On the Usability of Shortest Remaining Time First Policy in Shared Hadoop Clusters

Nathanaël Cheriere, Pierre Donat-Bouillud, Shadi Ibrahim, Matthieu Simonin

On the Usability of Shortest Remaining Time First Policy in Shared Hadoop Clusters

Nathanaël Cheriere
ENS Rennes / IRISA
Rennes, France
nathanael.cheriere@ens-rennes.fr

Pierre Donat-Bouillud
ENS Rennes / IRISA
Rennes, France
pierre.donat-bouillud@ens-rennes.fr

Shadi Ibrahim
Inria Rennes - Bretagne Atlantique
Rennes, France
shadi.ibrahim@inria.fr

Matthieu Simonin
Inria Rennes - Bretagne Atlantique
Rennes, France
matthieu.simonin@inria.fr

ABSTRACT

Hadoop has been recently used to process a diverse variety of applications, sharing the same execution infrastructure. A practical problem facing the Hadoop community is how to reduce job makespans by reducing job waiting times and execution times. Previous Hadoop schedulers have focused on improving job execution times, by improving data locality but not considering job waiting times. Even worse, enforcing data locality according to the job input sizes can be inefficient: it can lead to long waiting times for small yet short jobs when sharing the cluster with jobs with smaller input sizes but higher execution complexity. This paper presents *hSRTF*, an adaption of the well-known Shortest Remaining Time First scheduler (i.e., SRTF) in shared Hadoop clusters. *hSRTF* embraces a simple model to estimate the remaining time of a job and a preemption primitive (i.e., kill) to free the resources when needed. We have implemented *hSRTF* and performed extensive evaluations with Hadoop on the Grid'5000 testbed. The results show that *hSRTF* can significantly reduce the waiting times of small jobs and therefore improves their makespans, but at the cost of a relatively small increase in the makespans of large jobs. For instance, a time-based proportional share mode of *hSRTF* (i.e., *hSRTF-Pr*) speeds up small jobs by (on average) 45% and 26% while introducing a performance degradation for large jobs by (on average) 10% and 0.2% compared to Fifo and Fair schedulers, respectively.

CCS Concepts

•Computer systems organization → Distributed architectures; •Networks → *Cloud computing;*

