# An Eye on the Elephant in the Wild: A Performance Evaluation of Hadoop's Schedulers Under Failures

Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu

▶ **To cite this version:**

Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu. An Eye on the Elephant in the Wild: A Performance Evaluation of Hadoop's Schedulers Under Failures. ARMS-CC'15-The second workshop on Adaptive Resource Management and Scheduling for Cloud Computing, held in conjunction with PODC 2015,, Jul 2015, Donostia-San Sebastián, Spain. hal-01184236

**HAL Id: hal-01184236**

**https://inria.hal.science/hal-01184236**

Submitted on 13 Aug 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# An Eye on the Elephant in the Wild: A Performance Evaluation of Hadoop's Schedulers Under Failures

Shadi Ibrahim[***], Tran Anh Phuong[†], and Gabriel Antoniu

Inria Rennes - Bretagne Atlantique Research Center,
Rennes, France

**Abstract.** Large-scale data analysis has increasingly come to rely on MapReduce and its open-source implementation Hadoop. Recently, Hadoop has not only been used for running single batch jobs but it has also been optimized to simultaneously support the execution of multiple jobs belonging to multiple concurrent users. Several schedulers (i.e., Fifo, Fair, and Capacity schedulers) have been proposed to optimize locality executions of tasks but do not consider failures, although, evidence in the literature shows that faults do occur and can probably result in performance problems.
In this paper, we have designed a set of experiments to evaluate the performance of Hadoop under failure when applying several schedulers (i.e., explore the conflict between job scheduling, exposing locality executions, and failures). Our results reveal several drawbacks of current Hadoop's mechanism in prioritizing failed tasks. By trying to launch failed tasks as soon as possible regardless of locality, it significantly increases the execution time of jobs with failed tasks, due to two reasons: 1) available resources might not be freed up as quickly as expected and 2) failed tasks might be re-executed on machines with no data on it, introducing extra cost for data transferring through network, which is normally the most scarce resource in today's data-centers. Our preliminary study with Hadoop not only helps us to understand the interplay between fault-tolerance and job scheduling, but also offers useful insights into optimizing the current schedulers to be more efficient in case of failures.

**Keywords:** MapReduce; Hadoop; cloud computing; failure; schedulers

## 1 Introduction

Data insight forms an essential part in today's decision making process. With the massive growth in available data, companies are spending millions of dollars on business intelligence and big data analytics [21]. Companies become data-driven, shifting the business policies from the traditional instinct-based decision

[***] **Contact author:** Shadi Ibrahim; Inria, Campus Universitaire de Beaulieu, 35042 Rennes, France; Phone: +33(0)299842534; email: shadi.ibrahim@inria.fr.

[†] This work was done while Tran Anh Phuong was an intern at Inria Rennes.

to analyzing available data. Internet-centric enterprisers are among the most active players in the big data analytic field. Yahoo! uses its large datasets to support research for the advertisement system. In 2008, Yahoo! reported that 24 billions events are processed per day in the effort of analyzing Web visitors' behavior [4]. Other institutes have also reported to process datasets in the order of terabytes or even petabytes [25][22].

The practice of analyzing huge amounts of data motivated the development of data intensive processing tools. In this context, Hadoop [2], an open-source implementation of Google MapReduce framework [9], is emerging as a prominent tool for big data processing and it is now widely adopted by both industry and academia [19][18]. For example, Facebook claimed to have the world's largest Hadoop cluster [7] with more than 2000 machines and running up to 25000 MapReduce jobs per day. Well-known cloud providers such as Amazon, Google and Microsoft respond to the need of data processing by equipping their software stack with a MapReduce-like system such as Hadoop in Amazon. Amazon Elastic MapReduce [1] is an example for platforms that facilitate large-scale data applications. Many successful case studies have demonstrated the simplicity, convenience and elasticity of MapReduce-cloud (i.e., MapReduce-based service in public cloud). For example, the New York Times rented 100 virtual machines for a day and used Hadoop to convert 11 million scanned articles to PDFs [13].

Hadoop has been recently used to support the execution of multiple jobs belonging to multiple concurrent users. Therefore, several schedulers (i.e., Fifo, Fair, and Capacity schedulers) were proposed to allow administrators to share their cluster resources among users with a certain level of fairness and/or performance guarantee. These schedulers adopt a resource management model based on *slots* to represent the capacity of a cluster: each worker in a Hadoop cluster is configured to use a fixed number of map slots and reduce slots in which it can run tasks.

In response to reliability, evidence shows that failures do happen in clouds [12]. In fact, researchers interested in fault tolerance have accepted that failure is becoming a norm, rather than an exception. For instance, Dean reported that in the first year of a cluster at Google there were thousand individual failures and thousand of hard drive failures [8]. Despite this prevalence of failures, with the absence of clear definition of Quality of Service (i.e., QoS) in the cloud (e.g., Amazon offers 99.99% uptime), the expenses of failures entirely rest on users, regardless of the root causes. With respect to MapReduce on clouds, cloud providers rely completely on the fault-tolerance mechanisms which are provided by Hadoop system. This policy entitles users the freedom to tune these fault tolerance mechanisms, but leaving also the consequences and the expenses on the users side.

