2009` dataset illustrate scheduling performance when the system is not heavily loaded, but has to cope with the demand for interactivity (small jobs).

The `FB2010` workload comprises 93 unique jobs, which have been selected by instructing SWIM to filter out small jobs. In particular, the number of Map tasks is substantially larger than the number of available slots in the system: Map phases require multiple "waves" to complete. The number of Reduce tasks varies between 10 and the number of available reduce slots in the cluster. The job inter-arrival time is a random variable with an exponential distribution and a mean of 38 seconds, making the total submission schedule roughly 1 hour long. The `FB2010` dataset is particularly demanding in terms of resources: as such, scheduling decisions play a crucial role in terms of system response times.

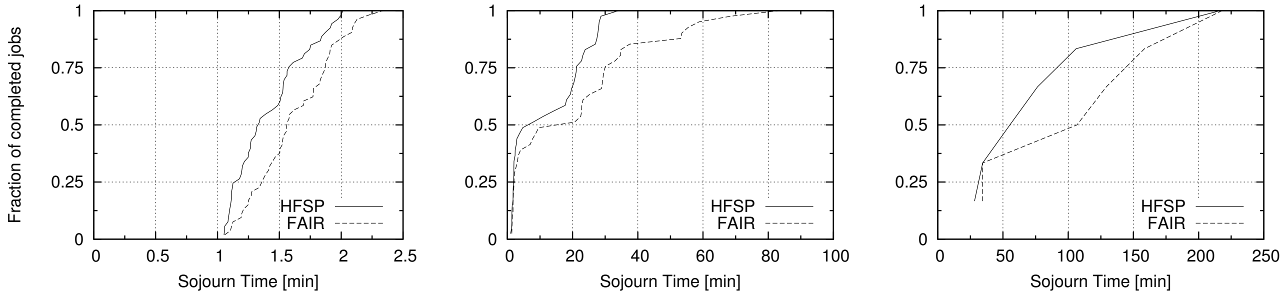
Scheduler Configuration. Unless otherwise stated, HFSP operates with the delay scheduling technique and eager preemption enabled. HFSP requires a handful of parameters to be configured, which mainly govern the estimator component: we use a uniform distribution to perform the fitting of the estimate job size; the sample set size s for Map and Reduce tasks is set to 5; the parameter Δ to estimate Reduce task size, is set to 60 seconds; we set the parameter $\xi = 1$. For the workloads we use in our experiments, the parameters described above give the best results.

The FAIR scheduler has been configured with a single job

[4] This is the configuration used by Zaharia *et. al.* [37].

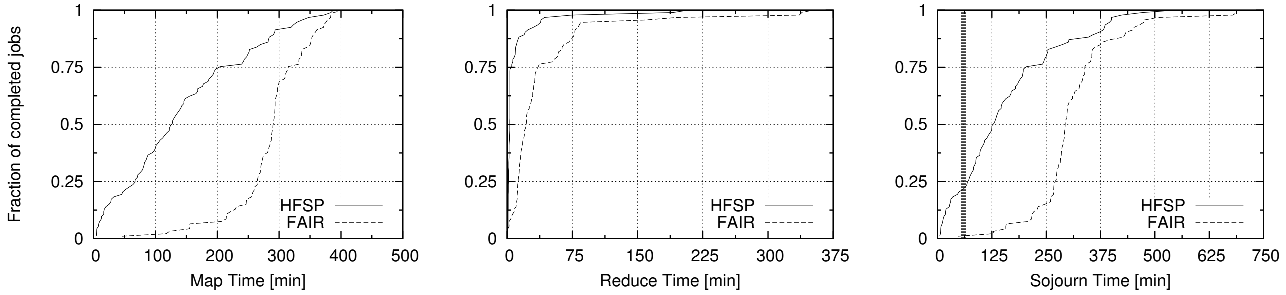
	% of Jobs	# Maps	# Reduces	Label
FB09	(Small jobs) 53	≤ 2	0	select
	(Medium jobs) 41	$2 < x \leq 500$	≤ 500	aggregate
	(Big jobs) 6	≥ 500	≥ 500	transform
FB10	39	≤ 1500	≤ 10	expand
	16	≤ 1500	$10 < x \leq 100$	expand and transform
	11	≤ 1500	$x > 100$	transform
	10	$1500 < x \leq 2500$	$x \leq 100$	aggregate
	7	$1500 < x \leq 2500$	$x > 100$	transform
	10	$x > 2500$	$x > 100$	transform

Table 1: Job types in each workload, as generated by SWIM. Jobs are labeled according to the analysis tasks they perform. For the FB2009 dataset, jobs are clustered in bins and labeled according to their size.