Keywords

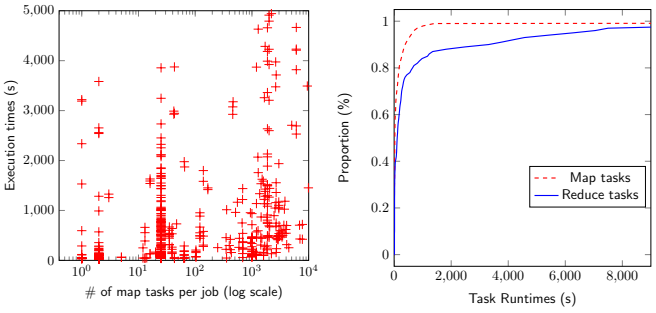
MapReduce; Hadoop; Scheduling, SRTF

1. INTRODUCTION

Large-scale data analysis has increasingly come to rely on MapReduce [5, 9] and its open-source implementation Hadoop [3]. Hadoop has recently been used to run multiple divers MapReduce applications belonging to multiple concurrent users, thanks to its built-in schedulers (i.e., Fifo, Fair and Capacity schedulers). A practical problem facing the Hadoop community is how to reduce job makespans[1], especially for *latency-sensitive* small jobs. It is reported that a typical production Hadoop cluster contains a significant fraction of small jobs (e.g., 75% of the jobs in Facebook clusters are small jobs [16]). Consequently, previous Hadoop schedulers have focused on improving job execution times by optimizing data locality (i.e., introducing short delays and launching multiple copies of map tasks to increase the possibility of local map execution [16, 12], and data-aware virtual machine placement [10]) or reducing the cross-rack communication [4]. However, a few efforts have focused on reducing job waiting times, although waiting time is as important as execution time to improve job makespans. Evaluating and prioritizing jobs according to their input data sizes may result in long waiting times for "real" short jobs when sharing the cluster with jobs with smaller input sizes but higher execution complexity. As shown in Figure 1(a), in workload traces collected over a period of one month in three different Hadoop research clusters [11, 2], we find that the execution times of MapReduce applications are not strongly depending on their input data sizes. Even worse, enforcing data locality according to the input sizes can be inefficient. The runtimes of map and reduce tasks are highly heterogeneous (see Figure 1(b)): it varies between applications. To this end, in this paper we present *hSRTF*, an adaption of the Shortest Remaining Time First scheduler (i.e., *SRTF*) in shared Hadoop clusters. *SRTF* is a well-known scheduling policy which favors tasks with short remaining times to finish and execute them first to reduce their waiting times [6]. However, several challenges arise when adopting *SRTF* to Hadoop including: preserving data locality, estimating job remaining times, and so on. To this end, this paper makes the following contributions:

[Contribution 1] We propose *hSRTF*, an adoption of the well-known *SRTF* scheduling policy in shared Hadoop clusters. *hSRTF* embraces a simple model to estimate the remaining time of a job to complete and thus to allocate as much resources as possible to satisfy its computation needs. To meet the dynamicity of Hadoop clusters (i.e., changes in

[1]In this paper, **job makespan** is the time between the submission and the completion time of a job. **Job waiting time** is the time between the submission time of the job and the starting time of the first map task. **Job execution time** is the time between the starting time of the first map task and the completion time of the job.



(a) Distribution of job execution times with respect to their input data sizes (number of map tasks): *There is no strong correlation between job execution times and job input sizes. Even worse for the same input size (for some cases), there is a factor of 40 difference in execution time between the shortest and the longest jobs.*

(b) CDFs of the task runtimes (map and reduce tasks): *Even though all map tasks have the same input size, their runtimes vary by 880% which is due to the complexity of the map task (the characteristic of the application) and data locality. On the other hand, the runtimes of reduce tasks experience a variation of 500%.*

Figure 1: Analysis of Workload traces from three different research clusters (which covers one month duration "January 2011"): (a) Distribution of job execution times and (b) CDFs of the task runtimes.

the resource utilization according to the arrival and types of jobs), *hSRTF* periodically estimates remaining times of running jobs and therefore keeps the estimation up-to-date. A preemption primitive (i.e., kill) to free the resources is adopted in *hSRTF*. Moreover, we have employed a technique to co-locate map and reduce tasks of the same jobs to reduce the need of data transfer during the shuffle phase.

[**Contribution 2**] A time-based proportional share mode of *hSRTF* is proposed in which resources are partitioned according to the remaining times of running jobs. This is an important feature in *hSRTF* especially when jobs have similar remaining times. Furthermore, it improves the time estimation by allowing a few tasks of each job to run.

[**Contribution 3**] We implement *hSRTF* as a pluggable scheduler in Hadoop and performed extensive evaluations with Hadoop on the Grid'5000 testbed. The results show that *hSRTF* can significantly reduce the waiting times of small jobs and therefore improves their makespans, but at the cost of a relatively small increase in the makespans of large jobs. For instance, a time-based proportional share mode of *hSRTF* (i.e., *hSRTF-Pr*) speeds up small jobs by (on average) 45% and 26% while introducing a performance degradation for large jobs by (on average) 10% and 0.2% compared to Fifo and Fair schedulers, respectively.

It is important to mention that the focus of this paper is to investigate the usability of the well-known scheduling policy (*SRTF*) in shared Hadoop clusters. Towards this goal, we have discussed a possible adoption of *SRTF* to Hadoop. This in turn allows us to study the impact of reducing waiting times on the job makespans.

Paper Organization. The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 discusses the design of *hSRTF* scheduler. We then describe our experimental methodology and results in Sections 4 and 5. Finally, we conclude the paper and propose our future work in section 6.