### Our contribution

Hadoop gracefully handles failures by simply re-executing all the failed tasks. However, all these efforts to handle failures are entirely entrusted to the core of

Hadoop and hidden from the job schedulers. This potentially leads to degradation in Hadoop's performance.

This paper aims at exploring the problem of failures in a shared Hadoop cluster. In particular, the paper presents a systematic performance evaluation of the three built-in schedulers (i.e., the Fifo scheduler, the Fair scheduler and the Capacity scheduler) under failure. Accordingly, we conduct a series of experiments to assess the impact of stress (a mild variance of failure) and failures in the execution of multiple applications in Hadoop. Our results reveal several drawbacks of current Hadoop's mechanism in prioritizing failed tasks. By trying to launch failed tasks as soon as possible regardless of locality, it significantly increases the execution time of jobs with failed tasks, due to two reasons: 1) available slots might not be freed up as quickly as expected and 2) the slots might belong to machines with no data on it, introducing extra cost for data transferring through network, which is normally the most scarce resource in today's data-centers [23].

**Paper Organization**

The rest of this paper is organized as follows: section 2 provides background into the Hadoop system, a glimpse on the fault-tolerance mechanism of Hadoop, as well as the three built-in schedulers. Section 3 provides an overview of our methodologies, followed by the experimental results in Sections 4 and 5. Section 6 discusses open-issues and new possibilities to improve Hadoop performance under failure. Finally, we conclude the paper and propose our future work in Section 7.

## 2 Background

We provide a brief background on Hadoop, its fault-tolerance mechanisms and scheduling in Hadoop clusters.

### 2.1 Hadoop

Hadoop, an open-source implementation of MapReduce [9], is used to process massive amounts of data on clusters. Users submit jobs as consisting of two functions: map and reduce. These jobs are further divided into tasks which is the unit of computation in Hadoop. Input and output of these jobs are stored in a distributed file system. Each input block is assigned to one map task and composed of key-value pairs. In the map phase, map tasks read the input blocks and generate the intermediate results by applying the user defined map function. These intermediate results are stored on compute node where the map task is executed. In the reduce phase, each reduce task fetches these intermediate results for the key-set assigned to it and produces the final output by aggregating the values which have the same key.

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. Then, JobTracker will assign map/reduce tasks depending on the capacity of the TaskTracker and also by considering the locality of the map tasks (i.e., among the TaskTrackers with free slots, the one with the data on it will be chosen for the map task).

### 2.2 Fault-tolerance in Hadoop

Hadoop employs a static timeout mechanism for the detection of fail-stop failures. It keeps track of each TaskTracker's last heartbeat, so that if a TaskTracker has not sent any heartbeat in a certain amount of time, that TaskTracker will be declared failed. Each TaskTracker sends a heartbeat every 0.3s (many literatures have claimed that the heartbeat interval is 3 seconds [11], however here we use the value we found in the source code). The JobTracker checks every 200s for any TaskTracker that has been silent for 600s. Once found, the TaskTracker is labeled as failed, and the JobTracker will trigger the failure handling and recovery process. Tasks that were running on the failed TaskTracker will be restarted on other machines. Map tasks that completed on this TaskTracker will also be restarted if they belong to jobs that are still in progress and contain some reduce tasks. Completed reduce tasks are not restarted, as the output is stored persistently on HDFS. Hadoop's failed tasks are either added to a queue for failed tasks (map task), or back to non-running queue (reduce task). Both queues are sorted in the order of failed attempts: tasks with higher times of failures are positioned at the beginning of the queues. In case tasks have the same number of failed retries (each task has five opportunities to run and therefore can fail at most four times[14]), the task ID is used to tie break.

### 2.3 Multiple job scheduling in Hadoop

Hadoop runs several maps and reducers concurrently on each TaskTracker to overlap computation, I/O, and communication. At each heartbeat, TaskTracker notifies JobTracker on the number of available slots it currently has. JobTracker assigns tasks depending on jobs' priority, number of non-running tasks and potentially other criteria. The first version of Hadoop comes with a fixed Fifo scheduler, which was good for traditional usages such as Web Indexing or log mining, but rather inflexible and could not be tailored to different needs or different workload types.

Since the bug report Hadoop-3412[1], Hadoop has been modified to accept pluggable schedulers, which allows the use of new scheduling algorithm to help

[1] `https://issues.apache.org/jira/browse/HADOOP-3412`

optimizing jobs with different specific characteristics. At the current stable version, Apache Hadoop is augmented with three readily available schedulers, namely the default Fifo scheduler, the Fair scheduler and the Capacity scheduler.

