HFSP: Bringing Size-Based Scheduling To Hadoop

Mario Pastorelli, Damiano Carra, Matteo Dell'Amico and Pietro Michiardi*†

***Abstract*—Size-based scheduling with aging has been recognized as an effective approach to guarantee fairness and near-optimal system response times. We present HFSP, a scheduler introducing this technique to a real, multi-server, complex and widely used system such as Hadoop.**

Size-based scheduling requires *a priori* job size information, which is not available in Hadoop: HFSP builds such knowledge by estimating it on-line during job execution.

Our experiments, which are based on realistic workloads generated via a standard benchmarking suite, pinpoint at a significant decrease in system response times with respect to the widely used Hadoop Fair scheduler, without impacting the fairness of the scheduler, and show that HFSP is largely tolerant to job size estimation errors.

***Index Terms*—MapReduce, Performance, Data Analysis, Scheduling**

I. Introduction

The advent of large-scale data analytics, fostered by parallel frameworks such as Hadoop [1], [2], Spark [3], [4], and Naiad [5], [6], has created the need to manage the resources of compute clusters operating in a shared, multi-tenant environment. Within the same company, many users *share* the same cluster because this avoids redundancy in physical deployments and in data storage, and may represent enormous cost savings. Initially designed for few 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 heterogeneity is substantiated by recent studies [7]–[9] that analyze a variety of production-level workloads.

An important fact that emerges from previous works is that there exists a stringent need for short system response times. Many operations, such as data exploration, preliminary analyses, and algorithm tuning, often involve *interactivity*, in the sense that there is a human in the loop seeking answers with a trial-and-error process. In addition, workflow schedulers such as Oozie [10] contribute to workload heterogeneity by generating a number of small "orchestration" jobs. At the same time, there are many batch jobs working on big datasets: such jobs are a fundamental part of the workloads, since they transform data into value. Due to the heterogeneity of the workload, it is very important to find the right trade-off in assigning the resources to interactive and batch jobs.

In this work, we address the problem of job *scheduling*, that is how to allocate the resources of a cluster to a number of concurrent jobs, and focus on Hadoop [1], the most widely adopted open-source implementation of MapReduce. Currently, there are mainly two different strategies used to schedule jobs in a cluster. The first strategy is to split the cluster resources equally among all the running jobs. A remarkable example of this strategy is the Hadoop Fair Scheduler [11], [12]. While this strategy preserves fairness among jobs, when the system is overloaded, it may *increase the response times* of the jobs. The second strategy is to serve one job at a time, thus avoiding the resource splitting. An example of this strategy is First-In-First-Out (FIFO), in which the job that arrived first is served first. The problem with this strategy is that, being blind to job size, the scheduling choices lead inevitably to poor performance: small jobs may find large jobs in the queue, thus they may incur in response times that are disproportionate to their size. As a consequence, the interactivity is difficult to obtain. Both strategies have drawbacks that prevent them from being used directly in production without precautions. Commonly, a *manual configuration* of both the scheduler and the system parameters is required to overcome such drawbacks. This involves the *manual setup* of a number of "pools" to divide the resources to different job categories, and the fine-tuning of the parameters governing the resource allocation. This process is tedious, error prone, and cannot adapt easily to changes in the workload composition and cluster configuration. In addition, it is often the case for clusters to be over-dimensioned [7]: this simplifies resource allocation (with abundance, managing resources is less critical), but has the downside of costly deployments and maintenance for resources that are often left unused.

In this paper we present the design of a new scheduling protocol that caters both to a fair and efficient utilization of cluster resources, while striving to achieve short response times. Our approach satisfies *both* the interactivity requirements of small jobs and the performance requirements of large jobs, which can thus coexist in a cluster without requiring manual setups and complex tuning: our technique automatically adapts to resources and workload dynamics.

Our solution implements a *size-based*, preemptive scheduling discipline. The scheduler allocates cluster resources such that job size information – which is *not available a priori* – is inferred while the job makes progress toward its completion. Scheduling decisions use the concept of *virtual time*, in which jobs make progress according to an *aging* function: cluster

*M. Pastorelli is with Teralytics AG, Switzerland; this work has been done while he was at EURECOM. Email: pastorelli.mario@gmail.com

D. Carra is with the Computer Science Department, University of Verona, Verona, Italy. E-mail: damiano.carra@univr.it

M. Dell'Amico is with Symantec Research Labs, France; this work has been done while he was at EURECOM. Email: della@linux.it

P. Michiardi is with the Networking and Security Department, EURECOM, France. Email: michiard@eurecom.fr

†Manuscript received July 7, 2014; revised November 26, 2014.

resources are "focused" on jobs according to their priority, computed through aging. This ensures that neither small nor large jobs suffer from starvation. The outcome of our work materializes as a full-fledged scheduler implementation that integrates seamlessly in Hadoop: we called our scheduler HFSP, to acknowledge an influential work [13] in the size-based scheduling literature.

The contribution of our work can be summarized as follows:

- We design and implement the system architecture of HFSP (Section III), including a (pluggable) component to estimate job sizes and a dynamic resource allocation mechanism that strives at efficient cluster utilization. HFSP is available as an open-source project.[1]
- Our scheduling discipline is based on the concepts of virtual time and job aging. These techniques are conceived to operate in a multi-server system, with tolerance to failures, scale-out upgrades, and multi-phase jobs – a peculiarity of MapReduce.
- We reason about the implications of job sizes not being available *a priori*, both from an abstract (Section III) and from an experimental (Section IV) point of view. Our results indicate that size-based scheduling is a realistic option for Hadoop clusters, because HFSP sustains even rough approximations of job sizes.
- We perform an extensive experimental campaign, where we compare the HFSP scheduler to a prominent scheduler used in production-level Hadoop deployments: the Hadoop Fair Scheduler. For the experiments, we use PigMix, a standard benchmarking suite that performs real data analysis jobs. Our results show that HFSP represents a sensible choice for a variety of workloads, catering to fairness, interactivity and efficiency requirements.

II. Background: Traditional Scheduling

First Come First Serve (FCFS) and Processor Sharing (PS) are arguably the two most simple and ubiquitous scheduling disciplines in use in many systems; for instance, *FIFO* and *Fair* are two schedulers for Hadoop, the first inspired by FCFS, and the second by PS. In FCFS, jobs are scheduled in the order of their submission, while in PS resources are divided equally so that each active job keeps progressing. In loaded systems, these disciplines have severe shortcomings: in FCFS, large running jobs can delay significantly small ones; in PS, each additional job delays the completion of *all* the others.

In order to improve the performance of the system in terms of delay, it is important to consider the size of the jobs. Size-based scheduling adopts the idea of giving priority to small jobs: as such, they will not be slowed down by large ones. The Shortest Remaining Processing Time (SRPT) policy, which prioritizes jobs that need the least amount of work to complete, is the one that minimizes the mean *response time* (or *sojourn time*), that is the time that passes between a job submission and its completion [14].

Figure 1 provides an example that compares PS to SRPT: in this case, two small jobs – j_2 and j_3 – are submitted while a large job j_1 is running. While in PS the three jobs run

[1]https://github.com/bigfootproject/hfsp

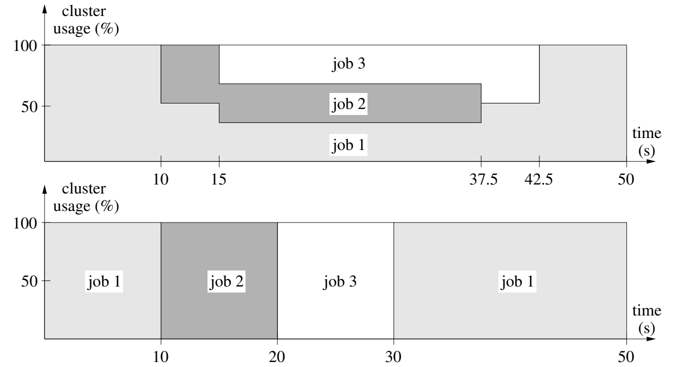


Fig. 1: Comparison between PS (top) and SRPT (bottom).

(slowly) in parallel, in a size-based discipline j_1 is *preempted*: the result is that j_2 and j_3 complete earlier. Like most size-based scheduling techniques, SRPT temporarily suspends the progress of lower-priority jobs; fortunately, this is not a problem in a batch system like Hadoop, for which results are usable only *after* the job is completed.

While policies like SRPT improve mean response time, they may incur in *starvation*: if small jobs are continuously submitted, large ones may never receive service [15]–[17]; this results in job mistreatment. To avoid starvation, a common solution is to perform *job aging*. With job aging, the system decreases virtually the size of jobs waiting in the queue, and keeps them sorted according to their virtual size, serving the one with the current smaller virtual size. Job aging assures that, as time goes by, each job is eventually scheduled; in the abstract case that job size is perfectly known *a priori*, the FSP discipline [13] exploits aging to provide a strong *dominance* fairness guarantee: *no job completes in FSP later than it would in PS*. FSP also guarantees excellent results in terms of both job response time and fairness when job sizes are not known exactly [18]; for these reasons, the design of HFSP is guided by the abstract ideas beyond FSP.

III. The Hadoop Fair Sojourn Protocol

The *Hadoop Fair Sojourn Protocol* (HFSP) is a size-based scheduler with aging for Hadoop. Implementing HFSP raises a number of challenges: a few come from MapReduce itself – *e.g.,*, the fact that a job is composed by tasks – while others come from the size-based nature of the scheduler in a context where the size of the jobs is not known *a priori*. In this section we describe each challenge and the proposed solutions.