2. RELATED WORK

Job scheduling in Hadoop has attracted a lot of attention in the last few years. Most work has focused on improving job makespans by improving data locality and thus reducing job execution times [16, 12, 10, 7]. Zaharia *et al.* [16], have proposed a simple scheduling algorithm called delay scheduling to achieve locality and fairness in cluster scheduling. When a job, that should be scheduled next according to fairness, cannot launch a local task, it waits for a small amount of time, letting other jobs launch tasks instead. Purlieus (locality-aware resource allocation) [10] employed a static placement of virtual machines according to the distribution of the data inputs to ensure the local execution of map tasks. Considering that the majority of jobs in production Hadoop clusters are small jobs, several techniques have been proposed to improve the execution time of small jobs. Given that locality is of high importance to small jobs, Venkataraman *et al.* [12] proposed to launch multiple copies of map tasks to guarantee that they will be executed locally. All the aforementioned work targets improving the makespans of small jobs by reducing their execution times (by enforcing data locality). *hSRTF* aims at improving the makespans of small jobs by reducing their waiting times, which is as important as execution time. Moreover, some of the proposed techniques can be applied in *hSRTF* to improve data locality of small jobs and thus further reduce their makespans.

3. SHORTEST REMAINING TIME SCHEDULING IN HADOOP

SRTF is a preemptive scheduling algorithm in which the process with the smallest amount of remaining time until completion is selected to execute [6]. However, several challenges arise when adopting *SRTF* to Hadoop including: preserving data locality (when needed), estimating job remaining times, and so on. Hereafter, we discuss how we addressed these challenges, starting by introducing the design principles of *hSRTF*.

3.1 Design Principles

We designed and implemented *hSRTF* with the following goals in mind:

One size doesn't fit all: To reduce the makespans of small jobs (which counts for almost 75% of the jobs in Facebook clusters [16]), current schedulers tend to evaluate and therefore prioritize jobs according to their input data sizes (i.e., the number of map tasks). They assume that tasks (e.g., map tasks) belonging to different jobs have the same runtime. This assumption is not necessary true since the runtime of a task differs between applications: even though they compute the same amount of data (i.e., map tasks are usually performed on a fixed-size block of data, 128MB in Hadoop), a complicated map (reduce) function will probably take more time to finish than a simple map (reduce) function (As shown in Figure 1). Even worse, as reported in [4, 16], shuffle-heavy jobs count for almost 60% and 20% of the total jobs in Yahoo! and Facebook clusters, thus, the makespans of these jobs are dominated by the completion of the last reduce task. *hSRTF* thus evaluates jobs accord-

ing to their remaining time which can better identify "real" small jobs.

Avoid blocked jobs: Considering a simple example of two jobs: Job_1 has a small size input data but heavy shuffle and reduce phase and Job_2 has a slightly bigger size input data and light reduce phase. Traditional schedulers try to allocate map and reduce slots to Job_1 to finish fast. However, even if the map phases of Job_1 and Job_2 are completed, Job_2 will be blocked waiting for the long reduce tasks of Job_1 to be completed. This results in a long makespan of some small jobs as discussed in [14]. *hSRTF* periodically evaluates the remaining times of running jobs and therefore allocates slots to jobs accordingly. This in turn reduces the overall blocking time, especially for short jobs.

Reduce waiting time: Previous Hadoop schedulers focus on improving job execution times, by improving data locality but do not consider job waiting times. Enforcing data locality according to the job input sizes can lead to long waiting times for small and short jobs. *hSRTF* allocates resources according to the shortest remaining times and thus ensures shorter waiting time for small jobs.

Preserving data Locality: Local execution of map tasks is crucial for Hadoop performance. *hSRTF*, by dedicating all the resources to jobs with the shortest remaining times, allows these jobs to expose the native data locality implemented in Hadoop and therefore can preserve similar data locality to the Fifo scheduler. We are now working on improving data locality, especially for small jobs, by making the preemption technique data-aware.