(a) Small jobs (b) Medium Jobs (c) Large Jobs

Figure 3: ECDFs of sojourn times for the FB2009 dataset. Jobs are clustered in classes, based on their sizes. HFSP improves the sojourn times in most cases. In particular, for small jobs, HFSP is slightly better than FAIR, whereas for larger jobs, sojourn times are significantly shorter for HFSP than for FAIR.



(a) Map phase (b) Reduce phase (c) Aggregate

Figure 4: ECDFs of sojourn times for the FB2010 dataset. The Map phase (left) shows significant improvements of HFSP over FAIR. Shorter sojourn times in the Map phase are reflected in the Reduce phase (middle), which shows that HFSP outperforms FAIR in terms of sojourn times. The ECDF of aggregate job sojourn times (right) indicates that both median and worst-case sojourn times are better with HFSP. The vertical line at 60 minutes (the workload has been consumed) indicates 20% of completed jobs with HFSP vs. 1% of completed jobs with FAIR.

pool, using the parameters suggested in the Hadoop scheduler documentation [3, 4].

4.2 Macro Benchmarks

We now use the Amazon Cluster and report the empirical cumulative distribution function (CDF) of sojourn times for FAIR and HFSP, when the cluster executes the FB2009 and FB2010 workloads. Although we do not include results we obtain with the FIFO scheduler, for reference, our experiments report a mean sojourn time 5 to 10 times bigger than that of HFSP, depending on the workload.

Figure 3 groups results according to job sizes: the FB2009 dataset contains three distinct clusters of job sizes (small, medium and large), with small jobs dominating the workload. Our results indicate a general improvement of job sojourn times in favor of HFSP. For small jobs, the fair share of cluster resources allocated by both HFSP and FAIR is greater than their requirements in terms of number of tasks.

In addition, very small jobs (with 1-2 Map tasks only) are scheduled as soon as a slot becomes free (both under the HFSP and FAIR), and therefore their sojourn time depends mainly on the frequency at which slots free-up and on the cluster state upon job arrival. Overall, we observe that HFSP performs slightly better than FAIR for small jobs. For medium and large jobs, instead, an individual job may require a significant amount of cluster resources. Thus, the advantage of HFSP is mainly due to its ability to "focus" cluster resources – as opposed to "splitting" them according to FAIR – towards the completion of the smallest job waiting to be served.

Figure 4 shows the results for the FB2010 dataset. We show the sojourn times of the Map and Reduce phases, and the total job sojourn times. In the Map phase (Figure 4(a)), the sojourn times are smaller for HFSP than for FAIR: the median sojourn time is more than halved. This is a consequence of the HFSP discipline, which dedicates cluster resources to individual jobs rather than spreading them on multiple "waves". As such, with HFSP, Reduce tasks enter the Shuffle phase earlier than what happens with FAIR, and have to wait less time for all their input data to be available. Therefore, the Reduce phase is shorter with HFSP, as shown in Figure 4(b). Clearly, the HFSP discipline in the Reduce phase also contributes to short sojourn times, with a median difference of roughly 30 minutes. Finally, the total job sojourn times indicate that the system response time with HFSP substantially improves. For illustrative purposes, we show a vertical line corresponding to 60 minutes, by which all jobs of the FB2010 workload arrived in the system. By that time, only one job completes with FAIR, whereas more than 20% of the jobs, are served with HFSP. More to the point, when 80% of jobs are served by HFSP, roughly 15% of jobs are served with FAIR.

In summary, HFSP caters both to workloads geared towards "interactive" small jobs and to a more efficient allocation of cluster resources, which is beneficial to large jobs. Next, we focus on HFSP in particular and study the behavior of its inner components in detail.

