HFSP: Size-based Scheduling for Hadoop

Mario Pastorelli*, Antonio Barbuzzi*, Damiano Carra†, Matteo Dell'Amico* and Pietro Michiardi*
* EURECOM – Campus SophiaTech, France
Email: pastorel@eurecom.fr,barbuzzi@eurecom.fr,della@linux.it,michiard@eurecom.fr
† University of Verona, Italy
Email: damiano.carra@univr.it

***Abstract*—Size-based scheduling with aging has, for long, 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, and show that HFSP is largely tolerant to job size estimation errors.

I. Introduction

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [1], has created the need to manage the resources of compute clusters that operate in a shared, multi-tenant environment. 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 heterogeneity is substantiated by recent studies [2], [3] 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. Data exploration, preliminary analyses and algorithm tuning on small datasets 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 [4] contribute to workload heterogeneity by generating a number of small "orchestration" jobs. Interactive and orchestration jobs should not wait too long before being served, even if larger production jobs are in execution. Commonly, the task of a cluster administrator involves the *manual setup* of a number of "pools" to dedicate resources to different job categories, and the fine-tuning of the parameters governing resource allocation. This process is tedious, error prone, and cannot adapt easily to changes in the workload composition. In addition, it is often the case for clusters to be over-dimensioned [2]: this simplifies resource allocation (with abundance, managing resources is less critical), but has the downside of costly deployments that are left unused for long.

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 [5], the most widely adopted open-source implementation of MapReduce. We proceed with 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 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 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 theoretic work [6] 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 II) 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 experiment campaign, where we compare the HFSP scheduler to a prominent scheduler used in production-level Hadoop deployments: the

[1] https://bitbucket.org/bigfootproject/hfsp

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 both to interactivity and efficiency requirements.

II. Size-Based Scheduling for Data-intensive Systems

Next, we outline the principles of size-based scheduling, in view of our goal to bring its benefits to a system like Hadoop MapReduce. In addition, we discuss on the feasibility of such an approach to job scheduling, when job sizes are not known a priori, but can only be evaluated approximately.

A. 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, the FIFO and FAIR schedulers in Hadoop are inspired by these two approaches. In FCFS, jobs are scheduled in the order of their submission, while in PS resources are divided evenly so that each active jobs keeps progressing. In loaded systems, these disciplines have severe shortcomings: in FCFS, large running jobs can delay very significantly small ones that are waiting to be executed; in PS, each additional job delays the completion of *all* the others.

Essentially, 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 *sojourn time* (or *response time*), that is the time that passes between a job submission and its completion [7]. Policies like SRPT may however incur in *starvation*: if smaller jobs are continuously submitted, larger ones may never get scheduled. In order to avoid starvation, a common solution is to perform *job aging*: virtually decreasing the size of jobs waiting in the queue, in order to make sure that they will be eventually scheduled.

Figure 1 compares PS with the SRPT scheduling discipline with an illustrative example: 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 (slowly) in parallel, in a size-based discipline j_1 is *preempted*: the result is that j_2 and j_3 complete earlier. It is worth noting that, in this case, the completion time of j_1 does not suffer from preemption: somewhat counter to intuition, this is often the case for SRPT-based scheduling [8].

B. Impact of Size Estimation Errors

In MapReduce, job size distribution is very skewed, ranging from few seconds to several hours [2], [3]. These sizes are difficult to obtain a priori, even though various recent works tackle the task of estimating MapReduce job sizes [9]–[13] (we discuss them in more detail in Section V); in addition, Lu *et al.* evaluate the impact of estimation errors on size-based scheduling for synthetic traces [14]. Unfortunately, the combination of these works is not sufficient to understand

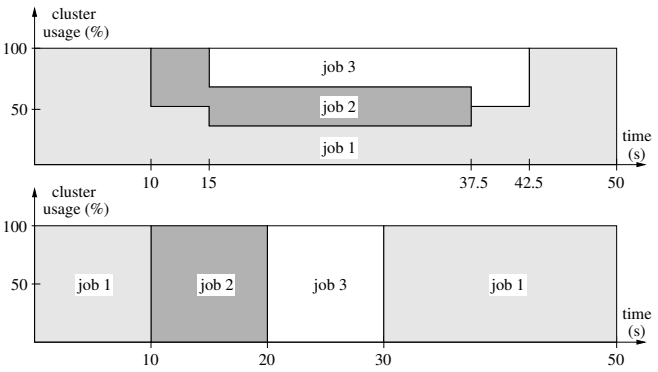


Fig. 1. Comparison between PS (top) and SRPT (bottom).

