HFSP: The Hadoop Fair Sojourn Protocol

Mario Pastorelli\textsuperscript{1} \quad Antonio Barbuzzi\textsuperscript{1} \quad Damiano Carra\textsuperscript{2} \quad Pietro Michiardi\textsuperscript{1}

\textsuperscript{1}EURECOM
Sophia Antipolis, France
first_name.last_name@eurecom.fr

\textsuperscript{2}University of Verona
Verona, Italy
damiano.carra@univr.it

ABSTRACT
This work presents the HFSP scheduler, which implements a size-based scheduling discipline for Hadoop. While the benefits of size-based scheduling disciplines are well recognized in a variety of contexts (computer networks, operating systems, etc.), their practical implementation for a system such as Hadoop raises a number of important challenges.

In HFSP 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 many 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 workload traces available in the literature.

1. INTRODUCTION
The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce\textsuperscript{11}, 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\textsuperscript{8,22} 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 opensource implementation of MapReduce, namely Hadoop\textsuperscript{3}. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several alternatives\textsuperscript{21,6,14,17,24,30} have been proposed to enhance scheduling: in general, existing approaches aim at two key objectives, namely fairness among jobs and performance. Our key observation is that fairness and performance are non-conflicting goals, hence there is no reason to focus solely on one or the other objective. Furthermore, we revisit the notion of scheduling performance and propose to focus on job sojourn time, which measures the time a job spends in the system waiting to be served and its execution time. Short sojourn times cater to the interactivity requirements discussed above.

We thus proceed with the design and implementation of a new scheduling protocol that caters both to a fair and efficient utilization of the cluster resources. Our solution, called 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 is not known a-priori. Essentially, 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 readily available 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 allow preemptive disciplines. HFSP is available as an open-source project.

We design and implement a new scheduling discipline inspired by the Fair Sojourn Protocol [13], which operates in a multi-processor context and that caters to short job sojourn times, when compared to widely used alternatives such as FIFO and processor-sharing schedulers. 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 (and contribute to their further development) state-of-the-art workload suite generators that take as input realistic workload traces. In addition we contribute to the development of the standard Hadoop emulator [2], which we use in conjunction to a large cluster deployed in the Amazon elastic computing cloud. Our results show that the average sojourn time achieved by the jobs of our workload 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 Sect. 2 we provide background information on a set of scheduling disciplines and on some details of Hadoop MapReduce. In Sect. 3 we describe in details the HFSP schedulers and its inner components. We evaluate the performance of our job scheduler in Sect. 4 and provide in Sect. 5 additional considerations. In Sect. 6 we discuss the related work, and we conclude our paper in Sect. 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 (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.

In [13], the authors provide a scheduling policy that strives to obtain both (near) optimal mean response times for all jobs and fairness across all jobs, 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. Essentially, FSP computes the completion time for each job under the 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 \( t_1 = 0 \), \( t_2 = 10s \) and \( t_3 = 15s \) 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).](image-url)
whereas in Sect. 3.1 we detail all the hidden intricacies of a multi-processor version of FSP, called Hadoop Fair Sojourn Protocol (HFSP).

Assume that jobs \( j_1, j_2 \) and \( j_3 \) require 100%, 55% and 35% of the system resources respectively. The arrival times are \( t_1 = 0s, t_2 = 10s \) and \( t_3 = 13s \) 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 \( j_3 \).

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. 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 report 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” 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 resource, based on the notion of “deficit.”

3. HADOOP FAIR SOJOURN PROTOCOL

The design and implementation of a new scheduling component for Hadoop is a delicate task, as scheduling and resource allocation decisions determine, to a large extent, job performance. In the following we highlight the key problems.
we address in this work, namely: the design of the scheduling algorithm for a multi-processor system (cf. Sect. 3.1), an on-line mechanism to estimate job size without “wasting” resources (cf. Sect. 3.2), and finally a set of new primitives to suspend and resume jobs, which is a requirement for preemptive scheduling disciplines (cf. Sect. 5.3).