Jobs: In MapReduce, jobs are scheduled at the granularity of *tasks*, and they consist of two separate phases, called MAP and REDUCE. We evaluate job sizes by running a subset of *sample* tasks for each job; however, REDUCE tasks can only be launched only after the MAP phase is complete. Our scheduler thus splits logically the job in the two phases and treats them independently; therefore the scheduler considers the job as composed by two parts with different sizes, one for the MAP and the other for the REDUCE phase. When a resource is available for scheduling a MAP (resp. REDUCE) task, the scheduler sorts jobs based on their virtual MAP (resp.

REDUCE) sizes, and grants the resource to the job with the smallest size for that phase.

Estimated and Virtual Size: The size of each phase, to which we will refer as *real size*, is unknown until the phase itself is complete. Our scheduler, therefore, works using an *estimated size*: starting from this estimate, the scheduler applies job aging, *i.e.,* it computes the *virtual size*, based on the time spent by the job in the waiting queue. The estimated and the virtual sizes are calculated by two different modules: the *estimation module*, that outputs the *estimated size*, and the *aging module*, that takes in input the estimated size and applies an aging function. In the following, we describe them.

A. The Estimation Module

The role of the estimation module is to assign a size to a job phase such that, given two jobs, the scheduler can discriminate the smallest one for that phase. When a new job is submitted, the module assigns for each phase an *initial size* S_i, which is based on the number of its tasks. The initial size is necessary to quickly infer job priorities. A more accurate estimate is done immediately after the job submission, through a *training stage*: in such a stage, a subset of t tasks, called the *training tasks*, is executed, and their execution time is used to update S_i to a *final estimated size* S_f. Choosing t induces the following trade-off: a small value reduces the time spent in the training stage, at the expense of inaccurate estimates; a large value increases the estimation accuracy, but results in a longer training stage. As we will show in Section III-C, our scheduler is designed to work with rough estimates, therefore a small t is sufficient for obtaining good performances.

Tiny Jobs: Every job phase with less than t tasks is considered as *tiny*: in this case, HFSP sets $S_f = 0$ to grant them the highest priority. Tiny jobs use a negligible fraction of cluster resources: giving them the highest priority marginally affects other jobs. Note that the virtual size of all other jobs is constantly updated, therefore every job will be eventually scheduled, even if tiny jobs are constantly submitted.

Initial Size: The initial size of a job phase with n tasks is set to $S_i = n \cdot \xi \cdot \overline{s}$ where $\overline{s}$ is the average task size computed so far by the system considering all the jobs that have already completed; $\xi \in [1, \infty)$ is a tunable parameter that represents the propensity of the system to schedule jobs that have not completed the training stage yet. If $\xi = 1$, new jobs are scheduled quite aggressively based on the initial estimate, with the possible drawback of scheduling a particularly large job too early. Setting $\xi > 1$ mitigates this problem, but might result in increased response times. Finally, if the cluster does not have a past job execution history, the scheduler sets $S_i = s_0$, where $s_0 > 0$ is an arbitrary constant, until the first job completes.

Final Size: When a job phase completes its training stage, the estimation module notifies the aging module that it is ready to update the size of that phase for that job. The final size is

$$S_f = \tilde{s} \cdot \left[(n-t) + \sum_{k=1}^{t}(1-p_k)\right],$$

where $(n-t)$ is the number of tasks of a job phase that still need to be run, *i.e.,* the total number of tasks minus the training tasks. The definitions of $\tilde{s}$ and p_k are more subtle. As observed in prior works [11], [19], MAP task execution times are generally stable and short, whereas the distribution of REDUCE task execution times is skewed. Therefore, $\tilde{s}$ is the average size of the t tasks that either *i)* completed in the training stage (and thus have an individual execution time s_k), or *ii)* that made enough progress toward their completion, which is determined by a timeout Δ (60 s in our experiments). The progress term p_k, which is measured by Hadoop *counters*, indicates the percentage of input records processed by a training task t_k. More formally, we have

$$\tilde{s} = \frac{1}{t}\sum_{k=1}^{t}\tilde{s}_k\,,$$

where

$$\tilde{s}_k = \begin{cases} s_k \text{ , if training task } t_k \text{ completes within } \Delta \\ \frac{\Delta}{p_k} \text{ , otherwise} \end{cases}.$$

This estimation can be erroneus due to *stragglers* [9], *i.e.,* tasks that take longer to complete than others in the same job. Such a phenomenon would lead to an over-estimation of job size. Fortunately, as opposed to under-estimations, over-estimations only have modest impact on the performance of size-based schedulers [18].

In HFSP, once S_f is set for the first time, after the training stage, it will not be updated, despite additional information on task execution could be used to refine the first estimate. Indeed, continuous updates to S_f may result in a problem that we call "flapping", which leads to poor scheduling performance: Figure 2 shows an example of this effect. If S_f estimates are not updated, there are two possible cases: the first estimate leads to *i)* a correct scheduling decision, or *ii)* an "inversion" between two jobs. In Figure 2, a correct scheduling decision implies that J_2 completes at time 2 and J_1 completes at time 5, while an inversion implies that J_1 completes at time 3 and J_2 completes at time 5. The mean *response time* in the first and second case are 3.5 and 4 respectively. The bottom scheme in Figure 2 illustrates the effects of a continuous update to S_f, which leads to flapping. In this case, the response times are 5 for J_1 and 4 for J_2 resulting in a mean response time of 4.5.

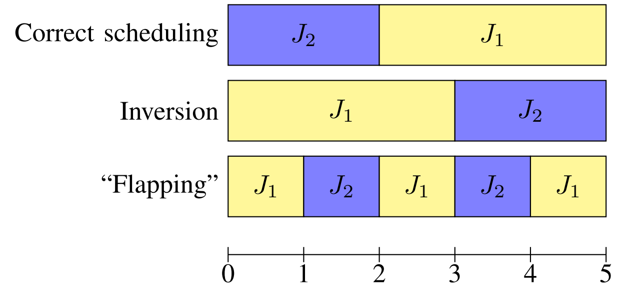
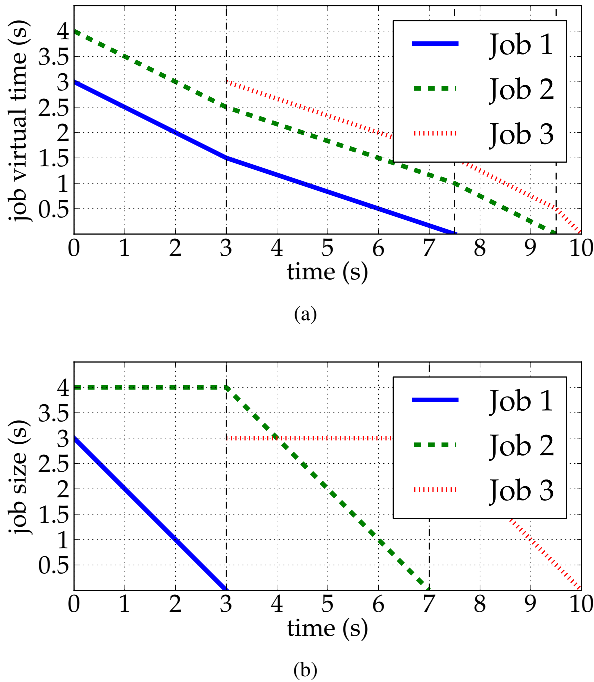


Fig. 2: Example of continuous updates of job sizes.

Clearly, jobs with similar sizes exacerbate the phenomenon we illustrate above. Since estimation errors for jobs with similar sizes is likely to occur, as we are going to show in Section III-D, HFSP avoids "flapping" problems using a unique estimate for S_f.



(a)

(b)

Fig. 3: Virtual time and job size progress in HFSP.

B. The Aging Module

The aging module takes as input the estimated sizes to compute *virtual sizes*. The use of virtual sizes is a technique applied in many practical implementations of well-known schedulers [13], [20], [21]: it consists in keeping track of the amount of the remaining work for each job phase in a *virtual "fair" system*, and update it every time the scheduler is called. The result is that, even if the job doesn't receive resources and thus its real size does not decrease, in the virtual system the job virtual size slowly decreases with time.

Job aging avoids starvation, achieves fairness, and requires minimal computational load, since the virtual size does not incur in costly updates [13], [20]. Figure 3 shows an example of how the job virtual size is decreased and how that affects the scheduling policy. We recall that HFSP schedules jobs with the smallest virtual size first. In Figure 3a, we show job virtual sizes and in Figure 3b the job real sizes (sizes are normalized with respect to the service rate). At time 0 there are two jobs in the queue, Job 1 with size 3 and Job 2 with size 4. Note that at the beginning, the virtual size corresponds to the real job size. The scheduling policy chooses the job with the smallest virtual size, that is Job 1, and grants it all the resources. At time 3, Job 3 enters the job queue and Job 1 completes. Job 3 has a smaller real size than Job 2, but a bigger virtual size; this is due to the fact that, while Job 1 was being served, the virtual times of both Job 1 and Job 2 have been decreased. When Job 1 finishes, Job 2 has virtual size 2.5 while Job 3 has virtual size equal to its current real size, that is 3.