**Fifo scheduler.** Fifo scheduler is the original scheduler that was integrated inside the JobTracker. In Fifo scheduling, the JobTracker simply pulls jobs from a single job queue. Although the scheduler's name suggests the prioritization of old jobs, Fifo scheduler also takes into account jobs' priority.

**Fair scheduler.** Fair scheduler uses a two-level scheduling hierarchy. At the top level, cluster slots are allocated across *pools* using weighted fair sharing i.e. the higher the weight a pool is configured with, the more resources it can acquire. Each user, by default, is assigned one pool. At the second level and within each pool, slots are divided among jobs, using either Fifo with priorities (the same with Fifo scheduler) or a second level of fair sharing. In the second level of fair scheduling, each job is assigned a weight equal to the product of its (user-defined) priority and the number of tasks. Jobs with higher number of tasks generally need more time to finish, and will be assigned more task slots. Note that Fair scheduler associates the number of tasks with the length of job, which means it assumes tasks have the same length. This assumption is not necessary true since the length of a task differs between applications: even with the same amount of data, a complicated map (reduce) function will probably take more time to finish than a simple map (reduce) function.

Fair scheduler uses Kill action to guarantee that pools meet their *minimum share*. The minimum share is defined as the minimum number of slots that a pool is given at run-time. The minimum share is automatically adjusted if the total minimum share of all pools exceeds the number of slots in the cluster.

Fair scheduler gracefully embraces short jobs, to which locality is of high importance. Fair scheduler is therefore equipped with the Delay technique, which allows the execution of a task on a TaskTracker to be postponed if the scheduler can find a local task for that TaskTracker. Postponed tasks are recorded so that if a task has waited for too long, it will be launched at the next free slot regardless of locality. This is to avoid starvation for tasks in a big cluster, where the chance for a task to have local data on a certain node is rather low.

**Capacity scheduler.** Capacity scheduler aims to facilitate sharing resources in multi-tenant Hadoop cluster. The central idea is that the resources are partitioned among tenants based on computing needs. Unused quotas are divided equally among using tenants.

Although the idea is rather similar to Fair scheduler, Capacity scheduler has some of its own characteristics. JobTracker is considered as a rather scarce resource, therefore the number of initialized jobs are limited i.e. not all jobs are always initialized upon submission. Jobs are divided into queues and accessed sequentially in a manner similar to Fifo. Once a job started, it will not be

preempted by other jobs. Preemption is an interesting functionality, but not yet been implemented.

## 3 Experiment settings

### 3.1 Cluster setup

We conduct our experiments on Grid'5000 [3]. Grid'5000 is a large-scale experiment testbed, which provides the research community with a highly configurable infrastructure. We perform our experiment on Rennes site with 21 nodes: 1 master node running the JobTracker and NameNode processes, and 20 slave nodes running TaskTracker and DataNode processes (all the other experiments, unless stated otherwise, will always have 1 dedicated node as the master, and many other slave nodes). Each node is equipped with 2.5Ghz Intel Xeon L5420 CPU with 8 cores, 16GB Memory and one 320GB SATA II hard drive. Nodes are connected using Gigabit Ethernet cable.

### 3.2 Hadoop setup

On the Grid'5000 testbed described above, we deployed and configured a Hadoop cluster using the 1.2.1 stable version. Each node is configured with 6 map slots and 2 reduce slots (1 slot per core on average). The number of reduce tasks is set at 40 tasks. The HDFS replication is set at 2, and the block size is set at 128MB (default block size in Hadoop). To better cope with small workload, we set the expiry time (the amount of time a JobTracker will wait before declaring a TaskTracker failed if there was no heartbeat from that TaskTracker) to 60 seconds instead of the default 600 seconds. Besides that, all the other configurations are kept at the default value.

### 3.3 Benchmarks

Throughout our evaluation, we use primarily the WordCount and TeraSort applications from the official Hadoop release's example package (i.e., the main benchmark used to evaluating MapReduce Hadoop [9][27]). Data is extracted from PUMA data set [5] to create different input sizes.

### 3.4 Jobs

We run a series of 6 jobs (from the combinations between applications and input sizes), see Table 1. Each job is submitted 10 seconds after each other, and job types are intermixed so that we have mixed sets of long, medium and short jobs of different characteristics.

| # | Application | Input size |
|---|---|---|
| **1** | TeraSort | 31GB |
| **2** | WordCount | 11GB |
| **3** | TeraSort | 2.8GB |
| **4** | WordCount | 30GB |
| **5** | TeraSort | 12GB |
| **6** | WordCount | 1.1GB |

**Table 1.** List of jobs and their input size used in the experiment, in the order of submission

## 4 Hadoop under stress

In this experiment, we want to understand Hadoop's behavior under the condition when one of the nodes in the cluster got stressed. In a shared cluster, nodes may run many different processes at the same time for utilization reasons. It is not uncommon to have some of the nodes having more running processes than other nodes. The situation becomes more significant when virtualization is employed [17]. Since each of the virtualized machines (i.e., VMs) has to compete for resources from the physical machines, spontaneous congestions can happen frequently [15].