3.1 The Job Scheduler

The original FSP discipline – which inspires HFSP – is designed for a single-server system, in which jobs have a simple structure. Extending the concepts of FSP to work in a multi-processor system is not trivial. MapReduce jobs have a complex structure, with temporal dependencies among tasks. Moreover, the discrete nature of compute slots to execute jobs affects how job aging – that tracks how much work has been done for each job in the system – is computed. In addition, HFSP requires an appropriate definition of job size, which can handle jobs with different “shapes” (e.g., 100 tasks of 1 s vs. 2 tasks of 50 s). Finally, data locality – that is, making sure that tasks operate on local data – requires special care in taking scheduling decisions.

We now describe the HFSP scheduling algorithm in detail. Let’s assume, for now, job sizes to be known and focus on the issues that arise in a multi-processor setting. Then, we’ll describe the complete operation of HFSP, including the process of estimating job sizes. We remark that the HFSP algorithm is applied, separately, to both the MAP and the REduce phase. The main difference between such phases lies in the how job size estimation is done (cf. Sect. 3.2.1).

The scheduling algorithm is divided in two parts. The first executes every time a new job arrives or whenever a task or a job completes; the HFSP algorithm “simulates” what would happen if the scheduler was to behave as processor sharing, computing an appropriate resource allocation and keeping track of the amount of work done by each job. Then the algorithm sorts jobs according to their projected finish time in the simulated system, which is used to take scheduling decisions in the “real” cluster. The second part executes when a free compute slot is available in the cluster. Such slot is scheduled to execute a task of the first job in the list of jobs sorted by projected finish time in ascending order. Scheduling task execution is conditioned by data locality, as explained later.

The virtual cluster. HFSP uses a virtual cluster to simulate a processor sharing scheduling discipline. The virtual cluster simulates the same resources available in the real cluster: it has the same number of machines and the same configuration of slots (MAP or REduce) per machine. When a job arrives in the system, the virtual cluster uses its (estimated) size to represent the amount of remaining work that job needs to do, which can be used to compute the projected finish time. It is fundamental to notice that in this work the size of a job is expressed in a “serialized” form, that is the sum of the runtimes of each of its tasks, as if they were to be executed in series on a single slot (cf. Sect. 5.2.1). As a consequence, the remaining amount of work of a job is independent of the resources available in the cluster. This choice simplifies the design of a job aging function and mitigates the impact of failures in the underlying cluster. Next, we describe how resource allocation in the virtual cluster and job aging work.

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/job 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.

Data locality. When assigning a new task to a free slot in the cluster, the HFSP algorithm uses the principle of delay scheduling: it first checks whether the tasks has local data to operate on; if data locality on such slot is not possible, the scheduler waits for another slot to become available. After a number of delayed task assignments (in practice, we use the same timeout mechanism used in the original delay scheduler §), the scheduler finally allocates a slot to the task. Note that unused slots for a non-local task are assigned to other jobs. The discussion above clearly applies to MAP tasks, as in general there is no data locality for REduce tasks. As explained in Sect. 5.2, HFSP implements a preemptive scheduling discipline, which calls for a mechanism to handle REduce tasks data locality as well.

3.1.1 HFSP: complete operation

HFSP is built as a hierarchical scheduler in which a top-level scheduler implements a dynamic resource allocation mechanism to provision cluster resources to the job scheduler (described above) and the component (similar in nature to a scheduler) used to estimate job sizes, that we call the Training module (cf. Sect. 5.2). Indeed, job size information is not available in practice. As such, the top-level scheduler aims at minimizing the delay (which contributes to sojourn times) required to proceed with job size estimation. In addition, the top-level scheduler also strives to avoid adding “waves” to a job, which could be introduced by job size estimation and that contribute to larger sojourn times.

