Preserving Fairness in Shared Hadoop Cluster: A Study on the Impact of (Non-) Preemptive Approaches

Orcun Yildiz, Shadi Ibrahim

Preserving Fairness in Shared Hadoop Cluster: A Study on the Impact of (Non-) Preemptive Approaches

Orcun Yildiz, Shadi Ibrahim

RESEARCH REPORT

N° 9384

December 2020

Project-Teams Myriads

ISSN 0249-6399 ISRN INRIA/RR--9384--FR+ENG

Preserving Fairness in Shared Hadoop Cluster: A Study on the Impact of (Non-) Preemptive Approaches

Orcun Yildiz[*], Shadi Ibrahim[†]

Project-Teams Myriads

Research Report n° 9384 — December 2020 — 19 pages

Abstract: Recently, MapReduce and its open-source implementation Hadoop have emerged as prevalent tools for big data analysis in the cloud. Fair resource allocation in-between jobs and users is an important issue, especially in multi-tenant environments such as clouds. Thus several scheduling policies have been developed to preserve fairness in multi-tenant Hadoop clusters. At the core of these schedulers, simple (non-) preemptive approaches are employed to free resources for tasks belonging to jobs with less-share. For example, Hadoop Fair Scheduler is equipped with two approaches: wait and kill. While wait may introduce a serious violation in fairness, kill may result in a huge waste of resources. Yet, recently some works have introduced new preemption approaches (e.g., pause-resume) in shared Hadoop clusters. To this end, in this work, we closely examine three approaches including wait, kill and pause-resume when Hadoop Fair Scheduler is employed for ensuring fair execution between multiple concurrent jobs. We perform extensive experiments to assess the impact of these approaches on performance and resource utilization while ensuring fairness. Our experimental results bring out the differences between these approaches and illustrate that these approaches are only sub-optimal for different workloads and cluster configurations: the efficiency of achieving fairness and the overall performance varies with the workload composition, resource availability and the cost of the adopted preemption technique.

Key-words: Clouds, Resource management, Data processing, Scheduling, Performance evaluation

Most of this work was carried out while the first author was affiliated with Inria, Univ. Rennes, CNRS, IRISA, France.

[*] Argonne National Laboratory, USA. **oyildiz@anl.gov**
[†] Univ. Rennes, Inria, CNRS, IRISA, France. **shadi.ibrahim@inria.fr**

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Préserver l'équité dans un cluster Hadoop partagé : une étude de l'impact des approches (non)-préemptives

Résumé : Récemment, le paradigme MapReduce et son implémentation open-source Hadoop sont devenus des outils très populaires pour l'analyse de données massives dans le Cloud. Le partage équitable des ressources entre les différentes tâches et utilisateurs est un problème important, en particulier dans les architectures multi-tenant comme le Cloud. De nombreuses stratégies d'ordonnancement ont donc été développées pour préserver l'équité dans les cluster Hadoop multi-tenant. Au cœur de ces ordonnanceurs, des approches simples et non-préemptives sont utilisées pour libérer des ressources pour des tâches appartenant à des utilisateurs en ayant eu jusque-là une part plus faible. Par exemple, Hadoop Fair Scheduler possède deux approches : "attendre" et "tuer". Si "attendre" peut causer des sérieuses ruptures d'équité, "tuer" peut aussi entraîner un important gaspillage des ressources. Cependant, certains travaux récents ont introduit des techniques préemptives (c'est-à-dire "arrêter-reprendre") dans les clusters Hadoop partagés. Dans ce travail, nous examinons précisément trois approches, incluant "attendre", "tuer" et "arrêter-reprendre", lorsque Hadoop Fair Scheduler est utilisé pour assurer une répartition équitable des ressources lors de l'exécution de plusieurs groupes de tâches concurrents. Nous avons mené des expériences étendues pour évaluer l'impact de ces approches sur les performances et l'utilisation des ressources tout en garantissant leur partage équitable. Les résultats de nos expériences mettent en évidence les différences entre ces stratégies et montrent que chacune est sous-optimale pour une partie des workloads et des configurations : la capacité à garantir l'équité et les performances globales varient en fonction de la composition des tâches, des ressources disponibles et du coût des techniques préemptives.

Mots-clés : Clouds, organisation des ressources, calcul sur données, ordonnancement, évaluation de performances

Contents

1 Introduction

Data-intensive applications have been accelerating many services in different areas such as science, business, health care and finance. This has boosted the growth of the data volumes which needs to be handled efficiently. In this context, MapReduce [9, 15] has emerged as the prevalent paradigm for processing massive amounts of data. MapReduce is adopted in both industry and academia due to its simplicity, transparent fault tolerance and scalability. Recently, it has also been optimized to support the execution of multiple concurrent jobs belonging to multiple users [14, 24, 8]. By analyzing the traces collected from a research cluster with 78 users [2], we have seen that there are at least 10 jobs running concurrently at 5% of the time. Sharing MapReduce clusters leads to lower cost and better utilization of resources but it introduces new challenges such as ensuring fairness among different users and jobs. While fairness (i.e., giving each job their fair share of resources) is an important aspect in multi-tenant clusters, achieving fair execution in MapReduce clusters is not trivial. One limiting factor is the static resource management model of MapReduce clusters. A cluster consists of workers and each worker has a fixed number of slots (i.e., slots represent the capacity of a worker) which can run tasks. Depending on the available resources, duration of tasks and the number of running tasks; it might not always be possible to provide fair share of resources for different users.

To this end, several multi-tenant schedulers have been proposed to solve the problem of fairness. In this context, schedulers assign the resources in multi-tenant environments to the users depending on their specified fairness policy. For example, Quincy [14], fair scheduler for Dryad, uses min-cost flow algorithm, while Hadoop Fair Scheduler (HFS) [24] uses max-min fair sharing strategy in order to operate shared MapReduce clusters towards fairness. While they apply different solutions for the fairness problem, they both employ similar approaches when the cluster status changes (e.g., new job arrives, running job(s) finishes) for ensuring fairness. Current supported approaches in MapReduce frameworks are twofold: 1) wait for running tasks until their completion, 2) kill the tasks belonging to the jobs with over-share to allocate resources for new jobs. For instance, HFS adopts wait approach by default and leaves kill approach optional to users in case fair share of some users are not met.