Resource utilization: *hSRTF* co-locates map and reduce tasks to run on the same set of nodes (on the same node if possible), this reduces the need to transfer the intermediate data (reduce inputs), and thus reduces the execution times of small jobs as it leverages in-memory data processing: reduce function computes the map output which is already buffered in the memory of the same node [8].

3.2 Estimating job remaining times

A typical MapReduce application consists of two (overlapped) phases: a map phase and a reduce phase. However, an accurate estimation of the execution time of a MapReduce application is a very hard task. It is contributed to by many factors including the size of input data, map complexity, reduce complexity, size of intermediate data, and the time in which map and reduce phases overlap. These factors vary according to the type of MapReduce applications, the underlying infrastructure, and the distribution of the data blocks. Therefore profiling the execution time could be costly, especially with the proliferation of MapReduce applications. Several efforts have therefore targeted predicating the execution time of the map and reduce phase separately and exploit the predication to optimize scheduling in Hadoop [13].

In *hSRTF*, we have developed a simple model to estimate job remaining times, as shown below:

$$remaining\_time = \left\lceil \frac{map\_unfinished}{map\_capacity} \right\rceil * avg\_map\_time + \left\lceil \frac{reduce\_unfinished}{reduce\_capacity} \right\rceil * avg\_reduce\_time \quad (1)$$

This equation is supported by the following assumptions:

[1] The remaining time needed by a job is computed as if it was the only job running on the cluster to compare every job with the same parameters. **[2]** The number of map and reduce tasks a job can launch at the same time is limited by the capacity of map slots and reduce slots of the cluster. That is why the estimation of the remaining time also takes into account those two parameters. **[3]** The remaining time of a job is estimated in two phases: the remaining time needed for the map phase, and the remaining time needed for the reduce phase. **[4]** During the map phase, the remaining time is dominated by the complexity of map tasks and the size of map inputs. Thus, we rely on the finished map tasks and/or on the progress score and the duration of ongoing ones to calculate avg_map_time. For the avg_reduce_time, we assume that reduce tasks have the same complexity of map tasks and therefore reduce runtime depends on the reduce input data. That is:

$$avg\_reduce\_time = \frac{input\_size\_per\_reducer}{avg\_map\_time} * block\_size \quad (2)$$

[5] During the reduce phase, the remaining time is dominated by reduce tasks only, therefore, we rely on the finished reduce tasks and/or on the progress score and the duration of ongoing ones to calculate avg_reduce_time.

When a new job is launched, we take the average values of the map and reduce tasks which belong to the currently running jobs. However, our estimation gets more realistic with time as more information about the job can be obtained. It is very important to mention that the remaining time is recomputed every 10 sec to cope with the dynamicity of currently running jobs and infrastructure.

3.3 Preemption

hSRTF uses *kill* action to free up slots for tasks belonging to jobs with the shortest remaining times. The preempted tasks are chosen from the jobs with the longest remaining time to finish. Furthermore, to minimize resources waste, tasks with the lowest progress score are selected.

3.4 Map and reduce task co-location

Given that most of the resources are dedicated to the job with the shortest remaining time, this allows to have a better placement of the reduce tasks. *hSRTF* tries to co-locate map and reduce tasks belonging to the same job. This reduces the need to send intermediate data through the network, which is normally the most scarce resource in today's datacenters: intermediate data just needs to be read from the memory to/from the disk (In case of very small jobs, data stays in-memory during map and reduce phase). To co-locate map and reduce tasks, we try to launch reduce tasks according to the location of currently running map tasks and to the distribution of the map inputs (assuming map tasks will eventually run locally).

3.5 Time-based proportional sharing

The *SRTF* scheduling algorithm was originally designed for monotask systems and all the resources were given to the job that would finish first. Thus in our work we have adopted the same policy: *hSRTF* always satisfies the resource needs of the job with the shortest remaining time.