When a job arrives in the system, the job scheduler uses an arbitrary size to proceed with its operation. In this work, the initial estimate for a MAP phase is the number of tasks (each corresponding to an HDFS block) times the average duration of recently executed MAP tasks of other jobs. The initial estimate is weighted by a configurable “confi-
**3.2 Job size estimation and training**

HFSP uses a Training module to produce estimates of job sizes, which the scheduler then uses to track the amount of work each job needs to do. The training module uses a pluggable estimator to output job size estimates. When a new job arrives in the system, the Training module executes a fraction of its tasks (that we label the sample set); while the job makes progress, the estimator measures task runtimes and builds a statistic, i.e., it constructs an approximate cumulative distribution function (CDF) of task times. This information is then used to compute the job duration. When scheduling a new job, the Training module assigns its minimum fair share, corresponding to the minimum number of slots (a parameter of HFSP) required by the estimator to build the CDF of task times. Execution slots are assigned according to a “fewer remaining tasks” discipline, which implies short jobs are given priority. As a final note, HFSP requires an additional parameter to decide the maximum amount of slots the top-level scheduler grants to the Training module: this is useful to avoid starvation in the job scheduler, for workloads with bursty arrivals of a large number of jobs.

Before delving into the details of runtime estimators, we stress that the allocation of resources to the Training module described above and, in more general terms, in Sec. 3.1.1 are by far more important to achieve short sojourn times than extremely accurate job size estimates, as we show in our experimental results.

**3.2.1 Runtime estimator**

In the following, we describe our task size estimator: note that the estimator is designed as a pluggable module. Our proposed simple estimator could be replaced by more sophisticated estimation techniques, therefore providing more accurate predictions.

Despite the intricacies of estimating MAP and REDUCE task time distributions (which are handled separately), the estimator is based on simple regression analysis to compute the parameters of a given distribution such that a measure of error (in our case least squares error) is minimized. In practice, each job can be configured such that a reference task time distribution is used by the estimator to come up with an estimated distribution based on the parameters obtained through regression analysis. In our experiments we consider a simplified setting in which there is no skew in task time distribution, which allows building job size estimates using first order statistics.

**MAP phase size.** As observed across a variety of jobs, MAP task execution times are generally stable and short. Now, how large the sample set size should be for computing an estimate of the whole duration of the MAP phase? The number of samples to be used is a trade-off between the estimation speed and accuracy. It is outside the scope of this work to come up with a mathematical model to set the sample set size. We have empirically observed that, using different data center traces, a sample set equal to five MAP tasks provide sufficiently high accuracy (cf. Sect. 4.2).

Let $\mathcal{M}_i$ represent the set of tasks associated to the MAP phase of job $i$, and $\sigma(m_{i,j})$ be the duration of a single MAP task $j$ of job $i$. Given a sample set, for which the duration $\hat{\sigma}(m_{i,j})$ of each MAP task is measured while they execute, the estimator returns an estimated CDF that characterizes the whole distribution of task times. The estimated CDF is then used to produce a vector of the form:

$$\mathcal{M}_i = [\hat{\sigma}(m_{i,1}), \hat{\sigma}(m_{i,2}), \ldots, \hat{\sigma}(m_{i,j}), \ldots].$$

The MAP phase duration $\theta(\mathcal{M}_i)$ is the sum of the duration of all MAP tasks, discounted by the amount of work done by tasks scheduled in the Training module.