Kill approach is simple and efficient in deallocating the resources but it wastes the resources by destroying the on-going work of the killed tasks. On the contrary, wait approach is waste-free but it might not be efficient when cluster contains long running tasks. Besides wait and kill approaches, preemption approach (i.e., pause-resume) can be also applied to enforce the fairness[1]. It is simply pausing the running tasks to make room for new tasks and resume the paused tasks where they left off when there is an available slot. In this work, we seek to provide a deep look into the different characteristics of the three different approaches (i.e., wait, kill and preemption) while enforcing fairness. To this end, we adopted the preemption approach on top of Hadoop Fair Scheduler, which was originally developed in [23]. We perform extensive experiments by using different workloads including real-life workloads from Facebook production clusters. We make the following contributions:

- *A study of the fairness in multi-tenant environments with respect to the three different approaches.* We find that preemptive action might be necessary in order to enforce fairness among users. Our results show that there is a serious fairness violation with wait approach in some scenarios. Specifically, we show evidence that wait approach is not able to provide equal distribution of the cluster resources, especially for the reduce slots usage due to long running reduce tasks. Similarly, Wang et al. [21] also indicated the fact that long running

[1] For simplicity, hereafter preemption approach refers to the pause-resume technique which was originally developed in [23].

(reduce) tasks may lead to starvation of short jobs and they introduced a technique for reduce task preemption in order to favor short jobs against long jobs.

- *An analysis of the performance and resource utilization under three different approaches.* We find that fairness violation not only degrades the performance of the jobs with less-share but also results in overall performance degradation. For resource utilization, our results show that kill approach results in noticeable resource waste (i.e., 11% resource loss in terms of slot usage) by destroying the on-going work of the killed tasks although it is able to enforce the fairness. Moreover, we observe that this introduces a performance penalty for MapReduce applications.

- *An analysis of the impact of resource availability on the effectiveness of three different approaches.* We discover that there is a correlation between the resource availability and the effectiveness of these approaches. In particular, we observe that wait outperforms other approaches when the cluster resources are only *fully* occupied for a short time and therefore slots (resources) are freed faster thus the tasks belonging to jobs with less-share wait for a small amount of time. We also find that kill and preemption approaches can be disadvantageous in low-utilized clusters. The main reason behind this is the re-execution of the killed tasks with kill approach and the introduced delay for the preempted tasks with preemption approach as described in Section 5.2.

The proposed work aims at providing a clearer understanding of the interplay between different (non-) preemptive approaches and fairness, performance and resource utilization. Moreover, it is important to mention that these approaches are applicable beyond the fairness problem. Therefore, users can benefit from our findings in different areas such as ensuring QoS requirements or improving the energy efficiency of MapReduce clusters.

The paper is organized as follows: Section 2 presents the background of our study. We discuss the three approaches: wait, kill and preemption in Section 3. Section 4 describes an overview of our methodologies, followed by the experimental results in Section 5. Finally, we conclude the paper and propose our future work in Section 6.

2 Background

In this section, we first provide a brief overview of Hadoop [4] and its fair scheduler, then we present the earlier works on fairness and preemption in MapReduce.

2.1 Overview of Hadoop

In Hadoop, job execution is performed with a master-slave configuration. JobTracker, Hadoop master node, schedules the tasks to the slave nodes and monitors the progress of the job execution. TaskTrackers, slave nodes, run the user defined map and reduce functions upon the task assignment by the JobTracker. Each TaskTracker has a certain number of map and reduce slots which determines the maximum number of map and reduce tasks that it can run. Communication between master and slave nodes is done through heartbeat messages. At every heartbeat, TaskTrackers send their status to the JobTracker [12]. Then, JobTracker will assign map/reduce tasks depending on the capacity of the TaskTracker and also by considering the locality of the map tasks [9, 13] (i.e., Among the TaskTrackers with empty slots, the one with the data on it will be chosen for the map task).

2.2 Overview of Hadoop Fair Scheduler

Hadoop Fair Scheduler (HFS) [1] is designed for multi-tenant clusters with fairness in mind. HFS groups jobs into *pools* and tries to achieve fairness between these pools. Furthermore, each pool can either adopt FIFO or fair sharing as its internal scheduling policy. In FIFO scheduling, jobs are prioritized according to their submission times. Therefore, if FIFO scheduling policy is adopted, there will be only pool-based fairness. On the other hand, job-based fairness can be enforced by HFS with the adoption of fair sharing as the internal scheduling policy of the pools. In HFS, user name is the default parameter for organizing jobs among the pools. Hence, HFS creates one pool for each user in the cluster.

While enforcing fairness between different jobs/pools, HFS checks the number of running tasks for each pool. When there is a free slot, HFS allocates this slot to the pool with the fewest number of running tasks. Then, depending on the internal policy of the pool, this slot will be given to the appropriate task. Moreover, HFS defines a *fair share* which it calculates by max-min fair sharing algorithm [18]. Normally, the pools with the running tasks should have their fair share in the cluster, which is the main goal of the Fair Scheduler. However, due to the dynamic nature of the multi-tenant clusters (e.g., new users start to submit jobs, earlier jobs finish their execution), active pools might have different number of resources than their fair share. To overcome this, HFS provides a kill technique (which needs to be enabled via configuration file), that kills the most recently launched jobs from the pools with over-share to give these resources to the pools with less-share than their fair share. HFS also allows users to adjust the timeout for this comparison (i.e., whether there are pools with over-share) which is defined as *fair share preemption timeout* in HFS.