4.3 Micro Benchmarks

In the following set of experiments, when not otherwise stated, we use the Mumak emulator to execute the workloads generated by SWIM and described earlier; in addition, we use synthetic workloads that reproduce peculiar cases to better explain how HFSP works.

Impact of Cluster Size. The goal of this experiment is to study the behavior of both HFSP and FAIR, when cluster resources available for scheduling decisions are scarce. Indeed, many real-life deployments [7] are smaller than 100 nodes, yet they are used to execute a large number of jobs on big data volumes. We use the FB2009 dataset, because it contains a mix of different job sizes, including small jobs. In our experiments, we vary the cluster size in the range of 10 to 100 machines. When the cluster size is small, we increase accordingly the storage space available at each node, to accommodate the data volume used in the FB2009 workload.

Figure 5 reports the mean job sojourn time for HFSP and FAIR, as a function of the number of machines in the cluster. When resources are scarce, HFSP achieves considerably smaller mean job sojourn times due to its ability to "focus" cluster resources to individual jobs. Put otherwise, for equivalent job sojourn times, the FB2009 workload requires a smaller cluster when HFSP is used, as compared to other schedulers, which is a desirable property as it may relate to significant cost savings. Clearly, our results hold (and the gap in favor of HFSP is even more substantial), when the FB2010 dataset is used.

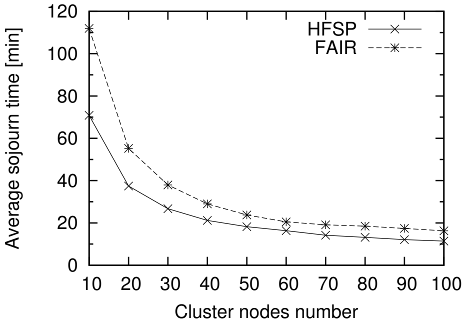


Figure 5: Impact of cluster size on scheduling performance, for both HFSP and FAIR, using the FB2009 dataset.

Robustness to Estimation Errors. In this experiments we study whether HFSP incurs performance degradation when job size – an essential ingredient for the correct operation of the scheduler – is incorrect. We do so by injecting artificial errors on the job size estimates reported by the Training module of HFSP, and measure the impact in terms of mean job sojourn times.

For this experiment, we focus only on errors injected in the estimator of the Map phase size. To do so, we use a modified, Map only version of the FB2009 dataset. In practice, the Map size estimate used by the virtual cluster of HFSP is a random variable uniformly distributed in the range $[\theta()\cdot(1-\alpha), \theta()\cdot(1+\alpha)]$, where $\theta()$ is the size estimate computed as in Section 3.2, and $\alpha \in [0.1, 1]$ is the artificial error we inject. We repeat each experiment 20 times for each value of α, to gain statistical confidence in our results.

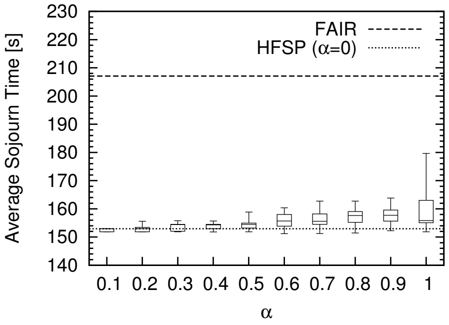


Figure 6: Impact of job size estimation errors on HFSP performance, using a Map only version of the FB2009 dataset.

Figure 6 reports the mean sojourn time for different α values. In addition, we show as a reference the mean sojourn time for the same experiments using FAIR, which are clearly not affected by estimation errors, and the mean sojourn time achieved by HFSP with $\alpha = 0$. In our experiments, HFSP is particularly resilient to wrong estimates: indeed, the mean sojourn times is slightly affected only for extremely large errors. Indeed, macroscopic changes in sojourn time happen only when long jobs are scheduled before much shorter ones, which would then incur a large sojourn time. In the FB2009 dataset – which is grouped in classes according to job size – "reversals" (i.e., jobs that are scheduled in a different order) only happen within the same class, with only a modest impact on sojourn times.