### 4.1 Stressing

We launch some extra processes (`while` loop for CPU stress & `dd` command for I/O stress) on one of the nodes from the slave set. These processes act as the stress factor on that node. Each process lasts for 30 seconds, and is launched interleaved with each other. Between any 2 stressing processes there is a gap of 30 seconds of stress-free to simulate sporadic stress condition. The first stress process is launched 60 seconds from the beginning of the experiment.

### 4.2 Results

**Total execution time and data locality.** Figure 1 presents the total execution time of the experiment test under 2 different scenarios. In normal situation, all the three schedulers demonstrate similar performances in term of finishing time: Fair and Capacity schedulers both finish after 249 seconds, and Fifo finishes after 246 seconds. In stressed condition, the three schedulers start to show some differences. Fifo scheduler once again finishes the first with 4 seconds of degradation. Capacity scheduler also suffers the same amount, while Fair scheduler gets prolonged for 7 seconds (finishes after 256 seconds) and becomes the slowest among the three.

Figure 2 presents the locality (the ratio between the number of locally executed tasks and total number of tasks) of the 3 schedulers under different situations. Fair scheduler demonstrates the highest locality as expected, thanks to the Delay technique. Fifo scheduler and Capacity scheduler demonstrate similar

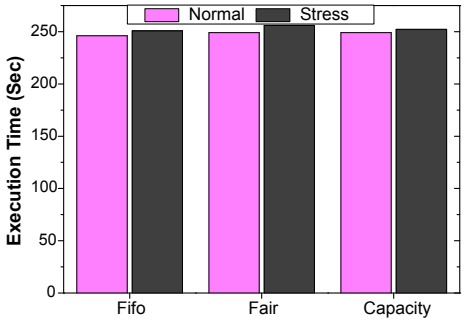


**Fig. 1.** Total execution time of the 3 different schedulers

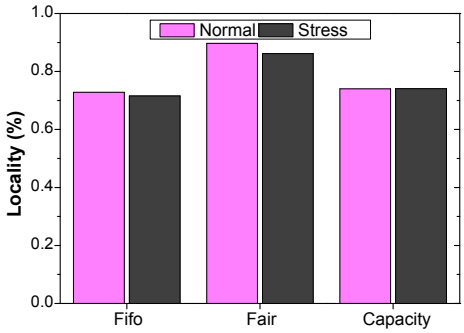


**Fig. 2.** Data locality of the 3 scheduler under different situations

performance in this aspect, and significantly lower compared to Fair scheduler. The difference was not reflected in the total execution time, as network bandwidth is rather abundant in our experiment: all the nodes are in the same rack, and besides Hadoop, there was no other users' running processes that involves network during the course of the experiment. The result of this abundance is that even a chunk of 128MB can be quickly transferred between nodes without much delay. This setting is normally not true in a multi-purpose, multi-tenant cluster: network bandwidth is generally considered as a scarce type of resource [24].

Under stressed conditions, all three schedulers witness degradation in data locality. This is because tasks on the stressed node are likely to become stragglers, and receive speculation from Hadoop. Although Hadoop also tries to provide locality for these speculative tasks, if the original copy was launched on a node with data, then the second launch will have less chance to be local.

Another effect of stressing a node is that the increase in number of speculative tasks is accompanied with more waste in the resources. Speculative tasks are launched, but they eventually are killed when the original tasks finish. We can see more about this phenomenon in figure 3 and 4.