However, to provide a fair share of resources when jobs have the same remaining times (e.g., large job was running for

Scheduler	Description
Fifo	Priority scheduler with respect to job submission time
Fair	Provides fair allocation of resources between different jobs
hSRTF-Pu	Allocates all resources to the job with the shortest remaining time
hSRTF-PuP	Similar to *hSRTF-Pu*, but with the possibility of preempting (kill) running tasks which belong to a job with the longest remaining time to provide early allocation to a job with the shortest remaining time
hSRTF-Pr	Allocates resources to jobs according to their remaining time
hSRTF-PrP	Similar to *hSRTF-Pr*, but with the possibility of preemption

Table 1: List of schedulers used in our evaluation

a long period and a new submitted small job) and to improve the time estimation (by allowing a few tasks of each job to run), we have implemented a time-based proportional share mode of *hSRTF*. That is, we allocate resources to jobs according to their remaining times. If a job takes n more time to finish than a reference job, it will have n time less resources than the reference job. Let us denote RT_i as the remaining time of job i and S_i its share of available resources. We have the set of equations for n jobs (0,..., $n-1$):

$$\forall i, S_i = \frac{RT_0}{RT_i} S_0 \tag{3}$$

$$\sum_{i=0}^{n-1} S_i = cluster\_resources \tag{4}$$

If a job receives more resources than what is needed (this happens at the end of a job), the used resources are redistributed to other jobs. However, there is a risk of starvation, jobs being always left behind to prioritize shorter jobs. In order to avoid such starvation, the number of resources a job receives is coefficiented with a starvation ratio ST_i:

$$ST_i = \frac{Time\_since\_launch + Estimated\_remaining\_time\_on\_share}{Estimated\_makespan\_alone} \tag{5}$$

We estimate the total makespan of the job – assuming that the current share of resources will remain the same until the end of the job (by changing map and reduce capacity in Equation 1 by S_i) – and we divide it by the estimation of the makespan of the job as if it had all the resources for itself. This ratio represents the extra time a job needs to complete when sharing the resources with other jobs. The more a job starves (its estimated total makespan increases), the more it will have resources.

4. METHODOLOGY OVERVIEW

Platform. The experiments were carried out on the Grid'5000 [1] testbed. For our experiments, we employed 58 nodes belonging to the Toulouse site of Grid'5000. These nodes are outfitted with 4-core AMD Opteron 2.6 GHz CPUs and 8 GB of RAM. Intra-cluster communication is done through a 1 Gbps Ethernet network.

Benchmarks. For our experiments, we selected two applications that are commonly used for benchmarking MapReduce frameworks: *distributed wordcount* and *distributed sort*.

Hadoop deployment. On the testbed described earlier, we configured and deployed a Hadoop cluster using the Hadoop 1.0.4 stable version [3]. The Hadoop instance consists of 58 nodes to serve as both datanodes and tasktrackers, among which, one node was also configured to serve as namenode and jobtracker. The tasktrackers were configured with 8 slots for running map tasks and 2 slots for executing reduce tasks. At the level of HDFS, we used the default chunk size of 128 MB and the default replication factor of 3 for the input and output data.

***hSRTF* Implementation.** *hSRTF* is built as a pluggable scheduler in Hadoop-1.0.4. As mentioned earlier, *hSRTF* can be tuned to operate in two different modes: pure mode represented as *hSRTF-Pu* and time-based proportional-share mode represented as *hSRTF-Pr*. Each mode may work with and without preemption. This gives us 4 different possibilities for running *hSRTF* (see Table1). By disabling preemption, we ensure a fair comparison with Fifo and Fair schedulers. Fair scheduler is configured with one pool and fairness is applied in-between the jobs within this pool.

5. EXPERIMENTS RESULTS

We run a mixed workload consisting of sort and wordcount applications. For both applications we vary the input data sizes as shown in Table 2. Each job is submitted 10 seconds after each other.

Figure 2 shows the CDFs of job makespans, waiting times and execution times for each group. *hSRTF* significantly improves the makespan of smaller jobs, but at the cost of slowing larger jobs. Hereafter we will explain these results for each group, separately, and present a detailed comparative discussion of the various used schedulers.