Data Locality. Next, we focus on data locality and measure the fraction of tasks that read data from the local disk of the machine they run on.

We use the same experiments described in Section 4.2, where we use the Amazon Cluster to execute the FB2009 and FB2010 datasets, for both HFSP and FAIR. For the sake of brevity, we only report results for the FB2009 dataset.

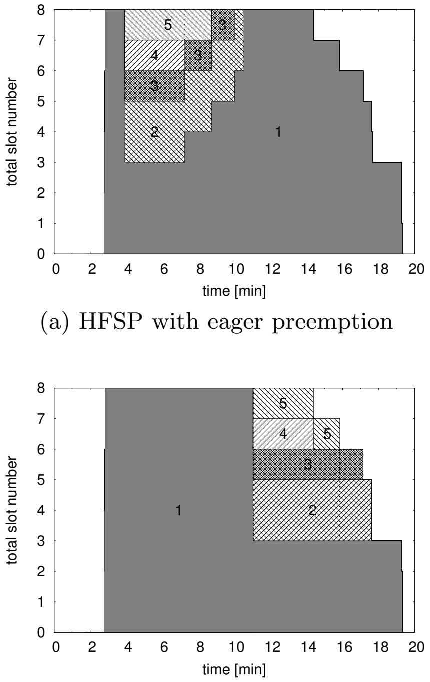
FAIR achieves 98% of data locality whereas HFSP always achieves 100% of data locality, over a total of more than 14,000 tasks across all experiments. Clearly, the delay scheduler mechanism [37] is beneficial to both FAIR and HFSP. Additionally, we observe that the result we obtain is also a consequence of resource allocation: with HFSP, a job scheduled for execution receives (if the cluster size allows it) all the resources required for its processing, whereas with FAIR, it is granted fewer resources. As a consequence HFSP copes better with the random data placement strategy used by HDFS, and obtains more local tasks, which contributes to shorter job execution times and hence smaller sojourn times.

4.4 Job Preemption

We now study in detail the various preemption mechanisms we discuss in Section 3.4. First, we compare the behavior of HFSP with eager preemption, with the Wait primitive, and with a Kill primitive, for a simple workload. The goal of such experiment is to illustrate the benefits of eager preemption, in terms of job sojourn times, and to discuss when alternative mechanisms are to be preferred. Then, we move to another set of experiments, in which we study the performance of the Suspend and Resume operations, executed in a small local cluster.

For the first set of experiments we use a simple, synthetic workload composed of five jobs, and focus solely on Reduce tasks, since Map tasks are not preempted with HFSP. We set up a small cluster in Mumak, with 4 machines with 2 Reduce slots each. In our workload, the first job, j_1, has 11 reduce tasks each of duration roughly 500 seconds and arrives at time 2 minutes and 20 seconds. All the other jobs arrive at time 2 minutes and 30 seconds and all have one Reduce task, except for j_2 that has two Reduce tasks. For jobs $j_2 \cdots j_5$, Reduce task times are smaller than the one of j_1.

Figure 7 illustrates a resource allocation graph that we obtain by processing Hadoop log files: on the y-axis we report the cumulative slot utilization per job, on the x-axis we report time, in minutes. In Figure 7(a), which shows the behavior of HFSP with eager preemption, when jobs j_2, j_3, j_4 and j_4 arrive, they preempt j_1 and occupy the cluster with their tasks. Note that HFSP suspends only the required number of tasks of j_1 to accommodate the newly arrived jobs. When jobs $j_2 \cdots j_5$, the suspended tasks of job j_1 are resumed. The average sojourn time in this simple example is about 9 minutes. Instead, in Figure. 7(b), in which HFSP uses the Wait primitive, when jobs j_2, j_3, j_4 and j_4 arrive, the cluster is fully occupied by j_1. As such, HFSP waits for job j_1 to complete the required number of tasks necessary to allocate the new jobs, before proceeding with scheduling. As a consequence, the average sojourn time is 15 minutes, roughly 40% larger than with preemption. We also repeat the very same experiment by implementing a simple Kill primitive: in this case, job j_1 has a larger completion time because 6 of its tasks are killed due to the arrival of jobs $j_2 \cdots j_5$, and thus need to be scheduled again. We omit the resource allocation graph for this case, as it is very similar to that in Figure 7(a).