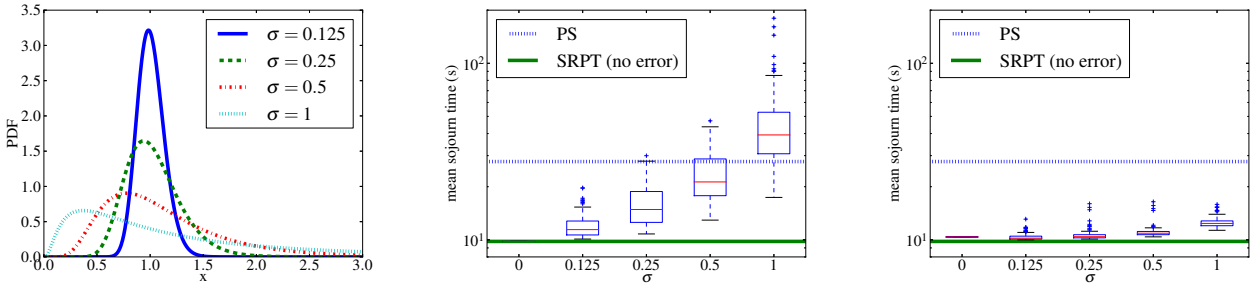
which level of estimation errors would be acceptable for size-based scheduling in our context of extremely diverse job sizes.

To evaluate, at an abstract level, the interplay between estimation errors and performance, we first use a simulation-based approach. The simulator, which is available as free software,[2] abstracts from the details of a full-fledged MapReduce system: it assumes that jobs can utilize the full cluster capacity, and that they can be preempted istantaneously. Thus, a job is only characterized by its submission time and the amount of time it would need to complete if utilizing the full cluster capacity; we will lift these assumptions by evaluating our real system implementation in Section IV. In our simulation, we replay a 24-hour trace made available with the SWIM tool [15], which comprises 24,442 production jobs at Facebook in 2010, and has been used to validate other works [16], [17].

We consider estimation errors to be log-normally distributed: in our simulator, a job having size of s will be estimated as $\hat{s} = sX$, where X is a random variable with distribution Log-$\mathcal{N}(0, \sigma^2)$: the choice of the log-normal distribution reflects the intuition that an under-estimation $\hat{s} = s/t$ ($t > 1$) is as likely as an over-estimation $\hat{s} = ts$; in Section IV-C, we show experimentally that our estimation errors are indeed well approximated by such distribution. We report results for the default simulator settings, which reflect a heavily-loaded system where aggregated disk bandwidth is larger than network bandwidth. Details on the simulator and its parameter space are available in a technical report [18].

Figure 2(b) shows box-plots (that is, the most important percentiles of a cumulative distribution function) for the mean sojourn times achieved by SRPT with 100 simulation runs for each value of σ, comparing it with mean sojourn times using PS and SRPT without estimation errors. Notice that, because of the large variance in sojourn times, we plot them on a logarithmic scale. Since the trace is fixed, what changes between simulation runs are only estimation errors; for reference, log-normal distributions with different values of σ are shown in Figure 2(a). Clearly, in situations of high load, a FCFS policy performs poorly (results are outside of the plotted graphs, with a mean sojourn time of 1934 s). Instead,

[2]https://bitbucket.org/bigfootproject/schedsim



(a) Log-normal distributions Log-$\mathcal{N}(0, \sigma^2)$. (b) SRPT. (c) SRVT.

Fig. 2. Impact of job size estimation errors on the performance achieved by size-based scheduling disciplines. Given a reference error distribution: *i)* size-based scheduling is robust to approximate job sizes; and *ii)* job aging further mitigates the impact of such errors.

the performance of SRPT is strongly dependent on the error distribution. When $\sigma = 0.25$, where more than half of the estimations are wrong by a factor of 20% or more, SRPT largely outperforms PS. For higher values of σ, however, estimation errors impact scheduling choices, and the sojourn time dramatically increases.

We find that aging policies effectively counter these shortcomings: even a bad scheduling choice – caused, for example, by overestimating the size of a job – is, in the long run, "corrected" by decreasing job size through aging. We verify our claim by implementing a common aging policy, where the remaining processing time is decreased by the amount of work performed in a *virtual* PS scheduler [6], [19], [20]. This technique, which we label Shortest Remaining Virtual Time (SRVT), results (in the absence of estimation errors) in scheduling jobs in series, following the order in which they would complete with the virtual PS. As shown in Figure 2(c), the mean sojourn time of SRVT is slightly worse than that of SRPT in the absence of errors. However, when errors are present, aging is vastly preferable.

We can conclude that size-based scheduling appears largely beneficial to systems like Hadoop MapReduce, even in the presence of large estimation errors; we attribute this to the fact that job sizes vary by several orders of magnitude in characteristic workloads, and that this eases the task of a scheduler that only has to distinguish between jobs having very different sizes. In addition, job aging mitigates estimation errors. These findings inspire us in the design of our system, which is described next.