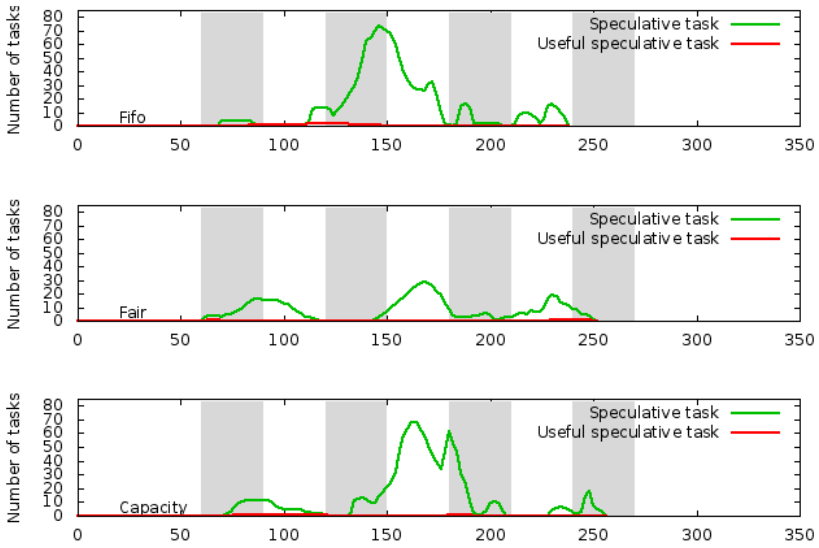


**Fig. 3.** Speculation execution in normal situation of the 3 schedulers

**Speculation execution.** Figure 3 shows the number of running speculative tasks at different points in time of the normal scenario. The green color depicts the total number of running speculative tasks, while the red color, noted as "Useful speculative tasks", illustrates those speculative tasks that actually finish before the original one, and hence, are meaningful. Note that the figure shows the number of running speculative tasks at different points in time during the course of the experiment: tasks are generally accounted for more than once. The shaded region marks the duration during which stress processes are running. Although there were no stress processes during normal scenario, we keep the shaded color for the ease of comparison with stressed scenarios. Figure 3 shows that the speculation mechanism in Hadoop is not very effective in this scenario: most of the speculative tasks are actually wasted.

Figure 3 also compares the difference in how each scheduler chooses tasks to speculate. Fifo scheduler and Capacity scheduler show a similar pattern in speculative execution: when running Capacity without concerning about share, the *default* queue in Capacity is basically a Fifo queue. However, there are still some differences in the number of concurrently running speculative tasks at a moment: Fifo scheduler has the padding mechanism to slow down the rate of assigning tasks while Capacity scheduler does not, and this difference alters the number of occupied slots in the last wave of a job. Fair scheduler has less speculative tasks compared to the other two schedulers.

Figure 4 illustrates the speculative execution of the 3 schedulers, but in the scenario of stress. Both the number of speculative tasks, as well as the number of useful tasks increase (though slightly), showing the effect of stressing processes on Hadoop. Once again, Fifo and Capacity schedulers show similar behavior,

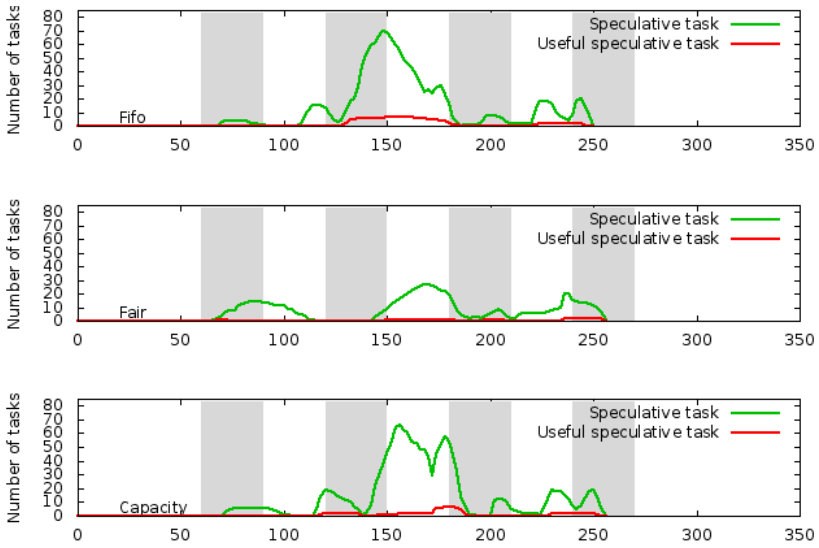


**Fig. 4.** Speculation execution in stressed situation of the 3 schedulers

while Fair scheduler still introduces less number of speculative tasks compared to the other two. The occurrence of speculative tasks is generally delayed a few seconds: this is because stressed node takes more time to finish its tasks, and therefore delays the last wave for a short period of time. We can also observe more "useful speculative tasks" (red tasks): speculative mechanism proves to be useful, though limited.

## 5 Hadoop under failure

We evaluate Hadoop's performance when there are failures in the system. To mimic the failure, we simply kill the TaskTracker process on one of the slave nodes. Failure injection time is set at 80 seconds since the beginning of the experiment. TaskTracker process is never restarted (fail-stop). DataNode process is kept running, so that no re-replicate activities occur. Data is still accessible from that node, but there will be no more tasks to be launched from the same machine.

The default expiry time (the amount of time after which a TaskTracker will be declared "lost" if there was no hearbeat) is 600 seconds. This value is considerably large compared to the job size (the largest job in this experiment only takes around 200 seconds to finish). We change this value to 60 seconds for a more timely reaction to failure.

### 5.1 Result

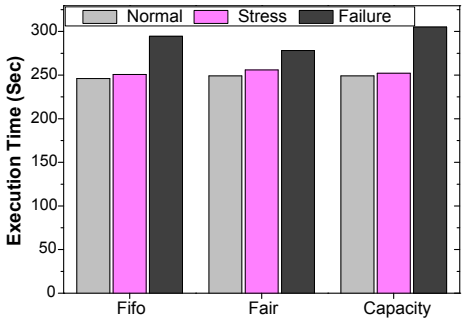


**Fig. 5.** Total execution time of Hadoop in 3 scenario: Normal, stress and failure

**Total execution time and locality.** Figure 5 presents the total execution time in 3 different scenarios: Normal, stress and failure. The stress scenario is included for better comparison. As we can see from 5, failure prolongs the execution of Hadoop jobs by a significant amount of time as much as 56 seconds (Capacity). Fair scheduler appears to suffer the least: its execution time is prolonged for only 29 seconds, and it also finishes the fastest among the three schedulers under failure (278 seconds compared to 294 seconds for Fifo, and 305 seconds for Capacity scheduler).