Virtual Cluster: The part of the aging module that implements the virtual system to simulate processor sharing is called *Virtual Cluster*. Jobs are scheduled at the granularity of tasks, thus the virtual cluster simulates the same resources available in the real cluster: it has the same number of "machines" and the same configuration of (MAP or REDUCE) resources per machine. When the resources change in the real cluster, the scheduler notifies it to the aging module that updates the virtual cluster resources. We simulate a *Max-Min Fairness* criterion to take into account jobs that require *less* compute resources than their fair share (*i.e.,* $1/n$-th of the resources if there are n active jobs): a round-robin mechanism allocates virtual cluster resources, redistributing the fair share not used by small jobs (jobs that need less than $1/n$-th of the resources) among the other jobs.

Job priorities and elaborate configurations can be integrated in HFSP by modifying the virtual cluster component: for example, the PSBS discipline [22] implements job priorities by simulating a *generalized* PS allowing for job weights in the virtual time; analogously, HFSP can take a similar approach.

Estimated Size Update: When the estimated size of a job is updated from the initial estimation S_i to the final estimation S_f, the estimation module alerts the aging module to update the job phase size to the new value. After the update, the aging module runs the scheduler on the virtual cluster to reassign the virtual resources.

Failures: The aging module is robust with respect to failures. The same technique used to support cluster size updates is used to update the resources available when a failure occurs; once Hadoop detects the failure, job aging will be slower. Conversely, adding nodes will result in faster job aging reflecting the fact that with more resources the cluster can do more work.

Manual Priority and QoS: Our scheduler does not currently implement the concept of different job priorities assigned by the user who submitted the job; however, the aging module can be easily modified to simulate a Generalized Processor Sharing discipline, leading to a scheduling policy analogous to Weighted Fair Queuing [23]. A simple approach is to consider the user assigned priority as a *speed modifier* to the aging of the job virtual sizes: when the aging module decreases the virtual size of the job, it subtracts to the job virtual size the virtual work done multiplied by the job modifier. A job modifier bigger (resp. smaller) than 1 speeds up (resp. slows down) the aging of a job.

C. The Scheduling Policy

In this section we describe how the estimation and the aging modules coexist to create a Hadoop scheduler that strives to be both efficient and fair.

Job Submission: Figure 4 shows the lifetime of a job in HFSP, from its submission to its completion and removal from the job queue. When a job is submitted, for each phase of the job, the scheduler asks to the estimation module if that phase is tiny. If the answer is affirmative, the scheduler assigns $S_f = 0$, meaning that the job must be scheduled as soon as possible. Otherwise, the scheduler starts the training stage and sets the virtual time to the initial size S_i given by the estimator module. Periodically, the scheduler asks to the estimation module if it has completed its training stage, and, if the answer is positive, it notifies the aging module to update the virtual size of that job and removes the job from the training stage.

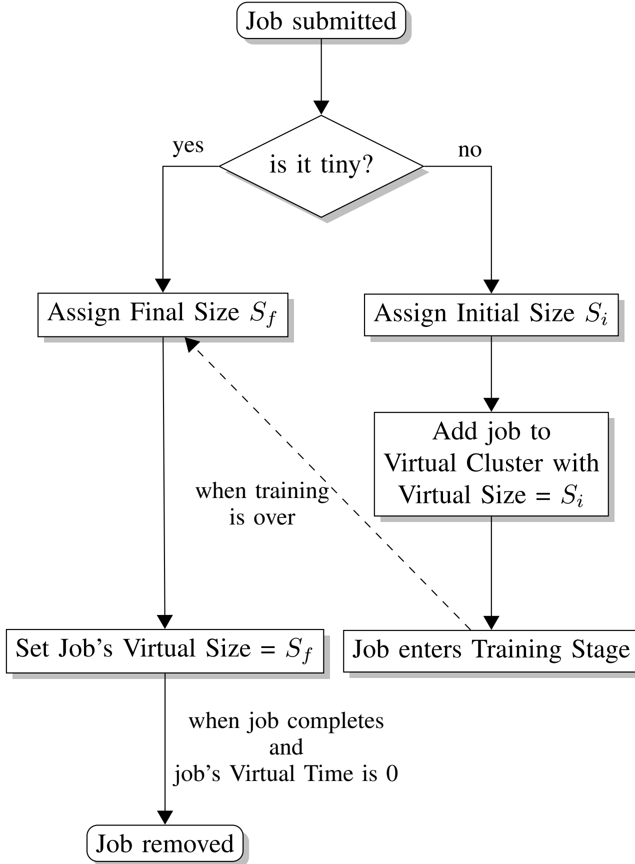


Fig. 4: Job lifetime in HFSP

Priority To The Training Stage: The training stage is important because, as discussed in Section III-A, the initial size S_i is imprecise, compared to the final size S_f. Completing the training stage as soon as possible is fundamental for an efficient scheduling policy. There are two strategies that are used by the scheduler to speed up the training stage: the first strategy is to set a low number of training tasks t, as discussed in Section III-A; the second strategy is to give priority to the training tasks across jobs – up to a threshold of $T \in [0, T_{max}]$ where T_{max} is the total number of resources in the cluster. Such threshold avoids starvation of "regular" jobs in case of a bursty job arrival pattern. When a resource is free and there are jobs in the training stage, the scheduler assigns the resource to a training task independently from its job position in the job queue. In other words, training tasks have the highest priority. Conversely, after a job has received enough resources for its training tasks, it can still obtain resources by competing with other jobs in the queue.

Virtual Time Update: When a job phase completes its training stage, the scheduler asks to the estimation module the final size S_f and notifies the aging module to update the virtual size accordingly. This operation can potentially change the order of the job execution. The scheduler should consider the new priority and grant resources to that job, if such job is the smallest one in the queue. Unfortunately, in Hadoop MapReduce the procedure to free resources that are used by the tasks, also known as *task preemption*, can waste work. The default strategy used by HFSP is to wait for the resources to be released by the working tasks. Section III-E describes the preemption strategies implemented in HFSP and their implications.

Algorithm 1 HFSP resource scheduling for a job phase.

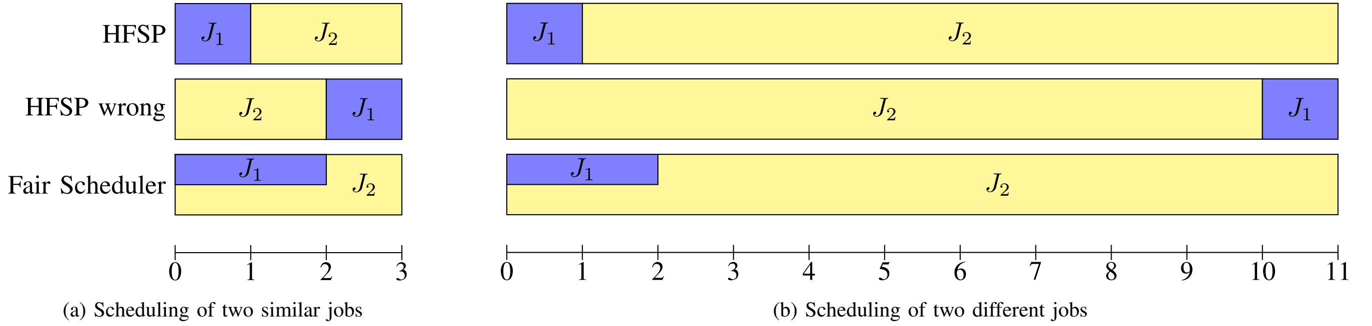
```
function ASSIGNPHASETASKS(resources)
    for all resource s ∈ resources do
        if ∃ (Job in training stage) and T_curr < T then
            job ← select job to train with smallest initial
                    virtual size
            ASSIGN(s, job)
            T_curr ← T_curr + 1
        else
            job ← select job with smallest virtual time
            ASSIGN(s, job)
        end if
    end for
end function

function ASSIGN(resource, job)
    task ← select task with lower ID from job
    assign task to resource
end function

function RELEASERESOURCE(task)
    if task is a training task then
        T_curr ← T_curr − 1
    end if
end function
```

Data locality: For performance reasons, it is important to make sure that MAP tasks work on local data. To this aim, we use the *delay scheduling* strategy [11], 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.

Scheduling Algorithm: HFSP scheduling – which is invoked every time a MapReduce slave claims work to do to the MapReduce master – behaves as described by Algorithm 1. The procedure `AssignPhaseTasks` is responsible for assigning tasks for a certain phase. First, it checks if there are jobs in training stage for that phase. If there are any, and the number of current resources used for training tasks T_{curr} is smaller or equal than T, the scheduler assigns the resource to the first training task of the smallest job. Otherwise, the scheduler assigns the resource to the job with the smallest virtual time. When a task finishes its work, the procedure `releaseResource` is called. If the task is a training task, then the number T_{curr} of training slots in use is decreased by one.

D. Impact of Estimation Errors

In this section we describe the impact of an erroneous scheduling, *i.e.,* giving resources to the wrong job, and how HFSP handles estimation errors.



(a) Scheduling of two similar jobs

(b) Scheduling of two different jobs

Fig. 5: Illustrative examples of the impact of estimation errors on HFSP.