2.3 Fairness and preemption in MapReduce

Several research efforts have been introduced with the aim of achieving fairness in the multi-tenant MapReduce clusters [14, 21, 11, 16, 19, 10, 17]. (Non-) preemptive approaches are employed at the core of each of these solutions to provide fairness while guaranteeing good performance and resource utilization. Wait approach is employed in Hadoop Fair scheduler [1] and ShuffleWatcher [6]. To compensate the performance degradation which may be caused by waiting for free resources (especially for short jobs), Hadoop Fair scheduler embraces a simple technique, called delay scheduler [24], to improve local execution of map tasks: when a scheduled job is not able to launch a local task, it waits for some time until it finds the chance to launch a task locally. Quincy [14] treats scheduling as an optimization problem and uses min-cost flow algorithm to achieve the solution. It employs the kill approach to set the cluster according to the configuration in the solution. Chronos [23, 22] is a failure-aware scheduler which introduces a preemption technique in order to improve MapReduce performance under failures while ensuring fairness. Wang et al. [21], leverage the fact that long running reduce tasks may lead to starvation of short jobs. Thus, they introduce a preemption technique in order to achieve fairness by favoring short jobs against long jobs. Our work is complementary to these studies, in the sense that it presents a deep and comprehensive analysis of the impact of the different approaches (including wait, kill and preemption that these works employ to achieve fairness) not only on the fairness guarantees but also on the performance of MapReduce applications and the resource utilization of MapReduce clusters; and provides a useful insight on how to select the most appropriate (non-) preemptive approach to get the best practical trade-off between fairness, performance and resource utilization.

3 Overview of the Three Approaches: Kill, Wait and Preemption

Preemption is a desirable concept in scheduling and has many different use cases, especially in the operating systems domain. In MapReduce systems, one can utilize preemption for several cases such as achieving better resource utilization or fair share of resources. For instance, to deal with the fair share of the resources, schedulers should reallocate the resources between jobs when the number of jobs changes. Since resources are normally fully utilized, fresh jobs (i.e., newly arrived jobs) are most likely to suffer by having less resources than their fair share. In this case, preemption can help releasing resources from the jobs with over-share than their fair share in order to allocate resources to these jobs.

In this work, we investigate the impact of three different approaches (i.e., wait, kill and preemption) while enforcing fairness. Currently, Hadoop Fair Scheduler supports wait and kill approaches for enforcing fairness. We also adopted the preemption approach which has been previously developed in [23] by implementing it on top of Hadoop Fair scheduler.

Kill is a simple preemption technique and it can take a fast action. To achieve fairness, it deallocates resources by killing the tasks belonging to the jobs with over-share and these resources will be allocated to the jobs with less-share. Although it simplifies the preemption mechanism, kill approach destroys the on-going work of the killed tasks. Therefore, this leads to waste of resources which can have a large impact on both application performance and resource utilization. This negative impact depends on several factors such as the current progress of killed tasks and the task type. For example, this impact will be larger if killed tasks were about to finish their execution, or vice versa. In addition, if the killed task is a reduce task, copy of the killed task has to transfer all the intermediate mapper outputs when it is launched. This in turn can further impact the MapReduce system negatively due to the extra cost of the data transfer through network, a well-known source of overhead in today's data-centers. Wait approach is the default strategy used by HFS for enforcing fairness. It simply does not take any action for allocating resources to fresh jobs and waits until currently running tasks finish their execution to give resources to the fresh jobs. While this approach can be efficient when the running tasks are short, it can introduce a serious fairness violation in the presence of long-running tasks which needs to be preempted. Besides wait and kill approaches, another strategy for enforcing fairness would be applying the preemption approach. Surprisingly, this approach is not supported by Hadoop or other MapReduce frameworks. In brief, it pauses the currently running tasks to allocate resources for new jobs and resumes them where they left off when there is an available slot. This preemption technique should be light-weight in order to be efficient. For instance, applying a naive checkpointing approach will introduce a large overhead since it requires flushing all the data associated with the preempted task to the disks.

For clarity, these three approaches (i.e., wait, kill and preemption) are displayed in Figure 1. Here, shorter task, *t2*, arrives later than the longer one, *t1*. We can see that *t2* starts its execution after *t1* finishes its execution. Here, waiting time of *t2* is equal to the remaining time of *t1* until its completion. With kill approach, we can observe that the on-going work of *t1* is wasted and this leads to the longest execution time in this scenario since this wasted amount of work needs to be re-executed after the completion of *t2*. For the preemption case, we can see that the work has been done conserved at the moment of the preemption. After *t2* finishes its execution, *t1* continues its execution where it left off. Here, Δt represents the overhead that can be caused by the preemption technique.

As we can observe from this scenario in Figure 1, the effectiveness of these three approaches depends on several factors. For example, if *t2* arrives later, this will lead to increase in the amount of the wasted work with kill approach. On the other hand, this alternative scenario

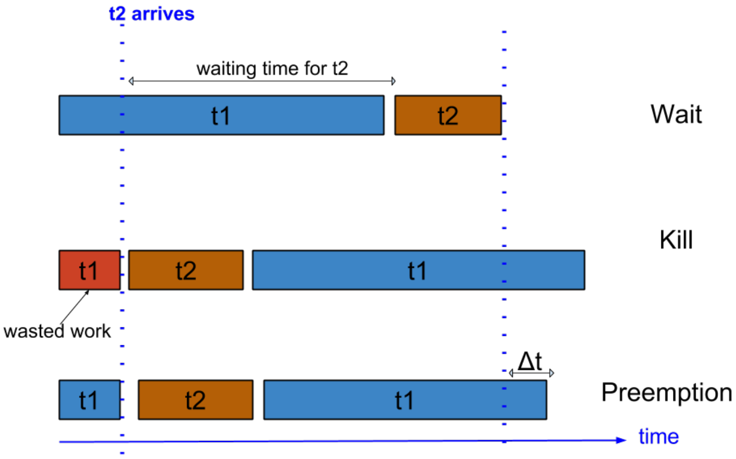


Figure 1: **Overview of the three approaches: wait, kill and preemption.**

would reduce the waiting time of *t2* with wait approach which can prevent the starvation of newly arrived jobs. For the preemption approach, its overhead (Δt) would be the main factor for its effectiveness which will depend on the implementation of the preemption technique. For instance, the overhead of the preemption technique depends on the task type in [23]. For map tasks, the overhead of the preemption is due to the waiting for the up-to-date information via new heartbeat messages in order to have a global and up-to-date view of the cluster status and it is less than 2 seconds given that the heartbeats are sent every 0.3 seconds. For reduce tasks, the overhead of the preemption is due to saving and restoring the necessary state of the task to/from the disk and also due to the waiting time during which the host (the node on which the task has been preempted) is occupied since preempted reduce tasks can only be resumed on the same node that they have been preempted.