III. System Implementation

Implementing a size-based scheduling protocol in Hadoop raises a number of challenges. A few of them come from the fact that MapReduce jobs are scheduled at the lower granularity of *tasks*, and that they consist of two separate phases – Map and Reduce – which are scheduled independently (Sec. III-A). In addition, job size is in general unknown a priori: to evaluate it, we develop an *estimation* module (Sec III-D) that provides, at first, a coarse estimation of job size upon submission, and then refines it after the first few *sample* tasks have been run. Estimations are used by an *aging* module (Sec. III-C) which outputs job priorities; finally the *scheduler* (Sec. III-B) uses such priorities to allocate resources while ensuring that sample tasks are allocated quickly, to converge rapidly to a more accurate job size estimation. Next, we introduce existing Hadoop schedulers; we then describe the components of our system.

A. Hadoop MapReduce

MapReduce, popularized by Google [1] and by Hadoop [5], is both a programming model and an execution framework. In MapReduce, a job consists of three phases and accepts as input a dataset, appropriately partitioned and stored in a distributed file system. In the first phase, called Map, a user-defined function is applied in parallel to input partitions to produce intermediate data stored on the local file system of each machine of the cluster; intermediate data is sorted and partitioned when written to disk. Then, a Reduce phase begins. It comprises a Shuffle sub-phase, where intermediate data is pulled by the *reducers*: data from multiple mappers is sorted and aggregated to produce output data.

Hadoop Scheduling: In Hadoop, the JobTracker coordinates the worker machines, called TaskTrackers. The scheduler resides in the JobTracker and allocates TaskTracker resources to running tasks: Map and Reduce tasks are granted independent *slots* on each machine.[3] The scheduler is called whenever one or more task slots become free, and it decides which tasks to allocate on those slots.

When a *single* job is submitted to the cluster, the scheduler assigns a number of Map tasks 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

[3]HFSP is currently implemented for Hadoop version 1, which is currently the most widely used in production settings [21]; Hadoop version 2 ("YARN") uses a different architecture. As we further detail in Section V, we believe that porting HFSP to YARN would mostly be an implementation effort rather than a research accomplishment.

multiple "waves", especially when processing very large data. Similarly, REDUCE tasks are scheduled once intermediate data, output from mappers, is available.[4] When *multiple* jobs are submitted to the cluster, the scheduler allocates available task slots across jobs.

In this work we consider the Hadoop Fair Scheduler, which we call FAIR. FAIR groups jobs into "pools" (generally corresponding to users or groups of users) and assigns each pool a guaranteed minimum share of cluster resources, which are split up among the jobs in each pool. In case of excess capacity (because the cluster is over dimensioned with respect to its workload, or because the workload is lightweight), FAIR splits it evenly between jobs. When a slot becomes free and needs to be assigned a task, FAIR proceeds as follows: if there is any job below its minimum share, it schedules a task of that particular job. Otherwise, FAIR schedules a task that belongs to the job that has received less resources.

B. The Job Scheduler

In our architecture, the scheduler operates on a set of job priorities that are output by the *aging* module (Sec. III-C), which uses job size information provided by the *estimation* module (Sec. III-D). Next, we highlight the main issues that we encountered while implementing our scheduler, and we motivate our design choices.

Job Preemption: Unlike the abstract protocols shown in Section II, which schedule full jobs, here scheduling is performed at the *task* granularity. From an abstract point of view, when the priority of a running job is lower than the one of a waiting task, the running job should be *preempted* to free resources for the other. In Hadoop, this can be implemented either by *killing* the running tasks of the preempted job, or by simply *waiting* for those tasks to complete. Note that scheduling choices are more critical in situations of high load, and that the choice of killing running tasks may result in increasing load even more, because the work done by killed tasks should be performed again. As such, in this work, we opt for wait-based preemption.

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

Priority to Training: Initially, the estimation module provides a rough estimate for the size of new jobs. This estimate is then updated after the first s sample tasks of a job are executed. To guarantee that job size estimates quickly converge to more accurate values, the scheduler gives priority to sample tasks across jobs – up to a threshold of $t\%$ of the total number of slots. Such threshold avoids starvation of "regular" jobs in case of a bursty job arrival pattern.

Data locality: For performance reasons, it is important to make sure that MAP tasks work on local data. For this reason, we use the *delay scheduling* strategy [16], 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 Policy: As a result of all the choices described above, our scheduling policy – which is called whenever a task slot frees up – behaves as follows:

1) Select eligible jobs: those with tasks waiting to be scheduled that conform to the delay scheduling constraints;
2) Sort them according to the priorities obtained from the aging module;
3) Check if sample tasks are running on less than $t\%$ of the slots, and if one or more eligible jobs need to execute sample tasks:
   a) If so, schedule a *sample* task from the highest priority of such jobs;
   b) Otherwise, schedule a task from the highest priority eligible job.