The small degradation of Fair scheduler can be explained by the fact that Fair scheduler allows multiple jobs to share the cluster proportional to their job sizes. Each job now has less resources at a point in time compared to that in Fifo. When a failure occurs, since jobs have been progressing slower than that in Fifo, they can overlap the meaningful effort (to finish other tasks) with the expiry time (failure detection window). Besides since the failed node only accounts for 5% of the total number of slots, there may be chances that none of the tasks on the failed node belongs to some jobs (specially reduce tasks). These jobs will not be blocked and can be finished even during the failure detection window. This helps limit the impact of a node failure on jobs under the scheduling of Fair scheduler.

Figure 6 shows the percentage of locally executed tasks over the total number of tasks in the 3 different scenarios: Normal, stress, and failure. Fair scheduler still enjoys the highest number of locality, even though this number is decreasing. Fifo and Capacity scheduler show some degradation, though this degradation is rather small compared to Fair scheduler. To explain about this phenomenon, remember that Fair scheduler was designed based on the assumption that most tasks are short and therefore, node will release slot quickly enough for other tasks to get locally executed. However in case of failure, the long failure detection time (expiry time) creates the illusion of long-lasting tasks on failed node.

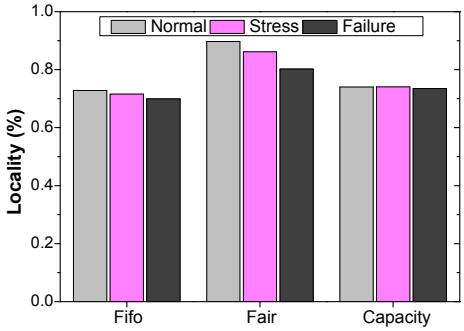


**Fig. 6.** Data locality of the 3 scheduler under normal, stress and failure scenario

These "fake" long tasks break the assumption of Fair scheduler, leading to high degradation.

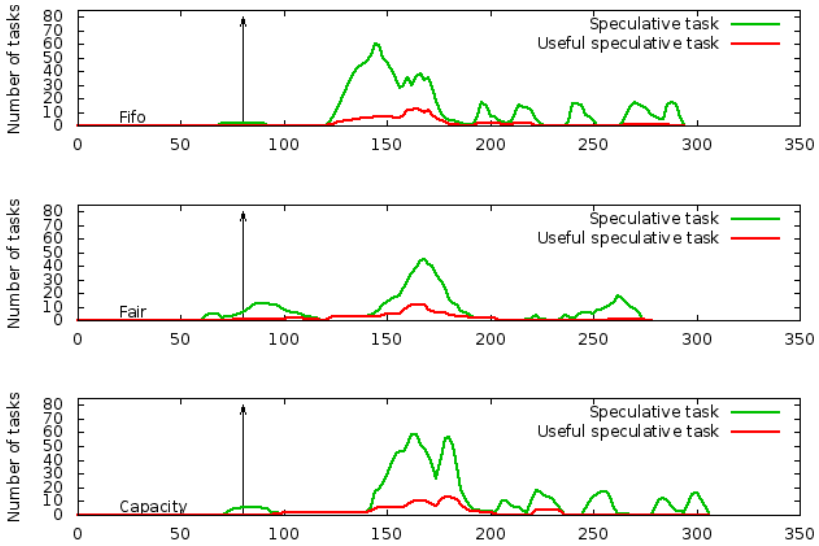


**Fig. 7.** Speculation execution in the Failure scenario of the 3 schedulers

**Speculative execution.** Figure 7 demonstrates the speculative execution of the 3 schedulers under failure. The number of "useful speculative" increased compared to normal execution (Figure 3). This is because during the 60 seconds between the failure of TaskTracker and its failure discovery, some speculative tasks were launched and finished successfully. Other than this increase in the number of successful speculative tasks, all other observations remain the same:

Fifo and Capacity schedulers show a similar pattern in speculative execution; Fair scheduler has the least number of speculative tasks among the three.