The estimation module outputs an estimated size S_f, which we refer to in the following as simply S. The error e is the ratio between the estimated size S and the real size R, therefore $S = R \cdot e$. We now consider two kinds of possible errors: the job size can be under-estimated ($e < 1$) or over-estimated ($e > 1$). Both errors have impact on the scheduler; however, the case of under-estimation is more problematic. Indeed, a job whose size is over-estimated obtains a lower priority: this impacts only that job, delaying the moment in which it obtains the resources to work. Even if the job enters the queue with a lower priority, *i.e.,* a large virtual size, the aging function will raise its priority as the time goes by, and the job will obtain eventually the resources. Instead, job under-estimation increases the priority of a job, and this can potentially affect all the other jobs in the queue. The aging function plays a crucial role in making HFSP tolerant to estimations errors. Other size-based scheduling policies, like SRPT, are more affected by estimation errors [18].

The study of the estimation error for a single job is not enough to understand the impact of the errors on our scheduling policy. Indeed, we need to consider the estimation errors of all the jobs in the queue and how their interplay can ultimately lead the scheduler to take wrong scheduling decisions. We exemplify this problem for two jobs: let us denote the size of an arbitrary phase of a job J_k as R_k, and its estimated size as $S_k = R_k \cdot e_k$, where e_k expresses the estimation error. Two jobs J_1 and J_2 with $R_1 < R_2$ are scheduled incorrectly (*i.e.,* J_2 is given priority over J_1) if $S_1 > S_2$, *i.e.,* $e_1/e_2 > S_2/S_1$. If J_1 and J_2 are similar in size, even a small e_1/e_2 ratio may invert the priorities of these jobs; if S_2/S_1 is instead large (*e.g.,* because the two sizes differ by orders of magnitude), then also e_1/e_2 must be analogously large. This works perfectly with a size-based scheduler because if two jobs are similar, inverting the priorities of the two has only a small impact on the overall performance. Instead, if two jobs are very different, switching them can lead to poor performance because the job with highest priority has to wait for an amount of time that may be much larger than its size. This is, however, unlikely to happen, because it requires that e_1 and/or e_2 are very far from 1. In other words, estimation errors should be very large for switching very different jobs.

Figure 5a and Figure 5b exemplify how a wrong scheduling can affect HFSP. Each figure has three cases: the first case, on top, is the correct HFSP policy; the second case, in the middle, shows what happens when HFSP gives priority to the wrong job and the third case, at the bottom, shows an ideal "fair" scheduler adopting a processor-sharing scheduling policy.

Figure 5a shows what happens when the two jobs have similar sizes, that are 1 and 2 seconds. If the estimation module outputs estimated sizes S_1 and S_2 such that $S_1 < S_2$, HFSP schedules first J_1 and then J_2 and the resulting mean response time is $\frac{1+3}{2} = 2$. On the contrary, if the estimation is wrong and $S_1 > S_2$, HFSP will schedule first J_2 and then J_1. While this scheduling choice is incorrect, the resulting mean response time is $\frac{2+3}{2} = 2.5$. While a response time of 2.5 is not optimal, it matches the mean response time obtained with Processor Sharing. We conclude that if two jobs are similar, switching them by giving priority to the bigger one does not affect heavily a metric such as the mean response time.

Figure 5b illustrates what happens when two jobs have sizes very different, in the example 1 and 10 seconds. As in the previous example, HFSP schedules J_1 and J_2 based on the estimator output. Here, however, the difference in the mean response time is large: the mean response time is $\frac{1+11}{2} = 6$ when J_1 is scheduled first and $\frac{10+11}{2} = 10.5$ in the other case. The wrong scheduling choice does not only lead to an almost doubled mean response time: it also heavily penalizes J_1, which has to wait 10 times its size before having any resource granted. The difference between the two scheduling choices is even more clear when they are compared to the Processor Sharing, that has a mean response time of $\frac{2+11}{2} = 6.5$. When two jobs are very different, a wrong scheduling choice can lead to very poor performance of HFSP. This situation, however, is much less likely to happen than the former, requiring a value of $e_1 > 10e_2$.

The t and Δ configuration parameters introduced in Section III-A play a role in the trade-off between the precision of the estimation module and the amount of time needed to obtain an estimation. In experiments not included due to space limitations, we have found that: *1)* our default values ($t = 5$, $\Delta = 60s$) hit a good point in the trade-off; *2)* the impact of these parameters on the scheduler's performance is limited. Indeed, simulation-based studies show that size-based schedulers perform well when based on estimated job sizes as soon as the correlation between job size and its estimation is at 0.2 or more. Our experimental findings show that such

levels of precisions are easily attainable in our scenario; in the case of very unstable clusters where this is not obtained, precision can be improved by raising t and/or Δ as needed.

In summary, scheduling errors – if caused by estimations that are not very far from real job sizes – yield response times similar to those of processor sharing policies, especially under heavy cluster load. Estimation errors lead to very bad scheduling decisions only when job sizes are very different from their estimations. As we show in our experimental analysis, such errors rarely occur with realistic workloads.

E. Task Preemption

HFSP is a preemptive scheduler: jobs with higher priority should be granted the resources allocated to jobs with lower priorities. In Hadoop, the main technique to implement preemption is by *killing* tasks. Clearly, this strategy is not optimal, because it wastes work, including CPU and I/O. Other works have focused on mitigating the impact of KILL on other MapReduce schedulers [24]. Alternatively, it is possible to WAIT for a running task to complete, as done by Zaharia *et al.* [11]. If the runtime of the task is small, then the waiting time is limited, which makes WAIT appealing. This is generally the case of MAP tasks but not of REDUCE tasks, which can potentially run for a long time. HFSP supports both KILL and WAIT and by default it is configured to use WAIT. In this section we describe how HFSP works when the KILL preemption is enabled.

Task Selection: Preempting a job in MapReduce means preempting some or all of its running tasks. It may happen that not all the tasks of a job have to be preempted. In this case, it is very important to pick the right tasks to preempt to minimize the impact of KILL. HFSP chooses to preempt the "*youngest*" tasks, *i.e.,* those that have been launched last, for three practical reasons: *i)* old tasks are the most likely ones to finish first, freeing resources to other tasks; *ii)* killing young tasks wastes less work; *iii)* young tasks are likely to have smaller memory footprints, resulting in lower cleanup overheads due to *e.g.,* purging temporary data.

When Preemption Occurs: Preemption may occur for different reasons. First, the training tasks always have priority over non-training tasks; therefore, training tasks can preempt other tasks even if they are part of a job with higher priority than theirs. This makes the training phase faster to complete and, considering that the number of training tasks is bounded, does not significantly affect the runtime of other jobs. Tasks that complete in the training stage lose their "status" of training task and, consequently, can be preempted as well.
Newly submitted jobs can, of course, preempt running tasks when the virtual size of the new job is smaller than the virtual size of the job that is served.
Task preemption may also happen when the estimation module updates the size of a job, from S_i to S_f. This can move the job to a new position in the job queue.
Aging can also be responsible for task preemption: as we saw in Section III-B, since we implement max-min fairness, jobs may have different degrees of parallelism, *e.g.,* small jobs may require less than they fair share; this results in a different aging, that may change the order in the execution list.
Finally, the last reason for task preemption is when the cluster resources shrink, *e.g.,* after a failure. In this case, HFSP grants "lost" resources to jobs with higher priority using preemption.

New Preemption Primitives: Neither KILL or WAIT are optimal: as we are going to see in Section IV-D3, waiting for tasks to complete penalizes fairness, while killing tasks wastes work. Better preemption primitives for MapReduce have been proposed [25], [26]: part of our future work to improve HFSP will include testing those solutions.

IV. EXPERIMENTS

We provide an extensive evaluation of the performance of HFSP, comparing it against the default scheduler in current Hadoop distributions (*e.g.,* Cloudera), namely the Fair scheduler [11], [12]. We do not include a comparison with the FIFO scheduler available in stock Hadoop: in our experiments, FIFO is drastically outperformed by all other schedulers, in accordance with results in the literature [18].

In the following, experiments are divided in *macro benchmarks*, in which we evaluate the performance in terms of fairness and response times of each scheduler, and *micro benchmarks*, in which we focus on specific details of HFSP.

A. Experimental Setup

We run our experiments on a cluster composed of 20 TASKTRACKER worker machines with 4 CPUs and 8 GB of RAM each. We configure Hadoop according to current best practices [27], [28]: the HDFS block size is 128 MB, with replication factor 3; each TASKTRACKER has 2 map slots with 1 GB of RAM each and 1 reduce slots with 2 GB of RAM. The `slowstart` factor is configured to start the REDUCE phase for a job when 80% of its MAP tasks are completed.

HFSP operates with the following parameters: the number of tasks for the training stage is set to $t = 5$ tasks; the timeout to estimate task size is set to $\Delta = 60$ seconds; we schedule aggressively jobs that are in the training stage, setting $\xi = 1$ and $T = 10$ slots for both MAP and REDUCE phases. The Fair scheduler has been configured with a single job pool, thus all jobs have the same priority.

We generate workloads using PigMix [29], a benchmarking suite used to test the performance of Apache Pig releases. PigMix is appealing because, much like its standard counterparts for traditional DB systems such as TPC [30], it *both* generates realistic datasets and defines queries inspired by real-world data analysis tasks. PigMix contains both Pig scripts and native MapReduce implementation of the scripts. Since Pig has generally an overhead over the native implementations of the same job, in our experiments we use the native implementations. For illustrative purposes, we include a sample Pig Latin query that is included in our workloads next:

```
REGISTER pigperf.jar;
A = LOAD 'pigmix/page_views' USING ...
    AS (user, action, timespent,
    query_term, ip_addr, timestamp,
    estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'pigmix/power_users_samples' USING ...
```

```
    AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name, phone;
C = JOIN B BY user LEFT OUTER,
      beta BY name PARALLEL 20;
```

The above Pig job performs a "projection" on two input datasets, and "joins" the results.