**REDUCE phase size.** Estimating the duration of the REDUCE phase requires a careful approach: the execution time of a REDUCE task can be broken down into (i) SHUFFLE time — that is, the time it takes to move output data from mappers to reducers —, (ii) sort time — because in Hadoop, input data to REDUCE tasks is always sorted —, and (iii) the time it takes to perform the actual work specified by the REDUCE function, that we label execution time. Since

---

3With respect to other jobs with an unknown size.
a REDUCE tasks can be orders of magnitude longer than MAP tasks, we aim at providing an estimate of their duration before their completion. Let $\hat{\sigma}(r_i)$ be the estimate of the execution time of a REDUCE task $r_{i,j}$ of job $i$; as a first approximation, we ignore the SHUFFLE and sort times, and we compute $\hat{\sigma}(r_{i,j})$ as follows:

$$\hat{\sigma}(r_{i,j}) = \frac{\Delta}{p_{i,j}} \forall j \in T_i.$$  

$\Delta$ is a configurable parameter (expressed in seconds) that sets the trade-off between estimation accuracy and speed, and $p_{i,j}$ is the progress done by task $r_{i,j}$ 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. This information is available once all MAP tasks are done producing the intermediate output data, which is materialized locally in each TaskTracker. As such, $p_{i,j}$ embeds the information on the skew of the distribution of REDUCE task times. In other words, $\hat{\sigma}(r_{i,j})$ is a measure of the I/O throughput of a REDUCE task while reading its input data from disk, normalized by the eventual skew in input data sizes. Finally, note that $\Delta$ establishes the maximum amount of time a REDUCE task will remain in execution for size estimation purposes, which constitutes a bound on the training time.

The estimator operates on a sample set $T_i$, for which the duration $\hat{\sigma}(r_{i,j})$ of each REDUCE task is measured. Then, the estimator returns an estimated CDF that characterizes the whole distribution of REDUCE task times, using the available information on the skew of REDUCE task input size distribution. The estimated CDF is then used to produce a vector of the form:

$$R_i = [\hat{\sigma}(r_{i,1}), \hat{\sigma}(r_{i,2}), \ldots, \hat{\sigma}(r_{i,j}), \ldots].$$

The REDUCE phase duration $\theta(R_i)$ is the sum of the duration of all REDUCE tasks, discounted by the amount of work done by tasks scheduled in the Training module.

### 3.3 Job Preemption

The HFSP scheduling discipline 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 commonly used technique to implement preemption for scheduling jobs in Hadoop is that of “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 in [31]. If the runtime of the task is small, then the waiting time is limited, which makes WAIT appealing. While the WAIT method is easy to implement and may provide good results, there are cases – tasks with long runtime – where the delay introduced by this approach may be too high.

In this work, we study the benefits of a more traditional approach to preemption, which we call eager preemption: tasks or jobs can be suspended in favor of other jobs, and resumed when they are awarded resources. Eager preemption requires the implementation of SUSPEND and RESUME primitives. In our implementation, we delegate to the operating system (OS) everything that is related to context switching.

The HFSP scheduler operates on the child Java virtual machine (JVM) that is launched by the parent JVM – namely the TaskTracker – to execute a particular MAP or REDUCE task. The child JVM is effectively a process, which can be suspended and resumed using standard POSIX signals, namely SIGSTOP and SIGCONT. When HFSP suspends a task of a job, the underlying OS eventually 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 requires to introduce a new set of states associated to a Hadoop task, the relative messages for the JobTracker and TaskTracker to communicate eventual state changes and their synchronization.

As discussed Sect. 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 proceeds with the suspension of a task of that job. This step is repeated until all tasks of the new job obtain resources. The selection of which job, among those running in the cluster, to tag for suspension follows a simple rule: the scheduler selects for suspension the tasks of jobs sorted in decreasing order of their size, which reinforces the underlying idea of the HFSP scheduling discipline. In the following, we provide additional considerations.

**Impact on data locality.** Generally, data locality only affects MAP tasks. Instead, with eager preemption, the HFSP scheduler also takes care of data locality for REDUCE tasks: indeed, when a job and its tasks need to be resumed, it is important to do so on the same machines in which they were suspended.