C. Aging Module

The aging module takes as input job size estimates produced by the estimation module, and outputs a priority for each active job, which is used by the scheduling module described above.

To do that, we adopt the notion of *virtual time*, a technique used in many practical implementations of well-known schedulers [6], [19], [20]. Essentially, we keep track of the amount of remaining work for each job in a virtual "fair" system, and update it every time the scheduler is called; job priorities are then output sorted by amount of remaining work. While the remaining work does not necessarily reflect accurately the completion time for queued jobs, the *order* in which those jobs complete in virtual time is all that matters for size-based scheduling to work. As shown at the abstract level in Section II and experimentally in Section IV, this technique is robust against inaccurate job size.

Job aging avoids starvation, achieves fairness, and it requires minimal computational load, since the virtual time does not incur in costly updates for jobs already in queue [6], [19].

Max-Min Fairness: The estimation module outputs job sizes in a "serialized form", that is the sum of runtimes of each task. As such, the physical configuration of the cluster does not influence estimated size. In the virtual time, instead, this becomes a factor: for example, a job requiring only a few tasks cannot occupy the whole virtual cluster, which has the same number of compute slots of the real one. We simulate a *Max-Min Fairness* criterion to take into account jobs that request *less* compute slots than their fair share (*i.e.*, $1/n$-th of the slots if there are n active jobs): a round-robin mechanism allocates virtual cluster slots, starting from small jobs (in terms of the number of tasks). As such, small jobs are implicitly

[4] More precisely, a "slowstart" setting indicates the fraction of mappers that are required to finish before reducers are awarded execution slots.

given priority, which reinforces the idea of scheduling small jobs as soon as possible.

Job Aging: Each job arrival or task completion triggers a call to the job aging function, which decreases the remaining amount of work for each job according to the virtual allocation described above and to the time that has passed from the last invocation of the aging function. The priorities output by the module correspond to the remaining amount of work per job, so that jobs with the least remaining work in the virtual time will be scheduled first.

Failures: The aging module is robust with respect to failures, and supports cluster size upgrades: the max-min fairness allocation uses the information about the number of slots in the system which is provided by the Hadoop framework; once Hadoop detects a failure, job aging will be slower. Conversely, adding nodes will result in faster job aging.

Job Priority and QoS: Our scheduler does not currently implement a concept of job priority; however, the aging function can be easily modified to simulate a Generalized Processor Sharing discipline, leading to a scheduling policy analogous to Weighted Fair Queuing [22].

D. Job Size Estimation

Size-based scheduling requires knowledge of job size. In Hadoop, such information is unavailable until a job completes; however, a first rough estimate of job size can use job characteristics known a priori such as the number of tasks; after the first sample tasks have executed, the estimation can be updated based on their running time.

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

1) Initial Estimation: In Hadoop, the number of MAP and REDUCE tasks each job needs is known a priori. In turn, each MAP task processes an *input split*: data essentially residing on a single, fixed-size, HDFS block. Our first job size approximation is therefore directly proportional to the number of tasks per job.

The size of a MAP (resp. REDUCE) job with k tasks is, at first, estimated as $\xi \cdot k \cdot l$, where l is the average size of past MAP (resp. REDUCE) tasks, and $\xi \in [1, \infty)$ is a tunable parameter that represents the propensity the system has to schedule jobs of unknown size. At the extreme $\xi = 1$, new jobs are scheduled quite aggressively based on the initial estimate, with the possible drawback of scheduling particularly large jobs too early. More conservative choices of $\xi > 1$ avoid this problem, but might result in increased response times by scheduling jobs later. We note that particularly small jobs, with s or less tasks, are scheduled immediately and finish in the training phase.

2) MAP *Phase Size:* It has been observed, across a variety of jobs, that MAP task execution times are generally stable and short [16], [17]. It is thus reasonable to perform job size estimation using only s sample tasks, albeit runtime skew may induce inaccurate size estimation. We recall here that the aging module described above does not require perfect accuracy.

Our estimation uses a measure of the execution time $\sigma_{i,j}$ for each sample task j of job i. For each job, we obtain an estimate of the MAP phase size by multiplying the average (sample) task runtime by k, which is the number of MAP tasks for the estimated job.

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

3) REDUCE *Phase Size:* The REDUCE phase can be broken down in two parts: SHUFFLE time – needed to move and merge data from mappers to reducers – and the execution time of the REDUCE function, which can only start when the SHUFFLE phase has completed.

Size of SHUFFLE*:* As soon as a REDUCE task is scheduled, it starts pulling data from the mappers; once data from all mappers is available, a *global sort* is performed by merging all the mappers' output. Since each mapper output is already locally sorted, a linear-time merge step is sufficient.

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

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

Once we obtain the size (or an estimation of it) for each sample task, we compute the total execution time using the same procedure described in Section III-D2. The final estimate of the whole REDUCE phase is obtained by adding the estimated SHUFFLE time to this estimated execution time.