Job arrival follows a Poisson process, and jobs are generated by choosing uniformly at random a query between the 17 defined in PigMix. The amount of work each job has to do depends on the size of the data it operates on. For this reason, we generate four different datasets of sizes respectively 1 GB, 10 GB, 100 GB and 1 TB. For simplicity, we refer to these datasets as to *bins* – see the first two columns of Table I. The third column of the table shows ranges of number of MAP tasks for each bin (PigMix queries operate on different subsets of the input datasets, which result in a variable number of MAP/REDUCE tasks).

We randomly map each job to one of the four available datasets: this assignment follows three different probability distributions – see the last three columns of Table I. The overall composition of the jobs therefore defines three workloads, which we label according to the following scheme:

- **DEV:** this workload is indicative of a "development" environment, whereby users rapidly submit several small jobs to build their data analysis tasks, together with jobs that operate on larger datasets. This workload is inspired by the Facebook 2009 trace observed by Chen *et al.* [19]; the mean interval between job arrivals is $\mu = 30\ s$.
- **TEST:** this workload represents a "test" environment, whereby users evaluate and test their data analysis tasks on a rather uniform range of dataset sizes, with 20% of the jobs using a large dataset. This workload is inspired by the Microsoft 2011 traces as described by Appuswamy *et al.* [31]; the mean interval between jobs is $\mu = 60\ s$.
- **PROD:** this workload is representative of a "production" environment, whereby data analysis tasks operate predominantly on large datasets. The mean interval between jobs is $\mu = 60\ s$.

The values of μ control the system load: in results not included due to space constraints, we found that μ amplifies the difference between schedulers without altering it qualitatively; when μ is low the cluster is likely to have enough resources to satisfy all demands: in that case, the scheduling policy does not make a difference; conversely, as soon as the scheduler is presented with a wider set of possibilities because more jobs are pending, the scheduling policy matters [32].

In this work, each workload is composed of 100 jobs, and both HFSP and Fair have been evaluated using the same jobs, the same inputs and the same submission schedule. For each workload, we run five experiments using different seeds for the random assignments (query selection and dataset selection), to improve the statistical confidence in our results.

Additional results – not included for lack of space – obtained on different platforms (Amazon EC2 and the Hadoop Mumak emulator), and with different workloads (synthetic traces generated by SWIM [19]), confirm the ones shown here; they are available in a technical report [32]. A larger set of workloads, which evaluate size-based schedulers on a larger variety of synthetic and real workloads, has been evaluated through simulation [18]; the results confirm the effectiveness of size-based disciplines in a large variety of cases.

TABLE I: Summary of the workloads used in our experiments.

Bin	Dataset Size	Averag. num. MAP Tasks	Workload		
			DEV	TEST	PROD
1	1 GB	< 5	65%	30%	0%
2	10 GB	$10 - 50$	20%	40%	10%
3	100 GB	$50 - 150$	10%	10%	60%
4	1 TB	> 150	5%	20%	30%

B. Metrics

In our experiments we use two main metrics to evaluate the scheduler performance. The first metric is the (job) *response time*, *i.e.,* the time that passes between a job's submission and its completion. This metric is widely used as metric of *efficiency* and *responsiveness* in the literature on scheduling. The second metric focuses on the notion of *fairness*. Even if there are different ways to measure fairness [33], a common approach is to consider the *slowdown*, *i.e.,* the ratio between a job's response time and its size. In this work we focus on the *per-job slowdown*, to analyze whether jobs are treated unfairly, which translates into very high slowdown values.

Note that, to compute the per-job slowdown, it is necessary to know the "real" size of a job: to obtain such information, we execute each job in isolation, and measure their runtime.

The Fair scheduler is inspired by the PS scheduling policy, which should ensure a fair treatment to all jobs, since it assigns equally the resources to each of them. For the purposes of this work, we will therefore consider as fair a scheduling policy for which the highest slowdown values are in line or better than those obtained by the Fair scheduler.

C. Macro Benchmarks

In this section we present the aggregate results of our experiments for response times and per-job slowdown, across all schedulers we examine.

1) Response Times: Figure 6 shows the *mean* response times for all workloads we evaluate: the mean response times are indicative of system responsiveness, and lower values are best. Overall, HFSP is substantially more responsive than the Fair scheduler, a claim that we confirm also by inspecting the full distribution of response times, later in this Section. As shown in Figure 6, the mean sojourn time for HFSP is 34%, 26%, 33% lower for the DEV, TEST, and PROD workload respectively. It is important to note that a responsive system does not only cater to a "development" workload, which requires *interactivity*, but also to more "heavy" workloads, that require an *efficient* utilization of resources. Thus, in summary, HFSP is capable of absorbing a wide range of workloads, with no manual (and static) configuration of resource pools, and only a handful parameters to set. Globally, our results can also be interpreted in another key: a system using HFSP can deliver the same responsiveness as one running other schedulers, but with less hardware, or it can accommodate more intensive workloads with the same hardware provisioning. Next, we

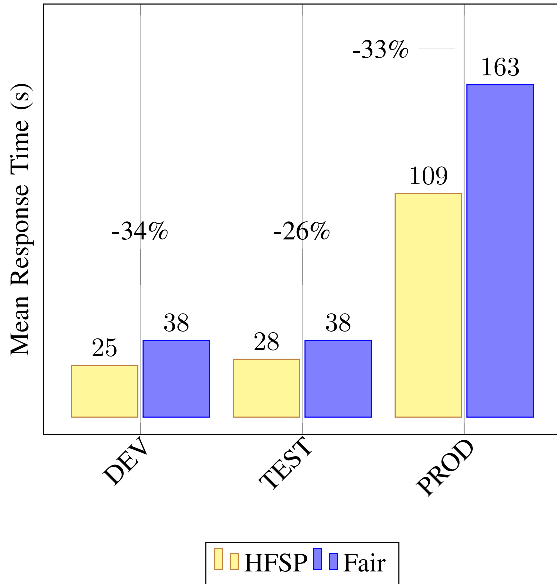


Fig. 6: Aggregate mean response times for all workloads.

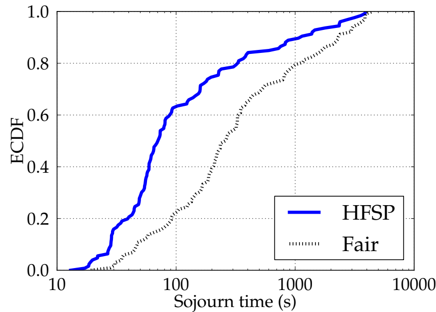


Fig. 7: ECDF of the response times for the DEV workload, excluding jobs from bin 1.

delve into a more detailed analysis of response times, across all workloads presented in the global performance overview.

The DEV workload is mostly composed of jobs from bin 1 that are treated in the same way by both HFSP and Fair schedulers: this biases the interpretation of results using only first order statistics. Therefore, in Figure 7, we show the *empirical cumulative distribution function* (ECDF) of job response times for all bins except bin 1. We notice that jobs that have a response time less or equal to 80 seconds are 60% in HFSP and only 20% in Fair. The reason of this boost in performance is the fact that HFSP runs jobs in sequence while Fair runs them in parallel. By running jobs in series, HFSP focuses all the resources on one job that finishes as soon as possible without penalizing other jobs, leading to increased performance overall.

The TEST workload is the most problematic for HFSP because the jobs are distributed almost uniformly among the four bins. In Figure 8, we decompose our analysis per bin, and show the ECDF of the job response times for all bins except for those in bin 1. For jobs in bin 2, the median response times are of 31 seconds and 45 seconds for HFSP and Fair, respectively. For jobs in bin 3, median values are more distant: 98 seconds and 290 seconds respectively for HFSP and Fair. Finally, for jobs in bin 4, the gap further widens: 1000 seconds versus almost 2000 seconds for HFSP and Fair, respectively. The submission of jobs from bin 3 and 4 slows down the Fair scheduler, while HFSP performs drastically better because it "focuses" cluster resources to individual jobs.

Our results for the PROD workload are substantially in favor of HFSP, that outperforms the Fair scheduler both when considering the *mean* response times (see Figure 6), and the full distribution of response times: in this latter case, *e.g.,* median response times of HFSP are one order of magnitude lower than those in the Fair scheduler.

2) Slowdown: Figure 9 shows the ECDF of the per-job slowdown. Recall that the slow down of a job equals its response time divided by its size: hence, values close or equal to 1 are best. Thus, we use Figure 9 to compare the HFSP and Fair schedulers with respect to the notion of "fairness" we introduced earlier: globally, our results indicate that HFSP is always more fair to jobs than the Fair scheduler.

In particular, we notice that TEST and PROD workloads are particularly difficult for the Fair scheduler. Indeed, a large fraction of jobs is mistreated, in the sense they have to wait long before being served. With the Fair scheduler, job mistreatment worsen when workloads are "heavier"; in contrast, HFSP treats well the vast majority of jobs, and this is true also for demanding workloads such as TEST and PROD. For example, we can use the median slowdown to compare the behavior of the two schedulers: the gap between HFSP and Fair widens from a few units, to one order of magnitude for the PROD workload.