In practice, when the job scheduler decides to allocate resources to a (current) job with some (or all) of its tasks suspended, it proceeds as follows. Upon the periodic heart-beat sent by a given TaskTracker, the job scheduler verifies the presence of suspended tasks of the current job. If the TaskTracker has a free slot and hosts a suspended task for the current job, the job scheduler RESUME such task. If there are no free slots, two conditions may arise: such slots are occupied by tasks of a job smaller or larger than the current one. In the former case, the job scheduler waits for such tasks to terminate; in the latter, the job scheduler SUSPEND tasks of larger jobs, and RESUME tasks of the current job. Essentially, the RESUME operation is similar to that of scheduling a new task of a job, albeit it is given higher priority with respect to the allocation of tasks of the same job on a TaskTracker occupied by a suspended task.

**Finite machine resources.** Suspending tasks has a cost in terms of storage space requirements. If many tasks on a single machine are suspended, context data could use a large fraction of the RAM available on a machine and eventually could also deplete the swap space. Despite this is an extreme case that arises with particular workloads (a large number of jobs arriving in decreasing size), we address it by
defining a set of thresholds (with hysteresis) 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.

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 is dedicated to a comparative analysis between FAIR and HFSP schedulers. We omit the default Hadoop scheduler, FIFO, for the sake of readability. Currently, we implemented HFSP for Hadoop 0.21, which is the stable release of Hadoop, used in production environments. The current release of HFSP, and the additional software we used in our experiments are available as open-source projects.

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 different schedulers we study in this work, in terms of the main performance metric we consider, namely 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, namely Mumak [2]. 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. 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 workloads to analyze the performance of scheduling protocols is a difficult task, that has only recently received some attention [3, 4, 8]. In this work, we use SWIM [3], that comprises workload and data generation tools. 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 a workload synthesized from production-cluster traces collected at Facebook, as done in [31, 9]; that we label FB-dataset. In total, the workload we generate comprises 100 unique jobs. We cluster such jobs into three main classes: small, medium and large jobs. The small job class consists of 53 jobs, of which 75% have a single MAP task, and 25% have 2 MAP tasks. The medium job class consists of 41 jobs. This class includes jobs whose number of MAP tasks ranges from 5 to 500. Half of them have no REDUCE tasks, the remaining jobs have a number of REDUCE tasks ranging from 2 to 100. The large job class consists of 6 jobs, 2 having about 3000 MAP tasks and no REDUCE tasks, 3 whose number of MAP tasks ranges from 700 to 1500 and whose number of REDUCE tasks ranges from 150 to 250, and finally one with 1000 REDUCE tasks and 200 MAP 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. Finally, we remark that the workloads we use on our experiments are I/O intensive only, as explained next.

Individual jobs. The datasets in our possession do not reveal what the actual MapReduce jobs do, to operate on data. However, some related works [9, 8] do a manual classification of such jobs into various categories. We use this information and design a benchmarking framework to generate individual jobs. Essentially, we define a template Java source code for a job, and we set the type of a job by specifying first the data size in input to the job. Hence, we derive the number of HDFS blocks and the number of MAP tasks for the job. In addition, we define the aggregate data size in output from the MAP phase. Given the number of REDUCE tasks, we derive the amount of data each reducer will operate on. In our implementation, the input size of each reducer can follow a variety of distributions, to account for different types of data analysis (e.g., operations on a graph with power-law degree distribution a la PageRank, operations on a Corpus with Zipf-like word frequency, etc.).

The results we present in Sect. 4.2 are obtained for a simple job configuration, because the current version of HFSP implements first-order statistic estimators (cf. Sect. 4.2.1) that assume uniformly distributed task sizes.

Schedulers configuration. Unless otherwise stated, HFSP operates with the delay scheduler (cf. Sect. 3.1) and eager preemption (cf. Sect. 3.2) enabled. HFSP requires a handful of parameters to be configured, which mainly govern the estimator component (cf. Sect. 3.2.1): the maximum number of slots that the top-level scheduler can allocate to the Training module, which we set to all the slots available in the cluster; the sample set size for MAP and REDUCE tasks, which we set to 5; the parameter $\Delta$ to estimate REDUCE task size, which we set to 60 seconds; and finally the confidence parameter $\xi$, which we set equal to 1.