(a) HFSP with eager preemption

(b) HFSP with Wait

Figure 7: Resource allocation graphs for a simple workload, with and without eager preemption.

It is possible to define alternative scenarios in which HFSP could achieve better sojourn times with the Wait primitive. In general, when task runtimes are short, the Wait primitive is to be preferred: eager preemption may need to perform process swapping, which could take longer than the remaining time for the task to complete. Instead, when task runtimes are long, eager preemption is a more sensible choice, as it brings shorter sojourn times.

Next, we study the time it takes, for the OS, to perform the Suspend and Resume operations, as this determines to a great extent whether eager preemption is to be preferred over the Wait primitive.

Swap Times. Recall that, in our implementation of HFSP, when one or more tasks of a job are preempted, the memory that they are using can be claimed by other tasks scheduled to occupy their slot. In this case, the Operating System (OS) may swap the memory contents to disk. When such preempted task are resumed, the OS reloads in memory the swapped process from disk. As such, the SUSPEND and RESUME operations may introduce delays that we seek to compare to task times. We remark that such delays are bounded: indeed, the memory footprint of a task is limited by the way a MapReduce job is engineered. When a task is preempted, the amount of memory it uses is bounded by the amount of RAM per slot, a parameter configured in Hadoop. As such, the disk I/O that characterizes cluster machines, together with the amount of RAM used by REDUCE tasks, are the main factors to consider when configuring HFSP to use eager preemption.

To verify our claim, we perform the following experiment on a local cluster, with a single TASKTRACKER and a single REDUCE slot. The hardware configuration we use is as follows: the TASKTRACKER machine has 8 GB of RAM, and a local 7.2Krpm disk with a read/write I/O speed of roughly 100 MBps. The Linux "swappiness" [9] is configured to a small (5) value; in addition, swap size is configured such that it can accommodate several memory dumps.

We define a simple workload[5] in which a job with a single REDUCE task occupies 6 GB of RAM (with randomly generated data), and the heap space of the child JVM running the task is set to 6 GB. Then, a new job with a single REDUCE task arrives in the system, and preempts the initial job. The "swap" time is defined as the time it takes for the system to SUSPEND the initial REDUCE task and to RESUME it. The new job is characterized by a variable memory footprint, varying from 0.5 GB up to 6 GB.

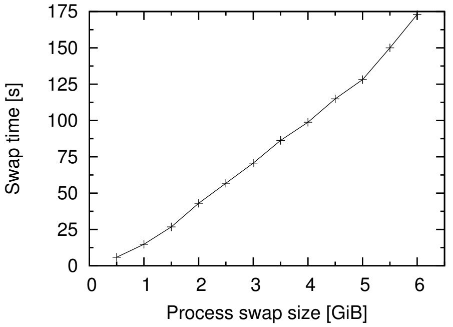


Figure 8: Swap times for a small cluster, using a synthetic workload of two jobs with one REDUCE task each. The second REDUCE task has a variable memory footprint, which corresponds to the amount of data swapped to disk by the OS, as reported on the x-axis.

Figure 8 reports the swap time for the experimental setup described above. Swap time is approximately linear in the size of the memory footprint of the preempting task, up to a process swap size of 5 GB. Our results indicate that the SUSPEND and RESUME primitives operates roughly at disk raw-speed: for example, for a process swap size of 2 GB, the systems writes, and then reads 2 GB in roughly 40 seconds, giving an I/O speed of roughly 100 MBps. Then, swap performance slightly decreases after 5 GB, which accounts for the overhead of the OS trying to balance resources dedicated to the swap partition and to caching.

Putting the results in perspective, we now refer to the experiments with the FB2010 dataset described in Section 4.2. Our logs indicate that at most 10% of running tasks have been suspended and then resumed: such tasks belong to 10 different jobs of the workload. In those experiments, the JVM heap space for a REDUCE task is set to 1 GB [2], which means that SUSPEND and RESUME operations induce at most a swap time around 20 s. Since the average duration of preempted tasks is one order of magnitude larger than the swap time, we conclude that eager preemption is a sensible configuration for HFSP with the workload we used.