5.1 Small Jobs

***hSFTF-Pu vs* Fifo.** We observe that *hSRTF-Pu* outperforms Fifo for all small jobs with an average speedup of 43% (see Figure 2(a)). This improvement however is not due to the reduction in job waiting times as shown in Figure 2(d), but it is because of the improvement in the job execution times (although both achieve similar data locality, as shown in Figure 3(a)). *hSRTF-Pu* launches map and reduce tasks belonging to small jobs as soon as a slot is freed up while Fifo was not able to launch reduce tasks for small jobs though map tasks were already running (completed). Thus small jobs under Fifo were blocked due to the long occupation of reduce tasks by other running jobs, a similar issue was reported in [14].

***hSFTF-Pr vs* Fair.** On the other hand, *hSRTF-Pr* reduces the makespan of (most) small jobs with an average speedup of 26%. This improvement is obviously due to the huge reduction in waiting times as shown in Figure 2(d). Surprisingly, as shown in Figure 2(g), *hSRTF-Pr* slightly

	Application	# of maps	# of reduces	# of jobs
Large jobs	Sort	256	32	3
Medium Jobs	Sort	64	8	8
Small Jobs	Sort	1	1	10
	WordCount	1	1	10

Table 2: List of jobs and their input size used in the experiment, in the order of submission

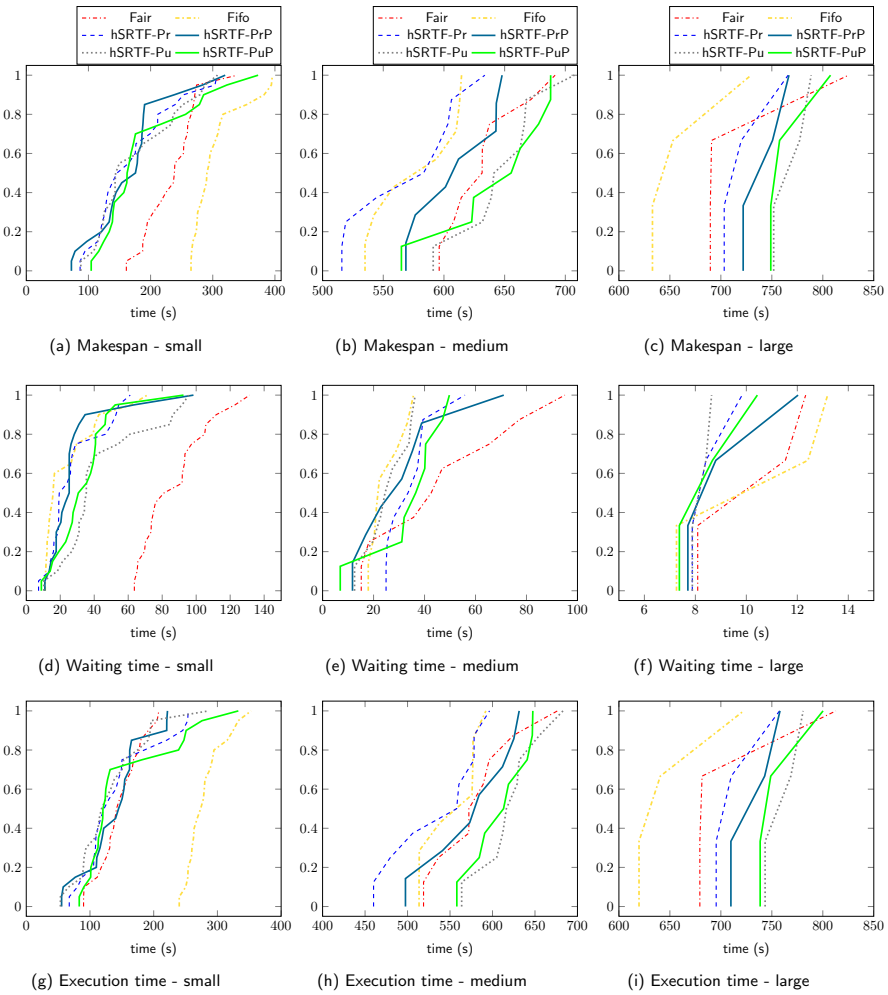


Figure 2: CDFs of job makespans, waiting times, and execution times under different schedulers, grouped by job sizes