The HFSP parameters described above are appropriate for the workload we use in our experiments. The FAIR scheduler has been configured using default parameters, and uses a single job “pool”.

4.2 Macro benchmarks
First, using the Amazon Cluster, we report the empirical cumulative distribution function (CDF) of sojourn times for FAIR and HFSP, when the cluster executes the workload

\footnote{For the sake of completeness, we also study the impact of the confidence parameter. Our results, as expected, point at slightly larger sojourn times due to training delays. We omit these results due to lack of space.}
Figure 3: ECDFs of sojourn times for the FB-dataset. Jobs are clustered in various classes, based on their sizes. HFSP improves the sojourn times in most cases. In particular, for small jobs, HFSP and FAIR are roughly equivalent, whereas for larger jobs, sojourn times are significantly shorter for HFSP than for FAIR.

In Fig. 3 we cluster results according to job sizes. Our results indicate a general improvement of job sojourn times in favor of HFSP: in particular, sojourn times are considerably smaller for medium and long jobs (cf. Figs. 3(b) and 3(c)). The reason for these results lies in the mix of jobs in the FB-dataset, which is biased toward extremely small jobs. In a cluster with 400 Map slots available, the fair share given to extremely small jobs is greater than their requirements in terms of number of tasks, therefore the behavior of HFSP and FAIR is similar. In addition, very small jobs (with 1-2 Map tasks) are scheduled as soon as a slot becomes free (both under the HFSP and FAIR scheduling disciplines), and therefore their sojourn time depends almost solely on the frequency at which slots free-up and on the cluster state upon job arrival. 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 “sharing” them according to FAIR – towards the completion of the smallest job waiting to be served, as explained intuitively in Sect. 2.

Note that in the large job class, two jobs receive the same treatment with HFSP and FAIR: the first is the largest job in terms of Map tasks in our workload, the other is the largest job in terms of Reduce tasks.

Next, we complement the results discussed above and compute the difference between the sojourn time with FAIR and with HFSP for each individual job, as shown in Figure 4. In our experiments, there is one individual job (with a single Map task, which lasts about 60 s), that exhibit a slightly better sojourn time in FAIR than in HFSP, for a difference of 9 s. We attribute this result to the asynchronous nature of Hadoop, whereby even a small job might have to wait for a slot to become available before being served. The goal of this experiment is related to an experimental validation of the dominance theorem discussed in [13]: we conjecture that even in a multi-processor system HFSP dominates FAIR. However, a formal proof of the dominance theorem is hindered by the initial job size estimation phase, which is a challenge that falls outside the scope of this article.

In summary, HFSP caters both to workloads geared towards “interactive” jobs (that is small jobs) and to a more efficient allocation of cluster resources, which is beneficial to large jobs. We further substantiate the latter claim with a series of experiments executed with Mumak. Our goal is to study scheduling performance with an increasingly resource-hungry workload: indeed, it is clear that the role of a scheduling algorithm is crucial when resources are scarce. We proceed as follows: instead of trying to inflate the FB-dataset with an arbitrarily large number of jobs and large sizes, we instead vary the cluster size, ranging from 10 to 100 nodes (with the same characteristics as for the Amazon-cluster), and use the same workload described above. When the cluster size is scaled down, we increase accordingly the storage space available at each node, to accommodate the data volume used in the workloads.

Fig. 5 reports the mean job sojourn time for FAIR and for HFSP, 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. In other words, for equivalent job sojourn times, the workload we execute

The large number of runs cause the Amazon-Cluster to be prohibitively costly for this experiment.
Robustness of HFSP to job size estimation errors. In this experiment, instead of stressing the estimator component, which is naive and certainly error prone when considering skewed task time distributions, we inject artificial errors on the overall job size estimates reported by the Training module. For this experiment, we use a modified, Map only version of the FB-dataset. This choice, besides for clarity of presentation, stems from the fact that Map and Reduce phases are independent (also for HFSP), and that we thus avoid the possibility for errors to propagate or even cancel-out due to the complex interplay of Map and Reduce scheduling decisions.