5. DISCUSSION

We now discuss several points that complement the work we have presented so far.

Preemption Performance. Some works have considered job preemption primitives alternative to the ones we propose in this work. For example, Cheng *et al.* [16] present a detailed analysis of a KILL primitive to implement job preemption and come up with a method to select the best tasks to kill to avoid hurting too much job performance. Instead, Anantharanayanan *et al.* [11] observe that job preemption can be implemented at the "application-level", meaning that individual tasks may be suspended and resumed by Hadoop, which would require a dedicated memory-management module to handle preemption.

One reason to avoid using OS paging might be that it could affect system stability [11]: a large number of interactive applications that require more RAM than that available on the system, substantially hinder the task of the OS.

Given the results we show in Section 4.4, it is reasonable to assume that an "application-level" preemption mechanism cannot perform better in moving data from/to RAM and disk, than what we propose in our work. In addition, Hadoop jobs are not interactive applications that would need frequent context switching. As such, while we do not dismiss any effort to study alternative mechanisms, we believe the eager preemption we present in this work to be effective and practical in achieving its goals.

Job with Different Priorities. The design of HFSP takes as a reference the Processor Sharing (PS) discipline to compute the order of the jobs to be scheduled. In PS, each job receives its equal share of the resources. A natural extension of the work would provide different priorities, or weights, to jobs: in this case, we shall consider the Generalized Processor Sharing (GPS) discipline, where each job receives an amount of resources in proportion to its weight. For instance, if $\mathcal{J}$ is the set with all the jobs in the system, then job k with weight w_k will receive a fraction $\frac{w_k}{\sum_{i \in \mathcal{J}} w_i}$ of the resources. This computation can be easily incorporated in the job aging computation done by the HFSP algorithm.

Job Size Estimation. In some cases, task execution time, which contributes to job duration, could be regarded as a highly variable quantity, making task time distributions highly skewed, thus difficult to estimate.

We remark that in HFSP, the estimator is designed as a pluggable module that can be replaced by more sophisticated estimation techniques, therefore providing more accu-

[5] We omit MAP tasks, as they are not preempted with HFSP.

rate predictions. In addition, in recent work Kwon *et al.* [24] address and greatly mitigate the issue of skew in task processing times with a plug-in module that seamlessly integrates in Hadoop, which can be used in conjunction with HFSP. Moreover, Popescu *et al.* [27] present an appealing approach to predict MapReduce "query" runtime, that can be also used in HFSP. We conclude by remarking that the original FSP discipline has also been studied in the case of inaccurate job size information [25]: according to such work, FSP is a stable algorithm that is robust to inaccurate job size, a result that we confirm in the context of this paper.

6. RELATED WORK

MapReduce in general and Hadoop in particular have received a lot of attention recently, both from the industry and from academia. In this work we focus on job scheduling, and consider the literature pertaining this domain. Several theoretical works tackle scheduling problems in a multi-processor system – see for instance [18]. These works, which represent elegant and important contributions to the domain, consider jobs with a simple structure (*i.e.,* a single phase) and make several simplifying assumptions on the underlying execution system. The main objective of such theoretical studies is to offer bounds on job performance provide optimality results. In contrast, in this work we take a system approach, and focus on the design and implementation of a scheduling mechanism taking into account all the details and intricacies of a real system.

More recently, the problem of job scheduling in MapReduce has revived interest in theoretical approaches to study job performance. Some works [12, 26] provide interesting approximability results but fail in providing a truthful model of the underlying MapReduce system. In the same vein, but with results that are readily applicable, Tan *et al.* [31] identify several shortcomings of the FAIR scheduler we also study in this work and proposes an elegant model of job runtimes. Their contribution aims at mitigating job starvation problems that arise when job runtimes are heavily skewed. In contrast, our goal is, more generally, to overcome problems of processor-sharing disciplines with respect to job sojourn times. As such, the results of Tan *et al.* could be extended to cover our scheduler.