4 Methodology

In this section, we describe the adoption of the three aforementioned approaches in Hadoop and the experimental environment: the platform and deployment setup.

4.1 Adoption of wait, kill and preemption approaches in Hadoop

We use the stable Hadoop version 1.2.1 in our experiments and leverage Hadoop Fair Scheduler (HFS) in assessing the impact of different approaches. HFS normally adopts wait approach as a strategy while enforcing the fairness and also provides the kill approach which is disabled by default. We further extended HFS to support the preemption approach. When HFS needs to deallocate resources from the jobs with over-share, preemption approach employs a pause and resume approach. In brief, it stores the necessary data on the local storage for preserving the state of the selected task for preemption with pause and restores back this information upon resume. It is important to note that, although we adopted these different approaches in Hadoop version 1.2.1, the same adoption can also be applied to the next generation of Hadoop, YARN [20]. Major difference of YARN from the Hadoop 1.x versions is that it separates the JobTracker into ResourceManager and ApplicationManager. The main motivation behind this new design is to provide better fault tolerance and scalability. On the other hand, YARN adopts a similar approach for ensuring fairness as Hadoop 1.2.x versions: it is equipped with the Fair scheduling policy as Hadoop 1.2.x and employs the same approaches (i.e., wait and kill) for ensuring fairness in shared Hadoop clusters.

4.2 Platform

The experiments were carried out on the Grid'5000 [5] testbed, more specifically we employed nodes belonging to Rennes site. On Rennes site, the nodes are configured with two 12-core AMD 1.7 GHz CPUs and 48 GB of RAM. Intra-cluster communication is done through a 1 Gbps Ethernet network. Grid'5000 allows us to create an isolated environment in order to have full control over the experiments.

4.3 Experimental deployment

We configured and deployed a Hadoop cluster using *19 nodes*. The Hadoop instance consists of the NameNode and the JobTracker, both deployed on a dedicated machine, leaving *18 nodes* to serve as both DataNodes and TaskTrackers. The TaskTrackers were configured with *8 slots* for running map tasks and *4 slots* for executing reduce tasks. At the level of HDFS, we used a chunk size of *256 MB* due to the large memory size in our testbed. We set a replication factor of *2* for the input and output data. We used the wordcount benchmark as a testing workload. Wordcount is a map-heavy workload with a light reduce phase, which accounts for about *70%* of the jobs in Facebook clusters [7].

5 Experimental Results

In this section, we provide a high-level analysis of the experimental results we obtained. First, we present comprehensive evaluation of the impact of different approaches on the performance of MapReduce applications and resource utilization. We then show the impact of workload composition and resource availability on the effectiveness of the three different approaches.

5.1 Impact of different approaches on MapReduce performance and fairness

We investigate the impact of three different approaches (i.e., wait, kill and preemption) on MapReduce performance and their efficiency in ensuring fairness. We report the job execution times and cluster share for jobs in terms of map/reduce slots for comparing the performance and fairness efficiency of these approaches. First, we compare wait and preemption approaches and then compare kill and preemption approaches.

Wait vs preemption. To evaluate the effectiveness of wait and preemption approaches in achieving fair execution between different jobs, we set up two pools with fair sharing as their internal policy. Then, we submitted a sequence of wordcount jobs with two different users (i.e., different user for each pool). While the first half of the jobs (i.e., jobs in the range of 1-5) belongs to pool 1, the second half belongs to pool 2. Notably, we submitted the jobs with random inter-arrival times (Table 1) by only enforcing the jobs in pool 1 are submitted earlier compared to the jobs in pool 2. The reason behind this condition is to be able to measure how quickly resources are given to new jobs (i.e., jobs in the range of 6-10). However, we also have a real-life scenario in Section 5.2. Here, we compare these approaches in terms of application performance.

	Job Number	Number of map tasks	Submission time
Pool 1	1	190	0
	2	190	10
	3	190	380
	4	190	870
	5	190	880
Pool 2	6	34	70
	7	34	80
	8	8	470
	9	16	1000
	10	16	1003

Table 1: **Job input sizes and submission schedule**: We submitted the jobs with random inter-arrival times by only enforcing the jobs in pool 1 are submitted earlier compared to the jobs in pool 2. The reason behind this condition is to be able to measure how quickly resources are given to new jobs. Note that, all the jobs have 72 reducers.

Figure 2 displays the execution times for each job and also its breakdown to map and reduce phases. We can see that preemption approach achieved over two-folds performance gain for the majority of the jobs in pool 2. This clearly shows that new jobs can reach their fair share quicker with the preemption approach because they do not have to wait for earlier jobs to finish. On the other hand, we observe long running times for the reducers in pool 2 with the wait approach which resulted in a significant performance degradation for these jobs. This stems from long waiting times for the resources (reduce slots) which are already occupied by the jobs in pool 1. For the jobs in pool 1, we observe a performance loss with preemption approach, especially for the reduce phase. This is due to the fact that the (reduce) tasks belonging to the jobs with over-share in the first pool are preempted to allocate resources for new jobs in pool 2 with preemption approach. However, performance loss is much smaller compared to the performance