IV. EXPERIMENTS

This Section focuses on a comparative analysis between the FAIR and HFSP schedulers. After evaluating the global performance of the two schedulers, we focus on the estimation error as output by our size estimation module.

A. Experimental Setup

We used a cluster composed by 36 TASKTRACKER machines with 4 CPUs and 8 GB of RAM each. We configured Hadoop according to advised best practises [23], [24]: the HDFS block size is 128 MB, with replication factor of 3; each TASKTRACKER has 2 map slots with 1 GB of RAM dedicated to each and 1 reduce slots with 2 GB of RAM. In total, our cluster has 72 MAP slots and 36 REDUCE slots. The slowstart factor is configured to start the REDUCE phase for a job when 95% of its MAP tasks are completed.

HFSP operates with the following parameters: the sample set size s for both MAP and REDUCE tasks is set to 5; the Δ timeout to estimate REDUCE task size is set to 10 seconds; we schedule aggressively jobs that are in the training phase, setting $\xi = 1$ and $t = 100\%$. The FAIR scheduler has been configured with a single job pool.

Workloads: We generate workloads using PigMix [25], a benchmarking suite used to test the performance of Apache Pig releases. PigMix is appealing to us because, much like its standard counterparts for traditional DB systems such as TPC [26], it generates realistic datasets with properties such as data skew, and defines queries inspired by real-world data analysis tasks.

We generated four datasets of sizes respectively 1 GB, 10 GB, 40 GB and 100 GB. Job arrival follows a Poisson process, and jobs are generated by choosing uniformly at random a query between the 17 defined in PigMix, and applying it to one of the datasets according to a workload-defined probability distribution. We evaluate two workloads:

- **SMALL:** this workload is inspired by the Facebook 2009 trace observed by Chen *et al.* [17], where a majority of jobs are very small. The mean interval between job arrivals is $\mu = 30\ s$.
- **LARGE:** this workload is predominantly composed of relatively heavy-duty jobs. In this case, the mean interval between jobs is $\mu = 120\ s$.

In Table I, we report the probability distribution for choosing a particular dataset size; we remark that PigMix queries operate on different subsets of the generated datasets, resulting in a variable number of MAP/REDUCE tasks. 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.

We have additional results – not included here for lack of space – that confirm our results on different platforms (Amazon EC2 and the Hadoop Mumak emulator), and with different workloads (synthetic traces generated by SWIM [17]). They are available in a technical report [27].

TABLE I
JOB SIZES IN OUR EXPERIMENTAL WORKLOADS.

Dataset size	Map tasks	Workload	
		SMALL	LARGE
1 GB	< 5	65%	0%
10 GB	$10 - 50$	20%	10%
40 GB	$50 - 150$	10%	60%
100 GB	> 150	5%	30%

TABLE II
MEAN SOJOURN TIME (MST) AND MEAN LOAD.

Workload	MST (s)		Mean Load	
	FAIR	HFSP	FAIR	HFSP
SMALL	63	53	2.26	1.99
LARGE	2,291	544	16.80	4.60

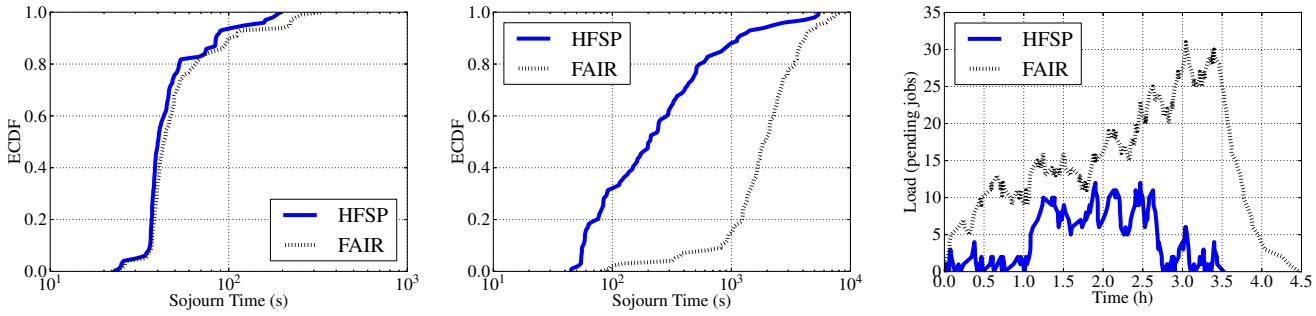
B. Macro Benchmarks

In order to evaluate the overall performance of our system, we compare FAIR with HFSP on sojourn time – the interval between a job's submission and its completion – and load, in terms of number of pending jobs (i.e., those that have been submitted and not yet completed). Table II shows mean sojourn time (across all jobs) and mean load (over the duration of the experiment) for our two workloads.