D. Micro Benchmarks

In this Section we study additional details of the HFSP and Fair schedulers, and introduce new results that measure cluster utilization and allow to assess job size estimation errors. Finally, we focus on job and task preemtpion and discuss about the impact on the performance of such a mechanism.

1) Cluster load: We now study the implications of job scheduling from the perspective of cluster utilization: when workloads (like the ones we use in our experiments) are composed by bursts of arrivals, it is important to understand the ability of the system to "absorb" such bursts, without overloading the queue of the pending jobs. We thus define the *cluster load* as the number of jobs currently in the system, either running or waiting to be served.

To understand how cluster load varies with the two schedulers, we focus on a individual run of the PROD workload, because of its high toll in cluster resources. Figure 10 illustrates the time-series of the cluster load for both HFSP and Fair schedulers. There are two significant bursts: in the first, between 0.4 and 0.6 hours, 18 new jobs arrive; in the second, between 1.34 and 1.5 hours, 35 new jobs are submitted. Clearly, Figure 10 shows that HFSP handles bursts of arrivals better than the Fair scheduler: for the latter, the cluster load increases corresponding to each burst, whereas the HFSP scheduler induces a smoother load. Indeed, since HFSP schedules jobs in series, it is able to serve jobs faster

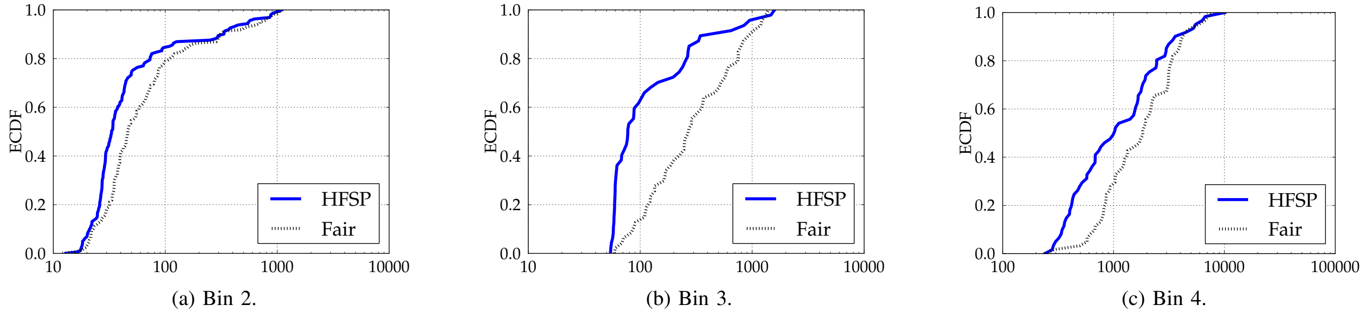


Fig. 8: ECDF of the response times for the TEST workload, grouped per bin.

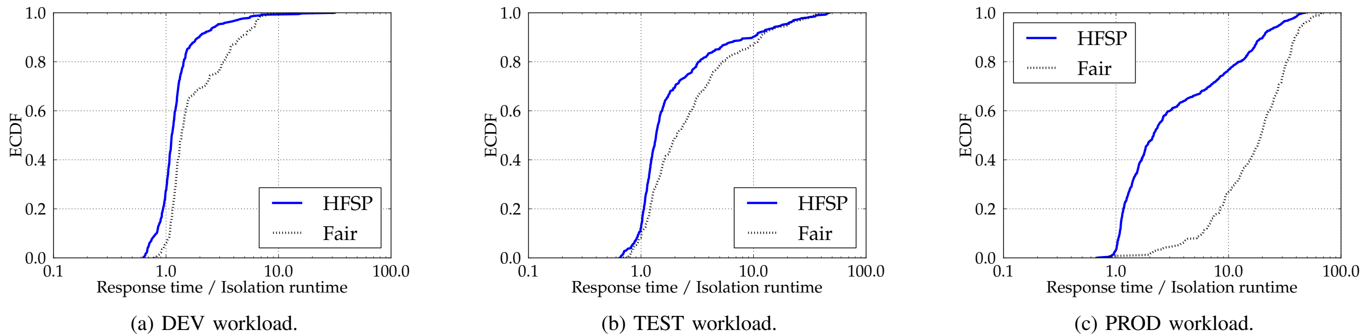


Fig. 9: ECDF of the per-job slowdown, for all workloads.

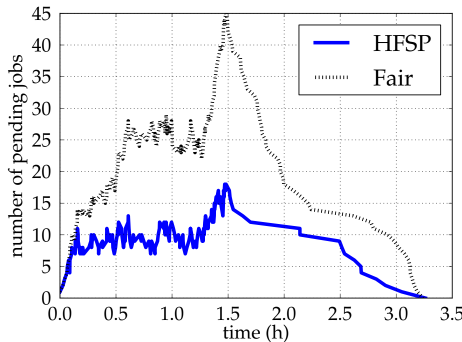


Fig. 10: Time-series of the cluster load, for an individual experiment with the PROD workload.

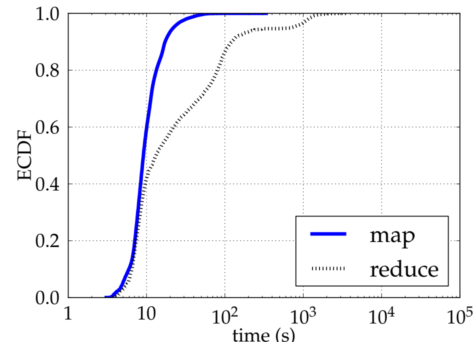


Fig. 11: ECDF of task run times, for the MAP and REDUCE phases of all jobs across all workloads.

– and thus free up resources more quickly – than the Fair scheduler, which instead prolongs jobs service, by granting few resources to all of them.

2) Task time and error: We now focus solely on HFSP, and analyze an apparently delicate component thereof, *i.e.,* the job size estimation module. Our experimental results indicate that HFSP is resilient to job size estimation errors: we show that, by focusing on a detailed analysis of *task times* (which determine the size of a job), and the estimation errors induced by the simple estimator we implemented, it only takes a few training tasks to deduce job sizes.

Figure 11 shows an aggregate distribution of task times for MAP and REDUCE phases for all jobs in all workloads and for all our experiments. It is possible to see that most MAP tasks complete within 60 seconds, and the variability among different tasks is limited. Instead, REDUCE task times variability is extremely high, with differences peaking at 2 orders of magnitude. Given the skew of REDUCE task times, it is reasonable to question the accuracy of the simple estimation mechanism we use in HFSP.

A closer look at task time distribution, however, reveals that within a single job, task times are rather stable. Figure 12 shows the ECDF of the normalized MAP and REDUCE task times: this is obtained by grouping task times belonging to the same job together, computing the mean task time for each job, and normalizing task times by the corresponding mean task time of the job they belong to. For instance, if a job has two tasks with task time equal to 10 and 30 seconds respectively,

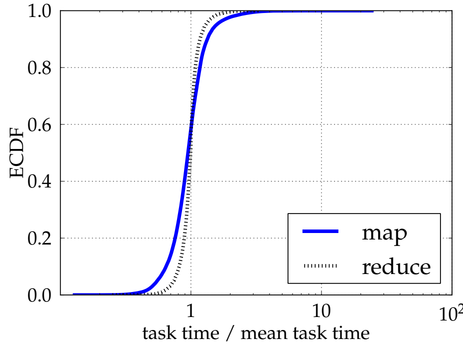


Fig. 12: ECDF of the normalized task run time, for the MAP and REDUCE phases of all jobs across all workloads.

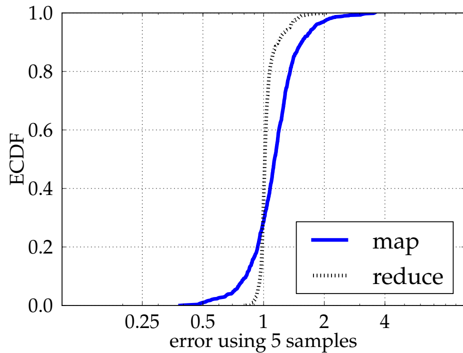


Fig. 13: ECDF of estimation errors, for the MAP and REDUCE phases of all jobs across all workloads.

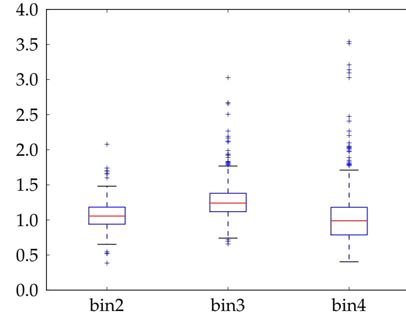


Fig. 14: Boxplots of estimation errors, grouped by bin, for the MAP phase of all jobs across all workloads.

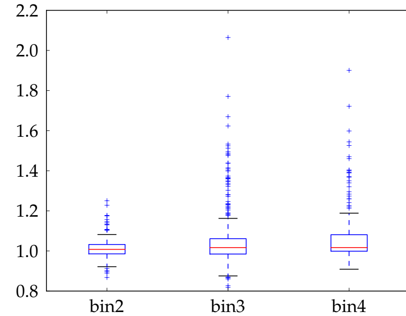


Fig. 15: Boxplots of estimation errors, grouped by bin, for the REDUCE phase of all jobs across all workloads.

then the mean task time for that job would be 20 seconds, and the normalized task times of these tasks would be 0.5 and 1.5 seconds respectively. As such, Figure 12 indicates that it is sufficient to use a small subset of tasks to compute a suitable job size estimate: this allows to distinguish large jobs from small ones, thus avoiding "inversions" in the job schedule.