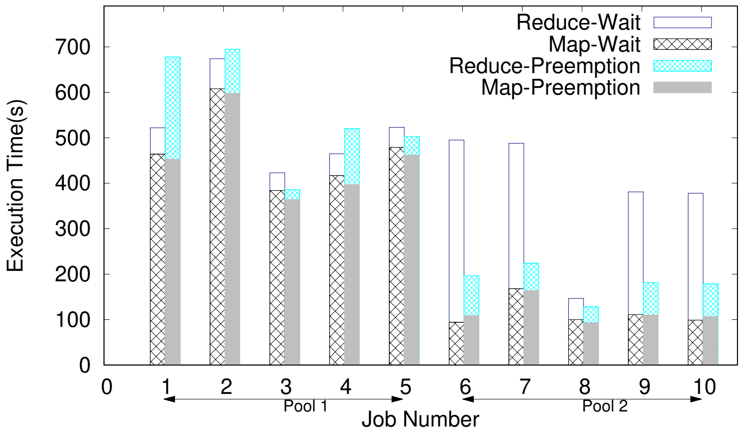
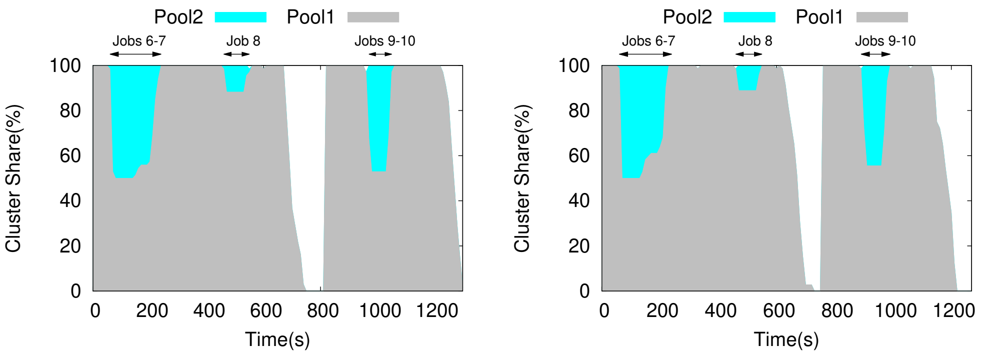


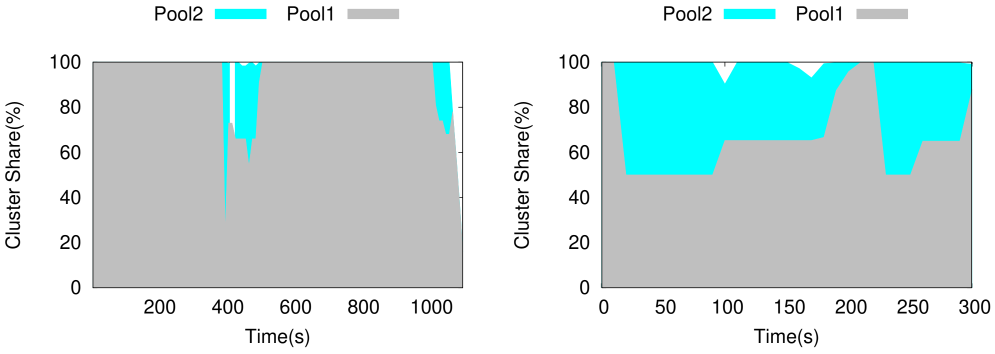
Figure 2: **Execution times of jobs under wait and preemption approaches and its breakdown to map and reduce phases.** Jobs in pool 2 experience longer reduce phases with wait approach compared to preemption approach.



(a) Wait

(b) Preemption

Figure 3: **Map slots usage**



(a) Wait

(b) Preemption

Figure 4: **Reduce slots usage.** Execution time of the reducers in the second pool reduced to 300 seconds with preemption approach while it was 1000 seconds with wait approach.

gain it achieved. In overall, preemption approach achieved 10% performance improvement for the total execution time.

To further investigate the performance difference in the second pool, we analyzed the map and reduce slots usage by the jobs in both pools with wait and preemption approaches. For the slot usage information, we show the cluster share of both pools in Figure 3 and 4. While Figure 3 displays the map slots usage, Figure 4 shows the reduce slots usage. Both approaches were able to provide equal distribution of cluster resources to the jobs in both pools for map slots usage. One might think that job 8 is under its fair share for the map resources but this stems from the fact that its demand (i.e., number of mappers in the job) is much smaller than its fair share. On the contrary to fair map slot usage under different approaches, we observed a significant difference in fair execution of the reducers. In Figure 4(a), we can clearly see the starvation of the jobs in the second pool with wait approach. This is because earlier submitted jobs in the first pool take all available reduce slots and this starvation continues until the completion of these reducers. On the other hand, reduce tasks in the second pool with preemption approach almost achieve their fair share, as shown in Figure 4(b). We also observe that these reducers only wait for a small amount of time (i.e., fair share preemption timeout) until the resources are

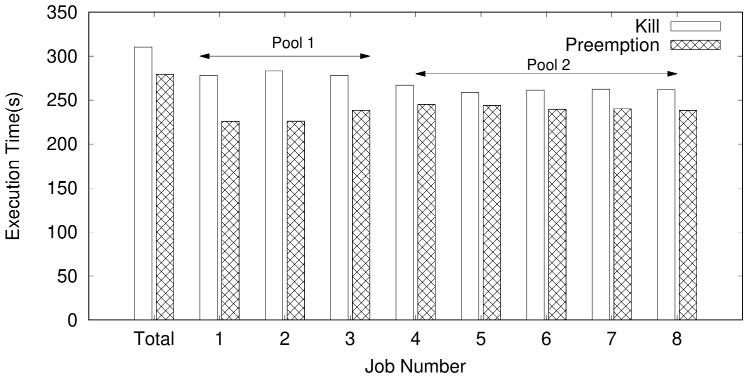


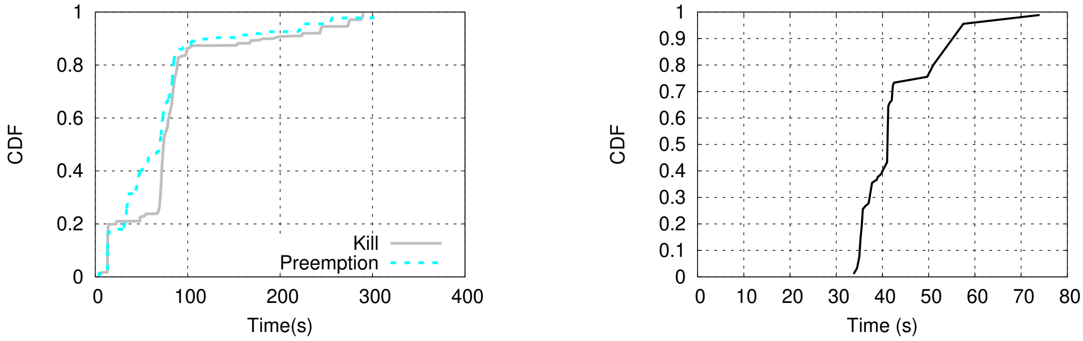
Figure 5: **Execution times of jobs under different preemption approaches**

given to them. This does not only lead to the violation of the fairness for reduce tasks with wait approach, but also contributes to a severe performance loss. We can see that while the reducers of the jobs in the second pool are completed in around 300 seconds with preemption approach, this was around 1000 seconds with wait approach. This stems from the starvation of the reducers in the second pool due to long running reducers in the first pool.