In the SMALL workload, HFSP decreases the mean sojourn time by around 16%. By observing the empirical cumulative distribution function (ECDF) of sojourn times in Figure 3(a), we notice larger differences between FAIR and HFSP for jobs with longer sojourn times (note the logarithmic scale on the x axis). In this workload, the system is on average loaded with around 2 pending jobs (see Table II); since these jobs are often small, the system is generally able to allocate all tasks of pending jobs, resulting in analogous scheduling choices (and therefore sojourn time) for both FAIR and HFSP. However, when system load is higher, HFSP outperforms FAIR.

Our results are strikingly different for the LARGE workload (Figure 3(b)), where the mean sojourn time with HFSP is less than a quarter of the one with FAIR. In this workload, most jobs require several task slots, and complete more quickly since HFSP awards them the entire cluster (if needed) when they are scheduled. Instead, the sharing strategy of FAIR has the drawback of increasing the sojourn time of all jobs. MAP phases of most jobs complete earlier in HFSP, making it possible to schedule REDUCE phases sooner than with FAIR. As a result, with HFSP, 30% of jobs complete within 100 seconds from their submission, while in the same time window FAIR only completes 2% of them; after 1,000 seconds from submission, 90% of jobs are completed with HFSP while only 15% are completed with FAIR.

Scheduling choices are more critical when the cluster is loaded by jobs that require many resources, and the difference between the SMALL and LARGE workloads exemplifies this clearly. Figure 3(c) shows the evolution of load run on the LARGE workload: even if the job submission schedule for HFSP and FAIR is the same, load is promptly decreased in HFSP by focusing resources on single jobs. The fact that



(a) Sojourn time for the SMALL workload. (b) Sojourn time for the LARGE workload. (c) Cluster load for the LARGE workload.

Fig. 3. Macro benchmark results.

scheduling becomes more critical in situations of high load is indeed confirmed by our simulation results [18].

These results allow us to conclude that HFSP performs better than FAIR in two very different workloads; the advantage is more pronounced when the job and workload size is large with respect to the cluster size. In that case, scheduling decisions become critical, and the inefficiencies of simple fair sharing become apparent.

C. Estimation Errors and Sojourn Times

We have shown that HFSP outperforms FAIR, in particular when applied to clusters with high load and heavy jobs. Next, we characterize estimation errors we measured in our experiments and discuss their impact on job sojourn times, in light of our initial analysis of scheduling performance, discussed in Section II-B.

In our experiments, task times are clearly skewed. Figure 4 shows the distribution of task times measured for all our experiments: most tasks complete within few tens of seconds, but around 10% of REDUCE tasks and a non-negligible number of MAP tasks need orders of magnitude more time to complete. As such, we now characterize the job size estimation errors induced by our sampling-based technique. As done in Section II-B, when s_i is the real size of the job and $\hat{s}_i^k$ the estimated size obtained using k sample tasks, we define the estimation error as $\varepsilon_i^k = \hat{s}_i^k / s_i$: $\varepsilon_j^k < 1$ means that job size is under-estimated, whereas $\varepsilon_j^k > 1$ in case of over-estimation.

Figure 5 shows the ECDF of estimation errors across all our experiments, for our setting of $k = 5$ sample tasks. The empirically observed error distribution maps well to the log-normal distribution we use in Section II-B. Using a maximum likelihood fitting method, we approximate the error distribution as Log-$\mathcal{N}(\mu, \sigma^2)$: for MAP, $\mu = 0.0976$ and $\sigma = 0.411$; for REDUCE, $\mu = 0.0878$ and $\sigma = 0.228$. The Kolmogorov-Smirnov goodness of fit test does not reject the fitting at a significance level of 0.05.

It is impossible to evaluate HFSP in a real deployment and in the complete absence of estimation errors, since execution time of a given job in Hadoop varies at each run, according to complex and rather unpredictable system properties [12], [28]. To isolate the impact of errors on scheduling and sojourn

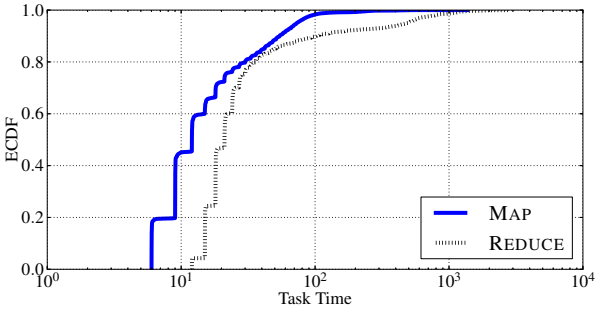


Fig. 4. Task time distribution.