We support the above claim with Figure 13, that shows the ECDF of the estimation error we measured for MAP and REDUCE phase across all jobs and workloads. For the MAP phase, some jobs (less than 5%) are under-estimated by a factor of 0.4/0.5, while some jobs (less than 4%) are over-estimated by a factor of 2/3.5. For the reduce phase, the number of under-estimated jobs is very small and in general under-estimation is negligible while over-estimation happens only for 10% of jobs by a factor smaller then 2. As a consequence, job size estimates are in the same order of magnitude of real job sizes. This means that two jobs from the same "bin" can be switched but two jobs from two different bins are hardly or never switched (see Section III-D).

In Figures 14 and 15 we decompose estimation errors according to the size of the job (bin for the dataset used), to understand whether errors are more likely to occur for larger or smaller jobs. Results for jobs in bin 1 are omitted because those jobs have less than 5 tasks and they finish before the estimation is done. The boxplots (indicating quartiles in the boxes and outliers outside of whiskers) show that HFSP tends to over-estimate rather than under-estimate job sizes. In our experiments, estimation error is bounded by the factor of 3.5 for the MAP phase and 2 for the REDUCE phase. The majority of the estimated sizes are around 1, showing that often HFSP estimates a size that is very close to the correct one. Since jobs from bins 3 and 4 have more tasks than jobs from bin 2, the estimations for bins 3 and 4 are less precise than the estimations for bin 2 (Table I). In all these cases, the quality of job size estimations are well within the limits that where size-based schedulers can perform efficiently [18].

Given these results, we conclude that the output of our estimator is good enough for HFSP to schedule jobs correctly.

3) Preemption techniques: As discussed in Section III-E, there are two possible approaches to task preemption: the WAIT and the KILL primitives; here we analyze the impact on system performance and fairness of both. To this aim, we focus on the TEST workload, which is the most problematic workload for HFSP (see Section IV-C1). Figures 16 and 17 show the ECDF of response times and slowdown.

We notice that killing tasks improves HFSP fairness: indeed, when a waiting job is granted higher priority than a running job, the scheduler kills immediately the running tasks of the latter, and freeing up resources almost immediately. Without the KILL primitive, the scheduler can only grant resources to smaller, higher priority jobs when running tasks are complete.

The situation changes when considering response times. Indeed, killing tasks wastes work, and this is particularly true for tasks that are close to completion. Figure 16 shows that the KILL primitive is especially problematic for large jobs: as

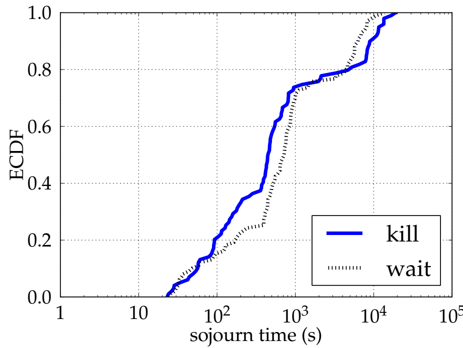


Fig. 16: ECDF of job response times, for the TEST workload. Comparison between preemption primitives.

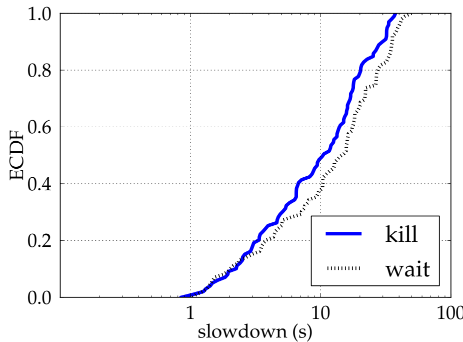


Fig. 17: ECDF of the per-job slowdown, for the TEST workload. Comparison between preemption primitives.

they last longer, they have more chances to have their tasks killed.

In summary, the lesson we learned is that the choice of the primitive to use to perform job preemption depends on the "functional" objectives of an administrator. If fairness is considered important, then the KILL primitive is more appropriate. Instead, if system responsiveness is the objective, then the WAIT primitive is a sensible choice. As a future research direction, we plan to integrate HFSP with new preemption primitives [25], [26] that aim at striking a good balance between fairness and responsiveness.

V. RELATED WORK

MapReduce in general and Hadoop in particular have received considerable attention recently, both from the industry and from academia. Since we focus on job scheduling, we consider here the literature pertaining to this domain.

Theoretical Approaches: Several theoretical works tackle scheduling in multi-server systems – a recent example is the work by Moseley and Fox [34]. These works, which are elegant and important contributions to the domain, provide performance bounds and optimality results based on simplifying assumptions on the execution system (*e.g.,*, jobs with a single phase). Some works provide interesting approximability results applied to simplified models of MapReduce [35], [36]. In contrast, we focus on the design and implementation of a scheduling mechanism taking into account all the details and intricacies of a real system.

Fairness and QoS: Several works take a system-based approach to scheduling on MapReduce. For instance, the Fair scheduler and its enhancement with a delay scheduler [11] is a prominent example to which we compare our results. Several other works [37]–[40] focus on resource allocation and strive at achieving fairness across jobs, but do not aim at optimizing response times. Sandholm and Lai [41] 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 [42] address the problem of scheduling jobs to meet user-provided deadlines, but assume job runtime to be an input to the scheduler.

Flex [43] is a proprietary Hadoop size-based scheduler. In Flex, "fairness" is defined as avoiding job starvation and guaranteed by allocating a part of the cluster according to Hadoop's Fair scheduler; size-based scheduling (without aging) is then performed only on the remaining set of nodes. In contrast, by using aging our approach can guarantee fairness while allocating all cluster resources to the highest priority job, thus completing it as soon as possible.

Job Size Estimation: Various recent approaches [44]–[47] propose techniques to estimate query sizes in recurring jobs. Agarwal *et al.* [46] report that recurring jobs are around 40% of all those running in Bing's production servers. Our estimation module, on the other hand, works on-line with *any* job submitted to a Hadoop cluster, but it has been designed so that the estimator module can be easily plugged with other mechanisms, benefitting from advanced and tailored solutions.

Complementary approaches: Task size skew is a problem in general for MapReduce applications, since larger tasks delay the completion of a whole job; skew also makes job size estimation more difficult. The approach of SkewTune [48] greatly mitigates 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. Tian *et al.* [49] propose a mechanism where IO-bound and CPU-bound jobs run concurrently, benefitting from the absence of conflicts on resources between them. We remark that also in this case it is possible to benefit from size-based scheduling, as it can be applied separately on the IO- and CPU-bound queues. Tan *et al.* [50], [51] propose strategies to adaptively start the REDUCE phase in order to avoid starving jobs; also this technique is orthogonal to the rest of scheduling choices and can be integrated in our approach. Hadoop offers a Capacity Scheduler [52], which is designed to be operated in multi-tenant clusters where different organizations submit jobs to the same clusters in separate queues, obtaining a guaranteed amount of resources. We remark that also this idea is complementary to our proposal, since jobs in each queue could be scheduled according to a size-based policy such as HFSP, and reap according benefits.

Framework Schedulers: Recent works have pushed the idea of sharing cluster resources at the framework level, for example to enable MapReduce and Spark [53] "applications" to run concurrently. Monolithic schedulers such as YARN [54] and Omega [55] use a single component to allocate resources to each framework, while two-level schedulers [56], [57] have a single manager that negotiates resources with independent,

framework-specific schedulers. We believe that such framework schedulers impose no conceptual barriers for size-based scheduling, but the implementation would require very careful engineering. In particular, size-based scheduling should only be limited to batch applications rather than streaming or interactive ones that require continuous progress.

VI. Conclusion

Resource allocation plays an increasingly important role in current Hadoop clusters, as modern data analytics and workloads are becoming more complex and heterogeneous. Our work was motivated by the increasing demand for system responsiveness, driven by both interactive data analysis tasks and long-running batch processing jobs, as well as for a fair and efficient allocation of system resources.

Alas, system responsiveness and fairness requirements have been traditionally at odds: a scheduling discipline that would satisfy one, had to sacrifice the other. For example, in our work we argued that the default scheduling mechanism used in typical Hadoop deployments, the Fair scheduler, achieves fairness but trades on system response times. Only a tedious, manual process involving an expert administrator could mitigate the shortfalls of a processor sharing-like discipline, albeit for a rather static workload composition.

In this paper we presented a novel approach to the resource allocation problem, based on the idea of size-based scheduling. Our effort materialized in a full-fledged scheduler that we called HFSP, the Hadoop Fair Sojourn Protocol, which implements a size-based discipline that satisfies simultaneously system responsiveness and fairness requirements.

Our work raised many challenges: evaluating job sizes online without wasting resources, avoiding job starvation for both small and large jobs, and guaranteeing short response times despite estimation errors were the most noteworthy. HFSP uses a simple and practical design: size estimation trades accuracy for speed, and starvation is largely alleviated, by introducing the mechanisms of virtual time and aging.