Summary. Our first experiments have yielded interesting results. First, we observe that wait as the default approach of HFS may not be efficient for ensuring fairness. This showed us the importance of an early action (i.e., preemption) for fair execution of different jobs when the cluster is fully utilized. Second, preemption does not only ensure fairness but this also leads to an increase in the performance of MapReduce applications. Specifically, we noticed that the starvation of fresh reducers (reducers which belong to the later submitted jobs) is the biggest contributor to the performance loss with wait approach.

Impact of the adopted preemptive approach. In order to compare preemption and kill approaches, we employed a similar job submission schedule as in the first set of experiments by submitting a sequence of wordcount jobs (8 jobs) with random inter-arrival times and again enforcing the later submission of the jobs from the second user to have the new jobs in pool 2. This time we reduced the number of jobs in pool 1 from 5 to 3 to have new jobs (jobs in pool 2) as a majority in the cluster which allows us to highlight the impact of the preemption technique on performance and resource utilization. The input data sizes of jobs in pool 1 and pool 2 is 17 GB and 4 GB (34 and 8 map tasks), respectively. Note that, we employed smaller input sizes since we are comparing preemptive approaches in which the running time of the jobs is not a major concern as in wait approach. Figure 5 shows the total execution time and individual execution times for each job. The results show that these two approaches have similar execution times unlike the big differences in the experiments with the wait approach. This is due to the fact that both approaches take an early action for enforcing fairness between different jobs while wait approach waits until completion of the running tasks before allocating resources for new jobs. Nevertheless, preemption outperforms kill approach for the total execution time as well as the individual execution time of each job. The reason behind this is waste-free behavior of the preemption approach while preempting the tasks. In our experiment, we find out that the number of killed/preempted tasks is equal to 79 and 68 for kill and preemption approaches, respectively.

To further investigate the different behaviors of these approaches, we plot CDFs of the successful tasks for the two approaches and the running times of killed tasks with kill approach in



(a) CDFs of running times of successful tasks for kill and preemption approaches

(b) CDF of running times of killed tasks for the kill approach

Figure 6: **CDFs of running times for killed and successful tasks**

Figure 6. In Figure 6(a), successful task represents the task which is completed without getting killed or preempted and here we see that two approaches have a similar trend. However, we also observe that there are more shorter successful tasks with the preemption approach. This is due to the shorter task durations of the preempted tasks after their resume in available slots. To note that, map preemption technique in our work creates a new subtask for the uncompleted key-value pairs at the time of preemption and this leads to a shorter life-time for these subtasks. On the contrary, kill approach destroys the on-going work of the killed tasks and these tasks start their execution from scratch when there is an available slot. Moreover, Figure 6(b) illustrates the wasteful nature of the kill approach by having killed tasks with more than 70s running time that contributes to the 23% of the total execution time.

What's more, we calculated the resource loss (in terms of time) with kill approach as following:

$$\frac{\sum t_k}{\sum t_k + \sum t_s} \tag{1}$$

where t_k is the killed task duration and t_s is the successful task duration. The percentage of the duration of killed tasks compared to the total time spent yielded 11% resource loss due to kill approach.

Summary. We observed that the cost of the preemption approach plays an important role on both application performance and resource utilization. Interestingly, the cost of the preemption approach did not only affect the preempted tasks but also affected the performance of other jobs in the cluster.

5.2 Real-life workloads

To investigate the different characteristics of the three approaches on the real-life workloads, we ran a benchmark suite based on the workload at Facebook production clusters. For the job submission schedule, we benefited from the historical Hadoop traces at Facebook [3]. We took the first 50 job submission times and ran this schedule with the wordcount benchmark. The job inter-arrival times have a mean of 56 seconds with the total submission schedule of 48 minutes.

To generate the job input sizes, we took the job input sizes (in terms of number of map tasks) at Facebook production clusters as a base. Then, we sort the jobs according to their sizes and divided them into three groups of increasing sizes. The workload composition is shown in

Group	Number of map tasks	% Jobs at workload
1	1-10	70%
2	10-100	15%
3	> 100	15%

Table 2: The workload composition based on the job sizes at Facebook traces

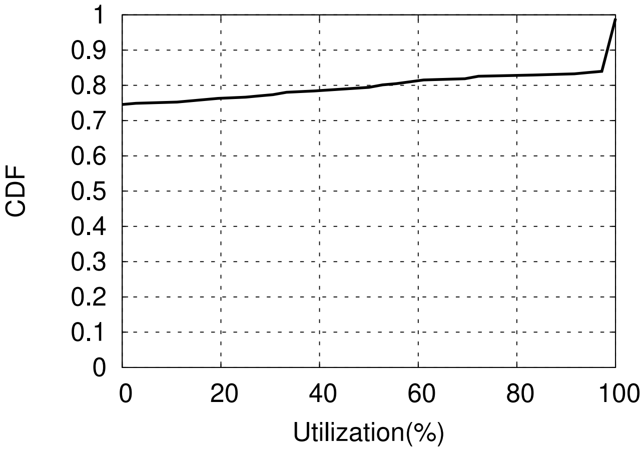
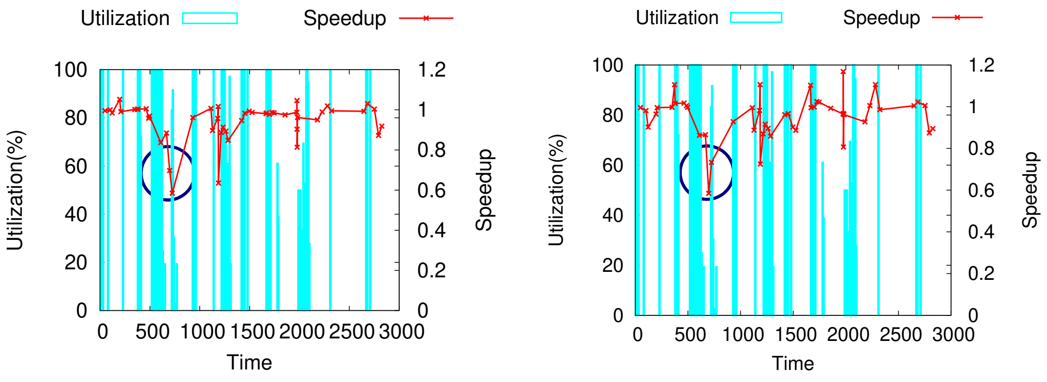