The works that are more closely related to ours, because they have a system approach to scheduling and aim at the design and implementation of a scheduling discipline, are numerous. For instance, the FAIR scheduler and its enhancement with a delay scheduler [37] is a prominent example to which we compare our results. Another work by Tan *et al.* [32] provides more system details on the mechanism used to overcome job starvation with the FAIR scheduler. Several other works [29, 22, 20, 21] focus on resource allocation and strive at achieving fairness across jobs, but do not consider sojourn times. Sandholm and Lai [30] study the resource assignment problem through the lenses of a bidding system to achieve a dynamic priority system and implement quality of service for jobs. Kc and Anyanwu [23] address the problem of scheduling jobs to meet user-provided deadlines, but assume job runtime to be an input to the scheduler. Finally, the work that is more closely related to ours is Flex [36], which provides a framework for the optimization of any given performance metric. In particular, when the performance metric is chosen to be the "max-sum" sojourn-time, Flex should minimize the average sojourn time, whereas in our work we cannot make any optimality claims. Flex is implemented as an add-on on top of the FAIR scheduler, and shares similar design principles to our work. For example, when configured to operate as a size-based scheduler, Flex implements an estimation module to infer job sizes. Unfortunately, we could not compare our proposal to Flex because it is a proprietary software without a sufficient documentation describing its internals.

Finally, various recent approaches [34, 35, 10, 27, 33] have been proposed to infer the size of MapReduce jobs for specific application scenarios. HFSP is designed such that the estimator module can be easily plugged with other mechanisms, benefiting from advanced and tailored solutions.

7. CONCLUSION

The problem of scheduling jobs in parallel systems have received a lot of attention in the past, including works that attempted at producing elegant mathematical models of such systems with the goal of studying the hardness of obtaining optimal scheduling. In this work we took a systems approach, glossing over mathematical constructs and optimality analysis: instead we were interested in studying the benefits of a size-based approach to scheduling jobs in a real system, namely Hadoop.

Our work was motivated by the realization that MapReduce has evolved to the point where shared clusters are used for a wide range of workloads, which include an increasingly large fraction of interactive data processing tasks. Existing schedulers in the state-of-the-art suggest, to overcome the inherent limitations of a simple first-come-first-served discipline, cluster resources to be shared equally among running jobs. As a consequence, we have witnessed the raise of deployment best practices in which long sojourn times were compensated by over-dimensioned Hadoop clusters. Armed with the realization that a large fraction of cluster resources were used for a small amount of time, given a selection of real-world workload traces, in this work we set off to study the benefits of a new scheduling discipline that targeted at the same time short sojourn times and fairness among jobs.

The HFSP scheduler we proposed in this article brought up several challenges. First, we came up with a general architecture to realize *practical* size-based scheduling disciplines, where job size is not assumed to be known a priori. The HFSP scheduling algorithm solved many problems related to the underlying discrete nature of cluster resources, how to keep track of jobs making progress towards their completion, and how to implement eager preemption primitives. Then, we used statistical tools to infer task time distributions and came up with an approach aiming at avoiding wasting cluster resources while estimating job sizes. Finally, we performed a comparative analysis of HFSP with the two standard schedulers that are most widely used today in production-level Hadoop deployments, and showed that HFSP brings several benefits in terms of shorter sojourn-times, even in small, highly utilized clusters.

There are several avenues that we are considering as part of our future work. First, we will extend our experimental study to cover a wider range of workloads, including those presenting issues related to skew in task time distributions. In this work, on of our goal was to avoid wasting work done, both during the training phase, and by using eager preemption. We plan to investigate how to integrate our preemption mechanism with the "speculative execution" mode of

Hadoop, since in some cases, it could be beneficial to sacrifice some work done by a job to gain in system reactivity for scheduling decisions. Finally, we will study the problem of scheduling complex job workflows, that result from the composition of several sub-jobs.

8. REFERENCES

[1] AWS elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Hadoop cluster setup. http://hadoop.apache.org/docs/stable/cluster_setup.html.

[3] Hadoop fair scheduler. http://hadoop.apache.org/docs/stable/fair_scheduler.html.

[4] Hadoop MapReduce JIRA 1184. https://issues.apache.org/jira/browse/MAPREDUCE-1184.

[5] Hadoop Mumak. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[6] Hadoop: Open source implementation of MapReduce. http://hadoop.apache.org/.