A large part of this article was dedicated to a thorough experimental campaign to evaluate the benefits of HFSP when compared to the default Fair scheduler in Hadoop. We defined several realistic workloads that are representative of typical uses of an Hadoop cluster, and proceeded with a comparative analysis using our deployment, configured according to current best practices. Our experiments, that amount to more than 1500 real jobs, indicated that HFSP systematically – and in some cases, by orders of magnitude – outperformed the Fair scheduler, both with respect to system response times and fairness properties.

Furthermore, we insisted on studying the sensitivity of HFSP to a seemingly delicate component, namely the one responsible for estimating job sizes, which are undoubtedly the most important ingredients of a size-based scheduler. Our experiments reveal that rough estimates are sufficient for HFSP to operate correctly, which validate our design choice of favoring speed, rather than accuracy.

Overall, this work showed that size-based scheduling disciplines, that have been for long relegated to a rather marginal role, are indeed very practical alternatives to current default schedulers. In particular, for Hadoop, we showed that our implementation of HFSP drastically improves on system response times, guarantees fairness and, crucially, only requires a handful of parameters to set. Furthermore, HFSP adapts easily to workload dynamics and it is tolerant to failures.

HFSP has been designed to optimize performance without stringent requirements in terms of parameter tuning. However, in large clusters, more elaborate configurations are often needed, *e.g.,* to manage user groups and job priorites. Such cases only require some additional engineering effort to be supported. For example, both the Fair and the Capacity scheduler [52] can be used as hierarchical schedulers: they allow instantiating user "pools" and define a per-pool scheduling policy. Additionally, to offer job-level priority, it is possible to alter the HFSP aging module as mentioned in Section III-B to emulate more complex policies *in the virtual time*. These ideas are within our plans for future developments.

Currently, we are extending HFSP such that it can use recent job preemption primitives, a necessary condition to allow even faster response times; moreover, we will consolidate our codebase and contribute it to the Hadoop community, casting HFSP to work for modern frameworks such as YARN and Mesos.

Acknowledgment

This work has been partially supported by the EU projects BigFoot (FP7-ICT-317858) and mPlane (FP7-ICT-318627).

References

[1] Apache, "Hadoop: Open source implementation of MapReduce," http://hadoop.apache.org/.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. of USENIX OSDI*, 2004.
[3] Apache, "Spark," http://spark.apache.org/.
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.
[5] Microsoft, "The naiad system," https://github.com/MicrosoftResearchSVC/naiad.
[6] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.
[7] 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*, 2012.
[8] K. Ren *et al.*, "Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads," in *Proc. of VLDB*, 2013.
[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *NSDI*, vol. 13, 2013.
[10] Apache, "Oozie Workflow Scheduler," http://oozie.apache.org/.
[11] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of ACM EuroSys*, 2010.
[12] Apache, "The hadoop fair scheduler," http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.
[13] E. Friedman and S. Henderson, "Fairness and efficiency in web server protocols," in *Proc. of ACM SIGMETRICS*, 2003.
[14] L. E. Schrage and L. W. Miller, "The queue m/g/1 with the shortest remaining processing time discipline," *Operations Research*, vol. 14, no. 4, 1966.
[15] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1998, pp. 270–279.

[16] A. S. Tanenbaum and A. Tannenbaum, *Modern operating systems*. Prentice hall Englewood Cliffs, 1992, vol. 2.
[17] Stallings, *Operating Systems*. Prentice hall, 1995.
[18] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi, "Revisiting size-based scheduling with estimated job sizes," in *IEEE MASCOTS*, 2014.
[19] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. of IEEE MASCOTS*, 2011.
[20] J. Nagle, "On packet switches with infinite storage," *Communications, IEEE Transactions on*, vol. 35, no. 4, 1987.
[21] S. Gorinsky and C. Jechlitschek, "Fair efficiency, or low average delay without starvation," in *Proc. of IEEE ICCCN*, 2007.
[22] M. Dell'Amico, D. Carra, and P. Michiardi, "Psbs: Practical size-based scheduling," *arXiv preprint arXiv:1410.6122*, 2014.
[23] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM TON*, vol. 6, no. 5, 1998.
[24] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous MapReduce workloads," in *Proc. of CNSM*, 2011.
[25] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 6.
[26] M. Pastorelli, M. Dell'Amico, and P. Michiardi, "Os-assisted task preemption for hadoop," in *Proc. of IEEE ICDCS workshop*, 2014.
[27] Apache, "Hadoop fair scheduler," http://hadoop.apache.org/docs/stable/fair_scheduler.html.
[28] ——, "Hadoop MapReduce JIRA 1184," https://issues.apache.org/jira/browse/MAPREDUCE-1184.
[29] ——, "PigMix," https://cwiki.apache.org/PIG/pigmix.html.
[30] TPC, "Tpc benchmarks," http://www.tpc.org/information/benchmarks.asp.
[31] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for hadoop: Time to rethink?" in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 20.
[32] M. Pastorelli *et al.*, "Practical size-based scheduling for MapReduce workloads," *CoRR*, vol. abs/1302.2749, 2013.
[33] A. Wierman, "Fairness and scheduling in single server queues," *Surveys in Operations Research and Management Science*, vol. 16, no. 1, 2011.
[34] K. Fox and B. Moseley, "Online scheduling on identical machines using SRPT," in *In Proc. of ACM-SIAM SODA*, 2011.
[35] H. Chang *et al.*, "Scheduling in MapReduce-like systems for fast completion time," in *Proc. of IEEE INFOCOM*, 2011.
[36] B. Moseley *et al.*, "On scheduling in map-reduce and flow-shops," in *In Proc. of ACM SPAA*, 2011.
[37] T. Sandholm and K. Lai, "MapReduce optimization using regulated dynamic prioritization," in *Proc. of ACM SIGMETRICS*, 2009.
[38] M. Isard *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *Proc. of ACM SOSP*, 2009.
[39] A. Ghodsi *et al.*, "Dominant resource fairness: Fair allocation of multiple resources types," in *Proc. of USENIX NSDI*, 2011.
[40] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.
[41] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in Hadoop," in *Proc. of JSSPP*, 2010.
[42] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *Proc. of CloudCom*, 2010.
[43] J. Wolf *et al.*, "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," in *Proc. of ACM MIDDLEWARE*, 2010.
[44] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for MapReduce environments," in *Proc. of ICAC*, 2011.
[45] ——, "Two sides of a coin: Optimizing the schedule of MapReduce jobs to minimize their makespan and improve cluster performance," in *Proc. of IEEE MASCOTS*, 2012.
[46] S. Agarwal *et al.*, "Re-optimizing Data-Parallel Computing," in *Proc. of USENIX NSDI*, 2012.
[47] A. D. Popescu *et al.*, "Same queries, different data: Can we predict query performance?" in *Proc. of SMDB*, 2012.
[48] Y. Kwon *et al.*, "Skewtune: mitigating skew in MapReduce applications," in *Proc. of ACM SIGMOD*, 2012.
[49] C. Tian *et al.*, "A dynamic MapReduce scheduler for heterogeneous workloads," in *Proc. of IEEE GCC*, 2009.
[50] J. Tan, X. Meng, and L. Zhang, "Delay tails in MapReduce scheduling," in *Proc. of ACM SIGMETRICS*, 2012.
[51] ——, "Performance analysis of coupling scheduler for MapReduce/Hadoop," in *Proc. of IEEE INFOCOM*, 2012.
[52] Apache, "Hadoop capacity scheduler," http://hadoop.apache.org/docs/stable/capacity_scheduler.html.
[53] M. Zaharia *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.
[54] Apache, "Hadoop nextgen MapReduce (yarn)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[55] M. Schwarzkopf *et al.*, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of EuroSys*, 2013.
[56] B. Hindman *et al.*, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.
[57] Apache, "Hadoop on demand," http://hadoop.apache.org/docs/stable/hod_scheduler.html.

Mario Pastorelli received his M.S. in Computer Science from the University of Genoa (Italy) and his Ph.D. in in Computer Science from Telecom ParisTech with a thesis on Job Scheduling for Data-Intensive Scalable Computing Systems. He is currently working at Teralytics AG as Software Engineer and Architect in the data acquisition and elaboration team.

Damiano Carra received his Laurea in Telecommunication Engineering from Politecnico di Milano, and his Ph.D. in Computer Science from University of Trento. He is currently an Assistant Professor in the Computer Science Department at University of Verona. His research interests include modeling and performance evaluation of peer-to-peer networks and distributed systems.

Matteo Dell'Amico is a researcher at Symantec Research Labs; his research revolves on the topic of distributed computing. He received his M.S. (2004) and Ph.D. (2008) in Computer Science from the University of Genoa (Italy); during his Ph.D. he also worked at University College London. Between 2008 and 2014 he was a researcher at EURECOM. His research interests include data-intensive scalable computing, peer-to-peer systems, recommender systems, social networks, and computer security.

Pietro Michiardi received his M.S. in Computer Science from EURECOM and his M.S. in Electrical Engineering from Politecnico di Torino. Pietro received his Ph.D. in Computer Science from Telecom ParisTech (former ENST, Paris). Today, Pietro is an Assistant Professor of Computer Science at EURECOM. Pietro currently leads the Distributed System Group, which blends theory and system research focusing on large-scale distributed systems (including data processing and data storage), and scalable algorithm design to mine massive amounts of data. Additional research interests are on system, algorithmic, and performance evaluation aspects of computer networks and distributed systems.