In practice, a “wrong” estimate is a random variable uniformly distributed in the range $[\bar{\theta}(1-\alpha), \bar{\theta}(1+\alpha)]$, where $\bar{\theta}$ is the correct job size estimate, and $\alpha \in [0,1]$ is the artificial error we inject. We repeat each experiment 20 times for each value of $\alpha$, to gain statistical confidence in our results.

In addition, we show as a reference the mean sojourn time for experiments using FAIR, which are clearly independent of estimation errors, and the mean sojourn time achieved by a “error-free” HFSP. In our experiments, HFSP is particularly resilient to wrong job size estimates, as the mean sojourn times is slightly affected only for extremely large errors. For the FB-dataset – which exhibit a marked distinction of job classes – a wrong scheduling decision would happen only if a job was to be handled as belonging to the wrong class: for example, a long job should be scheduled before a short job, which would then incur a large sojourn time. In our experiments, “reversals” (that is, jobs are scheduled in different order) appear for jobs in the same class, which clearly have only a modest impact on sojourn times. We carried out several other experiments with different, arbitrary workloads to highlight the most adverse scenarios for HFSP, but obtained similar results to those discussed here.

Impact of data locality. Next, we focus on data locality – which is a fundamental property to guarantee – and measure the fraction of tasks that read data from the local disk of the machine they run on. We compute data locality of both FAIR and HFSP for all the experiments discussed in Sect. 4.2.

FAIR achieves 98% of data locality whereas HFSP always achieves 100% of data locality, for a total of more than 14,000 tasks across all experiments. Clearly, the delay scheduler mechanism [31] 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.

Job preemption disciplines. Finally, we study in detail the various preemption mechanisms we present in Sect. 4.3 with the goal of assessing which is the more suitable option to use, depending on the workload. For this set of experiments, we use a simple, synthetic workload composed of five jobs, and focus solely on Reduce tasks. We simulate, using Mumak, a small cluster of 4 machines with 2 reduce slots each. 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 that of $j_1$.

Fig. 7 illustrates a resource allocation graph: on the y-axis we report the cumulative slot utilization per job, on the x-axis we report time, in minutes. In Fig. 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

---

Figure 5: Impact of cluster size (and hence cluster load) on scheduling performance.

Figure 6: Impact of job size estimation errors on HFSP performance.

Fig. 6 reports the mean sojourn time for different $\alpha$ values.

Clearly, we refer to Map tasks only.
example is about 9 minutes. Instead, in Fig. 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_2 has a larger finish time because 6 of its tasks are killed due to the arrival of jobs j_2 \cdots j_5. We omit the resource allocation graph for the sake of space.

Clearly, it is possible to define alternative scenarios in which HFSP could achieve better results with the WAIT primitive. In general, when task runtimes are short, the WAIT primitive is to be preferred, while when task runtimes are long, eager preemption brings shorter sojourn times. Finally, it is also possible to define pathologic workloads in which a sequence of jobs sorted in decreasing size would arrive sequentially in the system: we performed such experiments as part of our unit testing (e.g. to verify the hysteresis mechanism described in Sect. 4.3), but omit them for the sake of space.

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

Preemption performance. It may be reasonable to argue that the new preemption mechanism we introduce in this work could have an ill effect on job performance and hence on their sojourn time. When one or more tasks of a job are preempted, the memory that they are using can be claimed by other jobs executing new 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 context from disk. As such, the RESUME operation may introduce further delays that contribute to a longer job sojourn time. We remark that such delay is 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 is the main limiting factor that contributes to any additional delays to be added to the sojourn time of a job. Clearly, if the preempted task is not swapped, then such delay becomes negligible.