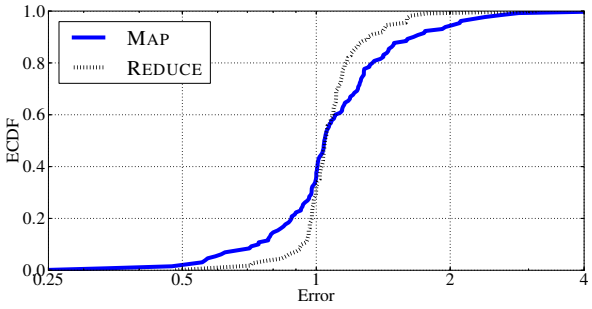


Fig. 5. Estimation error ε_i^k for $k = 5$.

time, we thus turn to our simulation results on SRVT which is a size-based scheduler with aging induced by fair sharing in virtual time. It thus can be seen as a model for HFSP which abstracts from the intricacies of a real system deployment.

In our observed errors, the estimation module tends to slightly over-estimate job sizes (i.e., $\mu > 0$ and error distributions are not exactly centered on 1): this results in marginally slower job aging with respect to the $\mu = 0$ case. Moreover, the σ values suggest that the impact of size estimation errors on sojourn time are small: as shown in Figure 2(c) on page 3, for log-normal error distributions with $\sigma < 0.5$, SRVT achieves a mean sojourn time which is close to the one obtained with no estimation errors.

We believe that HFSP is likely to be similarly tolerant to such errors. Indeed, the main difference between SRVT and HFSP is that the latter operates in an environment that

has to deal with the constraints imposed by Hadoop, such as data locality, task granularity, and dependencies between MAP and REDUCE phases (see Section III-B). These constraints limit the degrees of freedom available to the scheduler, and result in cases where HFSP will take the same scheduling choices regardless of estimation errors. With analogous error distributions and *more* possibility to deviate from optimal behavior, SRVT achieves near-optimal mean sojourn time: this suggests that, while HFSP could certainly benefit from more sophisticated and accurate size estimation methods, further improvements in sojourn times are likely to be marginal.

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 [29]. 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 [30], [31]. 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 [16] is a prominent example to which we compare our results. Several other works [32]–[35] focus on resource allocation and strive at achieving fairness across jobs, but do not aim at optimizing sojourn times. Sandholm and Lai [36] 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 [37] address the problem of scheduling jobs to meet user-provided deadlines, but assume job runtime to be an input to the scheduler.

Flex [38] is a size-based scheduler for Hadoop which is available as a proprietary commercial solution. 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 [9]–[12] propose techniques to estimate query sizes in recurring jobs. Agarwal *et al.* [11] 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 [39] 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.* [13] 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.* [40], [41] 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 [42], 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 [43] "applications" to run concurrently. Monolithic schedulers such as YARN [44] and Omega [45] use a single component to allocate resources to each framework, while two-level schedulers [46], [47] 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

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 a non-negligible fraction of interactive data processing tasks. As a consequence, we have witnessed the raise of deployment best practices in which long sojourn times – due to a fair sharing of resources among competing jobs – were compensated by over-dimensioned Hadoop clusters. In addition, we remarked that an efficient cluster utilization could be approximated through a tedious manual exercise, involving the creation of static resource pools to accommodate workload diversity and an important tuning effort.

To overcome such limitations, in this work we set off to study the benefits of a new scheduling discipline that targets at the same time short sojourn times and fairness among jobs. We thus proposed a size-based approach to scheduling jobs

in Hadoop, which we called HFSP. Our work brought up several challenges: evaluating job size on-line without wasting resources, avoiding job starvation both on small and large jobs, and guaranteeing short sojourn time despite estimation errors were the most noteworthy. We solved these problems in the context of a multi-server system using virtual time and aging, that is built to be tolerant to failures, scale-out upgrades, and supports the composite job structure of MapReduce.

We showed that a size-based discipline such as HFSP performs very well, and that a precise job size information is not essential for the scheduler to function properly. Our experimental results, in which we compared HFSP to the widely used FAIR scheduler, indicate that both interactivity and efficiency requirements were largely met: both small and large jobs do not starve, and the job sojourn time distribution is consistently in favor of HFSP. Our work has practical consequences as well: HFSP is simple to configure, and allows resource "pools" to be consolidated because workload diversity is intrinsically accounted for by the size-based discipline.

Our future work is related to job preemption. We are currently investigating a novel technique to fill the gap between killing running tasks and waiting for tasks to finish. Indeed, killing a task too late is a huge waste of work, and waiting for a task to complete when it just started is detrimental as well. Our next goal is thus to provide a new set of primitives to suspend and resume tasks to achieve better preemption.