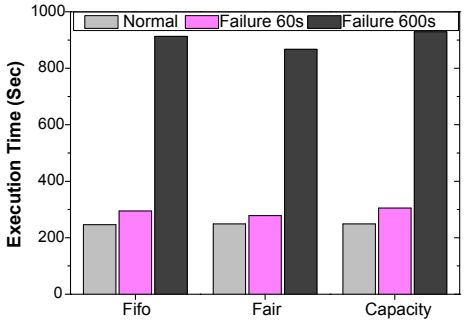


**Fig. 8.** Total execution time in 3 different situations: Normal, Failure with 60s of expiry time, Failure with 600s of expiry time

Although rather obvious, we also include a situation when the default expiry time (600 seconds) is used. The total execution time is by far longer in this default setting. This is because there are some already finished tasks on the failed node when it failed. These map outputs will either have to wait until the node is declared failed, or there are enough "Failed to fetch map output" [11] notifications in order to be re-executed. The longer the expiry time, the longer the job is blocked, and therefore total execution time becomes longer. Speculation mechanism does not improve the situation, as it can only speculate currently running tasks.

## 6 Discussion on Hadoop performance under failure

Failures significantly increase the execution time of Hadoop applications. There are two factors that contribute to this degradation: the delay in failure detection, and the failure handling mechanism that Hadoop employs.

Hadoop uses a fixed value for the expiry time regardless of the workload. The default value is 10 minutes, which is of disadvantages to small jobs [11]. Although the delay would be the same in value, the toll for larger jobs (jobs that take longer time to finish) is relatively smaller. Besides, during the period of failure detection, larger jobs might have other unfinished tasks to run, so the delay time overlaps with other tasks' execution time and therefore, the penalty can be reduced even more. However, it is not the case for small jobs. There has been effort in trying to adaptively adjust the expiry time. Zhu *et al.* [28] introduce a job size estimator in order to adjust this value according to job size. Smaller jobs will benefit greatly from this adaptive expiry time value in case of failure.

Other efforts to improve Hadoop performance under failure include attempts to protect intermediate data. Upon failure, intermediate map output that is stored on the failed machine becomes inaccessible for unfinished reduce tasks, and those tasks need to be re-executed. Ko *et al.* [20] propose an Intermediate Storage System (ISS) that incorporates three replication techniques (i.e., asynchronous replication, rack-level replication, and selective replication) to keep intermediate data safe under failures while minimizing the performance overhead of replication. Another attempt is from Bicer *et al.* [6]. Their system design can be seen as a checkpoint based approach, where the reduction object is periodically copied to another node. Therefore, if one worker fails, its reduction value exists on another node.

Preserving intermediate data can be promising in case of failures, but it induces a very high cost in term of resources (storage space) as well as time (replicating intermediate data to a number of machines is costly). It also affects the resources utilization, as intermediate data is generally only useful during the course of the job, and will be discarded after the job finishes.

There exist other problems regarding the failure handling mechanism of Hadoop that often gets overseen. When a task is declared failed, it gets "special treatment" in the manner that, failed tasks will be launched as soon as any slot becomes available, regardless of data locality. In a cluster where DataNode and TaskTracker processes co-reside, a machine failure will reduce the replication factor for those data splits originally on that node. Given that Hadoop tries its best to provide locality for tasks in normal situations, it is likely that the failed tasks will have one less machine to run locally, which in turn leads to lower locality in general.

Providing locality for tasks is crucial for performance of Hadoop in large clusters because network bisection bandwidth becomes a bottleneck [10][16]. Besides, since most of the Hadoop usage is for small jobs (jobs with a small number of map tasks)[26], it is difficult for a small job to obtain slots on nodes with local data. Data then has to be transferred through the network, which might significantly increase the execution time if network bandwidth is scarce. Providing locality for these jobs will greatly increase the performance of Hadoop in term of time and resource preserving.

Unfortunately, achieving high locality is not easy. Zaharia *et al.* [26] introduce the Delay technique inside the Fair scheduler to improve locality of tasks. Instead of strictly following the order of jobs, Fair scheduler allows behind jobs to launch their tasks first if the head-of-line job fails to launch a local task. However, Fair scheduler bases on an assumption that tasks are mostly short and slots are freed up quickly. In case of long tasks that occupy the slots, node may not free up quickly enough for other jobs to achieve locality.

In the effort to overcome the above-mentioned problems, hereafter, we will briefly discuss the potential benefit of applying preemption. Preemption allows a task to quickly release the slot for more urgent tasks. Locality can be assured with the employment of preemption. Also preemption allows the scheduler to have better control of the resources (i.e., the task slots) so that optimal efficiency

can be obtained. Shorter tasks can preempt a longer one to achieve fast response time.

Hadoop comes with an extreme version of preemption: Kill action. A task can be instructed to stop executing at any time. Traditional Hadoop and its default scheduler Fifo uses Kill to regulate the execution of speculative tasks. When either copy of the original - speculative pair of a task finishes, Hadoop instructs the other to stop to save resource consumption and preserve the correctness of execution. Fair scheduler uses Kill action to ensure fair share to pools. When a pool suffers from under-share for a long enough period of time, Fair scheduler issues Kill actions to over-share pools to reclaim the slots for under-share ones. However, Kill actions have a major drawback of wasting previous work.