improves job execution times, although Fair achieves perfect locality compared to *hSRTF-Pr*: as shown in Figure 3(a), Fair achieves 100% data locality while *hSRTF-Pr* obtains only 5% data locality. This can be explained due to two reasons: (1) under Fair, some jobs were blocked waiting for reduce slots to be freed up, and (2) *hSRTF-Pr* tries to co-locate map and reduce tasks to reduce data transfer during the shuffle phase, however, this was very beneficial for small jobs in our case as most jobs were executed on one node (map and reduce tasks) and therefore intermediate data were buffered in-memory. This avoids the extra latency introduced by transferring and reading data from disk, hence, this compensates the low data locality, especially for sort applications (in sort, reduce input (map output) has the same size as map input).

Preemption *vs* waiting. Finally, we observe that employing preemption can help in reducing the waiting time for most small jobs, as shown in Figure 2(d). *hSRTF-PrP* and *hSRTF-PuP* reduce the job waiting times compared to *hSRTF-Pr* and *hSRTF-Pu*, respectively. However, job makespans were not reduced because in some jobs, a reduce slot was freed up on a node different from the node running job's map task and thus the execution time increased.

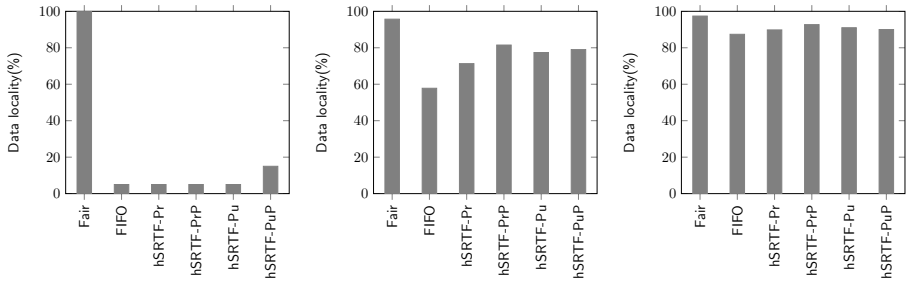
5.2 Medium Jobs

We observe that *hSRTF-Pr* is superior to all other schedulers (see Figure 2(b)). This is due to the reduction in job execution times and the moderate waiting times, as shown in Figure 2(h) and Figure 2(e). Here, as resources are allocated according to the shortest remaining time, medium jobs continue to run with minimal resources and even with maximum resources if their remaining time is shorter than small jobs. On the other hand, Fair still results in longer waiting time and therefore in longer makespans compared to both *hSRTF-Pr* and *hSRTF-PrP*.

Medium jobs suffer the same waiting times under both Fifo and *hSRTF-Pu*. However, *hSRTF-Pu* tries to allocate resources to small jobs and thus prolongs the execution time of medium jobs. What's more, the reduction of small job waiting times by exploiting preemption in *hSRTF-PuP* results in more waiting time compared to *hSRTF-Pu*.

5.3 Large Jobs

By trying to allocate and free up slots (i.e., when employing preemption) to allow small jobs to execute their tasks, large jobs will be blocked and will lose the work of their running tasks, if preempted. *hSRTF* therefore increases the execu-



(a) Data locality - small (b) Data locality - medium (c) Data locality - large

Figure 3: Data Locality under different schedulers, grouped by job sizes

tion time of large jobs and therefore increases job makespans (see Figure 2(i) and Figure 2(c)), despite the relatively small reduction in job waiting times, as shown in Figure 2(f). Moreover, preemption adversely impacts the performance of large jobs. For example, *hSRTF-Pr* outperforms *hSRTF-PrP* for all large jobs.