Figure 7: **CDF of cluster utilization under the workload based on the Facebook production clusters**

Table 2. It is important to note that most jobs at Facebook are small. We also note that Facebook clusters are busy only for a short time [24]; Figure 7 shows the CDF of cluster utilization under this workload composition . Finally, we ran this workload with two different users (i.e., the jobs are distributed to two pools) under three different approaches: wait, kill and preemption. We kept the same experimental deployment as described in Section 4.3 by only adjusting reducer slots to 2 for each node due to small size of the jobs in the workload. On the contrary to the results in Section 5.1, wait approach does not lead to a noticeable fairness violation and outperforms other approaches under this workload composition. Interestingly, preemptive actions affected the overall performance negatively by resulting in 8% - 10% performance degradation. We observe that job average execution time is 65.3 seconds with wait approach while this is 70.7 and 71.7 seconds with kill and preemption approaches respectively.

Performance degradation in kill and preemption approaches. To further investigate the noticeable performance degradation with the preemptive approaches, we analyzed the resource availability and its relation with the individual job execution times. To do so, we plot the cluster utilization and relative speedups (wait approach as a baseline) of the submitted jobs with respect to their submission times for kill and preemption approaches in Figure 8. Notably, we display the cluster utilization in terms of reduce slots since we only observed kill/preemption attempts for the reducers with kill and preemption approaches. We discover that kill and preemption approaches lead to performance degradation in this scenario due to their design choices. For kill approach, it destroys the on-going work of the preempted tasks, thus these tasks need to be re-executed from scratch which results in a performance degradation. On the other hand, preemption approach results in a performance degradation by introducing a delay for the preempted reduce tasks since these tasks can only be resumed on the same node that they have been preempted. In particular,



(a) Preemption

(b) Kill

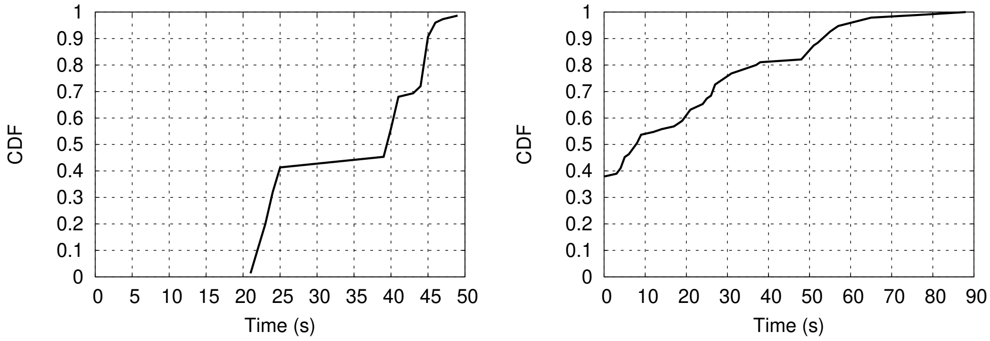
Figure 8: **Interplay between cluster utilization and job execution times with respect to two preemptive approaches: preemption and kill**

we can observe this performance loss with the jobs which are submitted after the peak time of the cluster (i.e., when the cluster resources are fully utilized, as shown in circles in Figure 8) for both approaches.

Kill vs preemption. As shown in Figure 8(b), kill approach has a greater performance variability compared to preemption approach; namely, it improved the performance of some jobs better compared to the preemption approach but this also led to more serious slowdowns for other jobs in the cluster. This stems from the fact that kill is a simple preemptive action and can take a faster decision for deallocating the resources from the tasks belonging to the jobs with over-share. On the contrary, preemption approach waits for the up-to-date information via new heartbeat messages in order to have a global and up-to-date view of the cluster status and this takes up to 2 seconds. Hence, this results in a more aggressive resource allocation with kill approach. In our experiment, we find out that the number of killed/preempted tasks is equal to 74 and 59 for kill and preemption approaches, respectively.

What's more, we can see the wasteful nature of kill approach in Figure 9(a) which displays the CDF of running times of the killed tasks. For preemption approach, as we indicated earlier that preempted reduce tasks can be resumed only on the same node where they have been preempted. Hence, this introduces a delay for re-launching the preempted tasks when there are no free reduce slots available on the same node as they have been preempted. We can see these delay times of the preempted tasks, the major cause of the performance degradation, in Figure 9(b). On the other hand, with wait approach newer tasks can obtain the reduce slots after waiting for a small amount of time since cluster is fully utilized only for a short time. Therefore, it does not suffer from long running reduce tasks as in the previous set of experiments. This shows us the importance of the waiting time besides the resource availability which depends on the running time of the tasks.

Summary. Our second set of experiments showed us the importance of the workload composition. While wait suffered from the previous workload type and resulted in a serious fairness violation and performance degradation, it outperformed other approaches under this workload composition. Importantly, we discovered a correlation between the resource availability and the effectiveness of these approaches. We observed that kill and preemption approaches lead to performance degradation if resources are not available only for a short time. This performance degradation is mainly contributed by the re-execution of the killed tasks for kill approach and



(a) CDF of running times of the killed tasks for the kill approach

(b) CDF of waiting times of the preempted tasks for the preemption approach

Figure 9: **CDFs of running times for killed and waiting times of preempted tasks**

the introduced delay for the preempted tasks for preemption approach. After a deep analysis of the results with both workload types, we can say that none of these approaches is superior in all situations. Therefore, an adaptive model that can select the optimal approach for ensuring fairness can be necessary; we will investigate its feasibility in our future work.