Acknowledgements

This work has been partially supported by the EU projects BigFoot (FP7-ICT-223850) and mPlane (FP7-ICT-318627).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. of USENIX OSDI*, 2004.
[2] 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.
[3] K. Ren *et al.*, "Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads," in *Proc. of VLDB*, 2013.
[4] Apache, "Oozie Workflow Scheduler," http://oozie.apache.org/.
[5] ——, "Hadoop: Open source implementation of MapReduce," http://hadoop.apache.org/.
[6] E. Friedman and S. Henderson, "Fairness and efficiency in web server protocols," in *Proc. of ACM SIGMETRICS*, 2003.
[7] 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.
[8] M. Harchol-Balter *et al.*, "Size-based scheduling to improve web performance," *ACM TOCS*, vol. 21, no. 2, 2003.
[9] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for MapReduce environments," in *Proc. of ICAC*, 2011.
[10] ——, "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.
[11] S. Agarwal *et al.*, "Re-optimizing Data-Parallel Computing," in *Proc. of USENIX NSDI*, 2012.
[12] A. D. Popescu *et al.*, "Same queries, different data: Can we predict query performance?" in *Proc. of SMDB*, 2012.
[13] C. Tian *et al.*, "A dynamic MapReduce scheduler for heterogeneous workloads," in *Proc. of IEEE GCC*, 2009.
[14] D. Lu, H. Sheng, and P. Dinda, "Size-based scheduling policies with inaccurate scheduling information," in *Proc. of IEEE MASCOTS*, 2004.
[15] Y. Chen *et al.*, "Statistical workload injector for MapReduce," https://github.com/SWIMProjectUCB/SWIM.
[16] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of ACM EuroSys*, 2010.
[17] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. of IEEE MASCOTS*, 2011.
[18] M. Dell'Amico, "A simulator for data-intensive job scheduling," EURECOM, Tech. Rep. RR-13-282, 2013.
[19] J. Nagle, "On packet switches with infinite storage," *Communications, IEEE Transactions on*, vol. 35, no. 4, 1987.
[20] S. Gorinsky and C. Jechlitschek, "Fair efficiency, or low average delay without starvation," in *Proc. of IEEE ICCCN*, 2007.
[21] Apache, "Hadoop wiki, powered by," http://wiki.apache.org/hadoop/PoweredBy.
[22] 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.
[23] Apache, "Hadoop fair scheduler," http://hadoop.apache.org/docs/stable/fair_scheduler.html.
[24] ——, "Hadoop MapReduce JIRA 1184," https://issues.apache.org/jira/browse/MAPREDUCE-1184.
[25] ——, "PigMix," https://cwiki.apache.org/PIG/pigmix.html.
[26] TPC, "Tpc benchmarks," http://www.tpc.org/information/benchmarks.asp.
[27] M. Pastorelli *et al.*, "Practical size-based scheduling for MapReduce workloads," *CoRR*, vol. abs/1302.2749, 2013.
[28] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. of USENIX OSDI*, 2010.
[29] K. Fox and B. Moseley, "Online scheduling on identical machines using SRPT," in *In Proc. of ACM-SIAM SODA*, 2011.
[30] H. Chang *et al.*, "Scheduling in MapReduce-like systems for fast completion time," in *Proc. of IEEE INFOCOM*, 2011.
[31] B. Moseley *et al.*, "On scheduling in map-reduce and flow-shops," in *In Proc. of ACM SPAA*, 2011.
[32] T. Sandholm and K. Lai, "MapReduce optimization using regulated dynamic prioritization," in *Proc. of ACM SIGMETRICS*, 2009.
[33] M. Isard *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *Proc. of ACM SOSP*, 2009.
[34] A. Ghodsi *et al.*, "Dominant resource fairness: Fair allocation of multiple resources types," in *Proc. of USENIX NSDI*, 2011.
[35] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.
[36] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in Hadoop," in *Proc. of JSSPP*, 2010.
[37] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *Proc. of CloudCom*, 2010.
[38] J. Wolf *et al.*, "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," in *Proc. of ACM MIDDLEWARE*, 2010.
[39] Y. Kwon *et al.*, "Skewtune: mitigating skew in MapReduce applications," in *Proc. of ACM SIGMOD*, 2012.
[40] J. Tan, X. Meng, and L. Zhang, "Delay tails in MapReduce scheduling," in *Proc. of ACM SIGMETRICS*, 2012.
[41] ——, "Performance analysis of coupling scheduler for MapReduce/Hadoop," in *Proc. of IEEE INFOCOM*, 2012.
[42] Apache, "Hadoop capacity scheduler," http://hadoop.apache.org/docs/stable/capacity_scheduler.html.
[43] M. Zaharia *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.
[44] Apache, "Hadoop nextgen MapReduce (yarn)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[45] M. Schwarzkopf *et al.*, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of EuroSys*, 2013.
[46] B. Hindman *et al.*, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.
[47] Apache, "Hadoop on demand," http://hadoop.apache.org/docs/stable/hod_scheduler.html.