5.4 Discussion

Improving job makespans can be achieved by reducing their execution times and waiting times. Our results demonstrate that *hSRTF* significantly reduces job waiting times compared to Fifo and Fair schedulers, regardless of the job sizes (as shown in Figure 2(d), Figure 2(e), and Figure 2(f)). This in turn results in a reduction in the makespans of small jobs: besides the reduction in job waiting times, *hSRTF* also reduces job execution times, thanks to the map and reduce tasks co-location techniques. Unfortunately, the obtained data locality is very low (as shown in Figure 3(a), it varies between 5% to 15% according to the applied *hSRTF* mode). *hSRTF* achieves better data locality for both medium and large jobs (see Figure 3(b) and Figure 3(c)). The makespans of these jobs vary according to the reduction/increase in their waiting times, the effectiveness in the co-location technique, the arrival of small jobs (i.e., the time when resources are claimed by jobs with shorter times), and the number of preempted tasks. Under *hSRTF*, the reduction in makespans of small jobs comes at the cost of a slight increase in makespans of large jobs. This increase in job makespans is higher when employing the preemption technique. However, using *hSFTR-Pr* leads to lowering the increase of the makespans of large jobs as it allows them to continue their executions, but with minimal resources share.

6. CONCLUSION AND FUTURE WORK

In this research work we investigate the usability of the Shortest Remaining Time First scheduling policy in shared Hadoop clusters. To do so, we have proposed an adaption of *SRTF* in Hadoop (named *hSRTF*). *hSRTF* embraces a simple model to estimate the remaining time of a job and a preemption primitive (i.e., kill) to free the resources when needed. Furthermore, a time-based proportional shared version of *hSRTF* is discussed. The results show that *hSRTF* can significantly reduce the waiting times of small jobs and therefore improves their makespans, but at the cost of a relatively small increase in the makespans of large jobs. Our future work lies in two aspects. First, to improve data locality in *hSRTF*: we plan to investigate several existing techniques to enforce data locality of small jobs [16, 7] and design a smarter preemption policy that considers data locality when preempting tasks to free up slots for jobs with short remaining times. Second, we plan to explore existing preemption techniques [15, 14] that could save the state of preempted tasks instead of simply killing them which in turn will result in lower resource waste.

7. ACKNOWLEDGMENTS

The Corresponding Author is Shadi Ibrahim (shadi.ibrahim@inria.fr). The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details).

8. REFERENCES

[1] Grid 5000 Project. https://www.grid5000.fr/mediawiki/index.php.
[2] Hadoop Log Dataset. http://www.pdl.cmu.edu/HLA/, 2015.
[3] The Apache Hadoop Project. http://www.hadoop.org, 2015.
[4] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *USENIX ATC'14*.
[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[6] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2):207–233, May 2003.
[7] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu. Maestro: Replica-aware map scheduling for mapreduce. In *IEEE/ACM CCGrid'12*.
[8] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *IEEE CLOUDCOM'10*.
[9] H. Jin, S. Ibrahim, L. Qi, H. Cao, S. Wu, and X. Shi. The mapreduce programming model and implementations. *Cloud Computing: Principles and Paradigms*, pages 373–390, 2011.
[10] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *ACM/IEEE SC '11*.
[11] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.
[12] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *USENIX OSDI'14*.
[13] A. Verma, L. Cherkasova, and R. H. Campbell. Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance. In *IEEE MASCOTS'12*.
[14] Y. Wang, J. Tan, W. Yu, X. Meng, and L. Zhang. Preemptive reducetask scheduling for fair and fast job completion. In *USENIX ICAC'13*.
[15] O. Yildiz, S. Ibrahim, T. A. Phuong, and G. Antoniu. Chronos: Failure-aware scheduling in shared hadoop clusters. In *IEEE BigData 2015*.
[16] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *ACM EuroSys'10*.