[7] Hadoop wiki, powered by. http://wiki.apache.org/hadoop/PoweredBy.

[8] Statistical workload injector for mapreduce. https://github.com/SWIMProjectUCB/SWIM.

[9] sysctl linux kernel documentation. http://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[10] S. Agarwal et al. Re-optimizing Data-Parallel Computing. In Proc. of USENIX NSDI, pages 21–35, 2012.

[11] G. Anantharanayanan et al. True elasticity in multi-tenant clusters through amoeba. In Proc. of ACM SOCC, pages 24:1–24:7, 2012.

[12] H. Chang et al. Scheduling in MapReduce-like Systems for Fast Completion Time. In Proc. of IEEE INFOCOM, pages 3074–3082, 2011.

[13] Y. Chen. We don't know enough to make a big data benchmark suite - an academia-industry view. In Proc. of WBDB, 2012.

[14] Y. Chen, S. Alspaugh, and R. Katz. Interactive query processing in big data systems: A cross-industry study of mapreduce workloads. In Proc. of VLDB, pages 1802–1813, 2012.

[15] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In Proc. of IEEE MASCOTS, pages 390–399, 2011.

[16] L. Cheng, Q. Zhang, and R. Boutaba. Mitigating the negative impact of preemption on heterogeneous mapreduce workloads. In Proc. of CNSM, pages 189–197, 2011.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Proc. of USENIX OSDI, pages 107–113, 2004.

[18] K. Fox and B. Moseley. Online scheduling on identical machines using SRPT. In In Proc. of ACM-SIAM SODA, pages 120–128, 2011.

[19] E. Friedman and S. Henderson. Fairness and efficiency in web server protocols. In Proc. of ACM SIGMETRICS, pages 229–237, 2003.

[20] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resources types. In Proc. of USENIX NSDI, pages 24–38, 2011.

[21] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In Proc. of USENIX NSDI, pages 22–22, 2011.

[22] M. Isard et al. Quincy: fair scheduling for distributed computing clusters. In Proc. of ACM SOSP, pages 261–276, 2009.

[23] K. Kc and K. Anyanwu. Scheduling Hadoop jobs to meet deadlines. In Proc. of CloudCom, pages 388–392, 2010.

[24] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In Proc. of ACM SIGMOD, pages 1934–1937, 2012.

[25] D. Lu, D. L. Huanyuan, and P. Dinda. Size-based scheduling policies with inaccurate scheduling information. In Proc. of IEEE MASCOTS, pages 31–38, 2004.

[26] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlos. On scheduling in map-reduce and flow-shops. In In Proc. of ACM SPAA, pages 289–298, 2011.

[27] A. D. Popescu et al. Same queries, different data: Can we predict query performance? In Proc. of SMDB, pages 275–280, 2012.

[28] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: A comparative workload analysis from three research clusters. In Technical Report, CMU-PDL-12-106, 2012.

[29] T. Sandholm and K. Lai. MapReduce optimization using regulated dynamic prioritization. In Proc. of ACM SIGMETRICS, pages 299–310, 2009.

[30] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In Proc. of JSSPP, pages 110–131, 2010.

[31] J. Tan, X. Meng, and L. Zhang. Delay tails in MapReduce scheduling. In Proc. of ACM SIGMETRICS, pages 5–16, 2012.

[32] J. Tan, X. Meng, and L. Zhang. Performance analysis of Coupling Scheduler for MapReduce/Hadoop. In Proc. of IEEE INFOCOM, pages 2586 –2590, 2012.

[33] C. Tian, H. Zhou, Y. He, and L. Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In Proc. of IEEE GCC, pages 218–224, 2009.

[34] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In Proc. of ICAC, pages 235–244, 2011.

[35] A. Verma, L. Cherkasova, and R. H. Campbell. Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. In Proc. of IEEE MASCOTS, pages 11–18, 2012.

[36] J. Wolf et al. FLEX: A slot allocation scheduling optimizer for MapReduce workloads. In Proc. of ACM MIDDLEWARE, pages 1–20, 2010.

[37] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proc. of ACM EuroSys, pages 265–278, 2010.