## 7 Conclusion

The unprecedented growth in data-center technologies in the recent years opens new opportunities for data-intensive applications on cloud. Many MapReduce-like systems have been introduced as services. MapReduce on cloud provides an economical way for smaller businesses to take advantages of this simple yet powerful programming paradigm. However, users have to pay the expenses of failures, which have become a norm rather than an exception. Thus, fault-tolerance for MapReduce becomes a topic that attracts much interest from both academy and industrial institutes. In this paper, we investigate how Hadoop and its built-in schedulers behave under failures. We observe that the current Hadoop's mechanism, by prioritizing failed tasks and not coordinating with the job schedulers, significantly increases the execution time of jobs with failed tasks, due to the time waiting for free slots and ignoring data locality when re-executing failed tasks. We believe the insights drawn from this paper can serve as guidelines for efficiently deploying and executing data-intensive applications in large-scale data-centers.

As future work, we intend to design and implement a new mechanism to improve the locality of failed task execution. As a first step, we are currently investigating the possibility of building a new preemption technique to allow flexible scheduling of failed tasks.

## 8 Acknowledgments

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/`).

## References

1. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`, Accessed on May 2015.

2. Apache Hadoop Welcome page. `http://hadoop.apache.org`, Accessed on May 2015.
3. Grid'5000 Home page. `https://www.grid5000.fr/`, Accessed on May 2015.
4. Size matters: Yahoo claims 2-petabyte database is world's biggest, busiest. `http://www.computerworld.com/s/article/9087918/`, Accessed on May 2015.
5. Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue Mapreduce benchmarks suite. *ECE Technical Reports. Paper 437*, 2012.
6. Tekin Bicer, Wei Jiang, and Gagan Agrawal. Supporting fault tolerance in a data-intensive computing middleware. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010)*, pages 1–12. IEEE, 2010.
7. Dhruba Borthakur. Facebook has the world's largest Hadoop cluster! http://hadoopblog.blogspot.fr/2010/05/facebook-has-worlds-largest-hadoop.html, Accessed on May 2015.
8. Jeffrey Dean. Large-scale distributed systems at google: Current systems and future directions. In *Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.
9. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)*, pages 137–150, San Francisco, CA, USA, 2004.
10. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
11. Florin Dinu and T.S. Eugene Ng. Understanding the effects and implications of compute node related failures in hadoop. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 187–198, New York, NY, USA, 2012. ACM.
12. Armando Fox, Rean Griffith, A Joseph, R Katz, A Konwinski, G Lee, D Patterson, A Rabkin, and I Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28:13, 2009.
13. Derek Gottfrid. Self-service, prorated supercomputing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-supercomputing-fun/, Accseed on May 2015.
14. Dachuan Huang, Xuanhua Shi, Shadi Ibrahim, Lu Lu, Hongzhang Liu, Song Wu, and Hai Jin. Mr-scope: a real-time tracing tool for mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 849–855, Chicago, Illinois, 2010.
15. Shadi Ibrahim, Bingsheng He, and Hai Jin. Towards pay-as-you-consume cloud computing. In *Proceedings of the 2011 IEEE International Conference on Services Computing (SCC'11)*, pages 370–377, Washington, DC, USA, 2011.
16. Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabirel Antoniu, and Song Wu. Maestro: Replica-aware map scheduling for mapreduce. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, pages 59–72, Ottawa, Canada, 2012.
17. Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating mapreduce on virtual machines: The hadoop case. In *Proceedings of the 1st International Conference on Cloud Computing (CLOUDCOM'09)*, pages 519–528, Beijing, China, 2009.
18. Hai Jin, Shadi Ibrahim, Li Qi, Haijun Cao, Song Wu, and Xuanhua Shi. The mapreduce programming model and implementations. *Cloud Computing: Principles and Paradigms*, pages 373–390, 2011.

19. Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: Right shoes for a running elephant. In *The 2nd ACM Symposium on Cloud Computing*, SOCC'11, pages 21:1–21:14, New York, NY, USA, 2011. ACM.
20. Steven Y Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making cloud intermediate data fault-tolerant. In *the 1st ACM symposium on Cloud computing (SOCC 2010)*, pages 181–192. ACM, 2010.
21. Eric Lai. Companies are spending a lot on Big Data. `http://sites.tcs.com/big-data-study/spending-on-big-data/`, Accessed on May 2015.
22. Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *The 1st ACM symposium on Cloud computing (SOCC 2010)*, pages 51–62. ACM, 2010.
23. Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.
24. B. Thirumala Rao, N. V. Sridevi, V. Krishna Reddy, and L. S. S. Reddy. Performance issues of heterogeneous hadoop clusters in cloud computing. *ArXiv e-prints*, July 2012.
25. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.
26. Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (EuroSys 2010)*, pages 265–278. ACM, 2010.
27. Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pages 29–42, San Diego, California, 2008.
28. Hao Zhu and Haopeng Chen. Adaptive failure detection via heartbeat under hadoop. In *2011 IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 231–238. IEEE, 2011.