Finally, we remark that our implementation of preemption may greatly benefit from “sand-boxing” techniques. As part of our future work, we plan to explore sand-boxing to bring HFSP closer to be “production-ready”.

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 J is the set with all the jobs in the system, then job k with weight w_k will receive a fraction \sum_{j \in J} w_j of the resources. This computation can be easily incorporated in the job aging computation (cf. Sect. 3.1) done by the HFSP algorithm.

Job size estimation. We believe reasonable to be skeptical about the ability of such estimate with precision job sizes, especially when considering a broader range of workloads and cluster configurations than those we explored in our experiments. Indeed, task execution time, which contributes to job duration, could be regarded as a highly variable quantity making task time distributions highly skewed.

We remark that in HFSP, the estimator is designed as a pluggable module that could eventually be replaced by more sophisticated estimation techniques, therefore providing more accurate predictions. Furthermore, to the best of our knowledge, task execution times are instead fairly stable, and exhibit a variability that is below 5%, especially for the kind of EC2 instances we used for our experiments with the FB-dataset. In addition, recent works [18] address and greatly mitigate the issue of skew in task processing times with a plug-in module that seamlessly integrate in Hadoop, which can be used in conjunction with HFSP. Moreover, other works [21] 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

\footnote{Our source of information comes from several discussions we had with engineers from the Amazon Web Services EC2 and EMR teams during Hadoop Summit 2012.}

Figure 7: Resource allocation graphs for a simple workload, with and without eager preemption.
also been studied in the case of inaccurate job size information [19]: 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. Scheduling, represents a fundamental problem in computer science and has received a lot of attention in the past. There are many theoretical works that tackle scheduling problems in a multi-processor system – see for instance [12]. 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, and strive at providing 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. Works such as [5, 20], 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, the work in [20] identifies 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 in [20] 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 [31] is a prominent example to which we compare our results. The work in [20] (which is related to [25]) provide more system details on the mechanism used to overcome job starvation with the FAIR scheduler. Many other works [25, 20, 19] focus on resource allocation and strive at achieving fairness across jobs. In [24], the authors 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. The work in [17] addresses the problem of scheduling jobs to meet user-provided deadlines, but assumes job runtime to be an input to the scheduler. Finally, the work that is more closely related to ours is Flex [21], 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 (which we suppose to be updated to recent work [21]) to infer job sizes. In this work, we were not able to perform a comparative analysis of Flex and HFSP: Flex is proprietary and the work in [30] does not give the necessary details to to fully understand its operation.

We shall also consider works that are related to the inner components of HFSP. First, we consider works that tackle the problem of inferring job size: there are numerous recent approaches [28, 29, 11, 21, 27] that provide effective means of estimating job sizes, albeit for some specific application scenarios. HFSP is designed such that the estimator module can be easily plugged with more advanced or tailored solutions, hence such works complement ours. Next, we consider works that study the problem of job preemption: the works that are more closely related to ours are [10, 5]. The authors of [10] present a detailed analysis of the 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, the work in [5] considers job preemption and precisely criticizes an approach based on OS paging, which is relevant to eager preemption. While we agree that un-expected OS paging, due for example to a badly configured cluster which makes extensive use of RAM swapping, is detrimental to job performance, we remark that the preemption primitives we implement in our work are controlled by the scheduler, which uses a threshold mechanism to avoid overloading the OS when facing adverse workloads (cf. Sect. 4).

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 strict preemption primitives. Then, we used standard 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 two standard schedulers that are mostly 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; we will also consider the impact of failures and study in more details the implications of eager preemption from the OS perspective. Finally, we will study the problem of scheduling complex job workflows, that result from the composition of several sub-jobs. The ultimate goal of our work is to contribute HFSP to the Hadoop ecosystem. Currently, the scheduler presented in this work is released as an open source project, and we will work toward a production-ready version of HFSP for its discussion within the Hadoop community.

8. REFERENCES