6 Discussion and Future Work

Hadoop is widely used to process large volumes of data and Hadoop clusters are usually shared among multiple users and jobs in order to maximize the resource utilization. In these multi-tenant environments, achieving fairness is a major challenge. To meet this challenge, Hadoop is equipped with its fair scheduler, Hadoop Fair Scheduler (HFS). HFS provides two simple approaches (wait and kill) to allocate resources for tasks belonging to the jobs with less share than their fair share. In this study, we also adopted the preemption approach which was originally developed in [23] on HFS and gave a comprehensive analysis on the efficiency of these three approaches while enforcing fairness. Our findings demonstrate that these approaches are sub-optimal for different job sizes and workload compositions. We confirm that wait as the default approach of HFS may not be efficient in case of long running tasks [24, 21]. In addition, we show that preemptive approaches (kill and preemption) not only help to achieve fairness but also improve the MapReduce performance. Surprisingly, these preemptive approaches did not outperform wait when cluster utilization is low and cluster consists of mostly short jobs. Hence, we claim that adaptive model that can select the optimal approach depending on the job sizes and characteristics, resource availability and workload composition is necessary. For instance, wait approach may be more efficient when the waiting time (the time until the resources will be freed for the tasks belonging to the jobs with less-share) is short. However, estimating this waiting time a priori to the job execution is not trivial since this waiting time varies according to (1) the running applications (i.e., by analyzing the traces collected from three different research clusters [2], we observe that the execution time of map and reduce tasks varies from 2 to 84631 seconds and from 9 to 81714 seconds, respectively); and (2) the progress of the application. Hence, adaptive model which can take decisions on-the-fly can improve MapReduce performance while ensuring fairness.

We also find out that the cost of the preemption approach plays an important role on both application performance and resource utilization. While kill approach is costly due to its wasteful nature by destroying the on-going work of the killed tasks, preemption approach has an overhead while preempting the reduce tasks. Moreover, we show that the implementation choices of the preemption approach determine its overhead. The preemption approach we adopted is limited for being able to resume the preempted reduce tasks only on the same node that they have been preempted. In order to achieve a better preemption technique, this limitation can be mitigated by replicating the necessary data of the preempted reduce tasks to other nodes in the cluster. However, this may lead to the unexpected overhead due to the data transfer in between nodes. Several avenues remain open for future work. One is to expand our experimental study by using other types of widely used fair scheduling policies (e.g., Dominant Resource Fairness [10]). As the ultimate goal, by leveraging the knowledge gained in our work, we plan to investigate a fairness model based on these three different approaches (i.e., wait, kill and preemption), workload composition and the resource availability and use this model to adaptively select an optimal approach for ensuring fairness.

7 Acknowledgments

This work is supported by the ANR KerStream project (ANR-16-CE25-0014-01). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations (see http://www.grid5000.fr/). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357.

References

[1] Hadoop Fair Scheduler. http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

[2] Hadoop Workload Analysis. http://www.pdl.cmu.edu/HLA/index.shtml

[3] SWIM Workload Repository. https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

[4] The Apache Hadoop Project. http://www.hadoop.org

[5] Grid'5000. https://www.grid5000.fr (2020)

[6] Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar, T.: Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). pp. 1–13 (2014)

[7] Chen, Y., Alspaugh, S., Katz, R.: Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. Proceedings of the VLDB Endowment 5(12), 1802–1813 (2012)

[8] Cheriere, N., Donat-Bouillud, P., Ibrahim, S., Simonin, M.: On the usability of shortest remaining time first policy in shared hadoop clusters. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16). p. 426–431 (2016)

[9] Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)

[10] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI 2011. vol. 11, pp. 24–24 (2011)

[11] Ghodsi, A., Zaharia, M., Shenker, S., Stoica, I.: Choosy: Max-min fair sharing for datacenter jobs with constraints. In: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013). pp. 365–378 (2013)

[12] Huang, D., Shi, X., Ibrahim, S., Lu, L., Liu, H., Wu, S., Jin, H.: Mr-scope: a real-time tracing tool for mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 849–855. Chicago, Illinois (2010)

[13] Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., Wu, S.: Maestro: Replica-aware map scheduling for mapreduce. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012). pp. 435–442 (2012)

[14] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP 2019). pp. 261–276 (2009)

[15] Jin, H., Ibrahim, S., Qi, L., Cao, H., Wu, S., Shi, X.: The mapreduce programming model and implementations. Cloud Computing: Principles and Paradigms pp. 373–390 (2011)

[16] Liu, H., He, B.: Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 14). pp. 970–981 (2014)

[17] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., Stoica, I.: Faircloud: sharing the network in cloud computing. In: Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication. pp. 187–198 (2012)

[18] Radunovic, B., Le Boudec, J.Y.: A unified framework for max-min and min-max fairness with applications. IEEE/ACM Transactions on networking 15(5), 1073–1083 (2007)

[19] Tang, S., Lee, B.s., He, B., Liu, H.: Long-term resource fairness: towards economic fairness on pay-as-you-use computing systems. In: Proceedings of the 28th ACM international conference on Supercomputing (ICS 14). pp. 251–260 (2014)

[20] Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing (SoCC 13. p. 5 (2013)

[21] Wang, Y., Tan, J., Yu, W., Meng, X., Zhang, L.: Preemptive reducetask scheduling for fair and fast job completion. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013)

[22] Yildiz, O., Ibrahim, S., Antoniu, G.: Enabling fast failure recovery in shared hadoop clusters: Towards failure-aware scheduling. Future Generation Computer Systems (2016)

[23] Yildiz, O., Ibrahim, S., Phuong, T.A., Antoniu, G.: Chronos: Failure-aware scheduling in shared hadoop clusters. In: 2015 IEEE International Conference on Big Data (Big Data 2015),. pp. 313–318 (2015)

[24] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys'10). pp. 265–278. Paris, France (2010)

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399