# Chronos: Failure-Aware Scheduling in Shared Hadoop Clusters

Orcun Yildiz, Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu

# Chronos: Failure-Aware Scheduling in Shared Hadoop Clusters

Orcun Yildiz, Shadi Ibrahim, Tran Anh Phuong, Gabriel Antoniu
Inria Rennes - Bretagne Atlantique Research Center,
Rennes, France
{orcun.yildiz, shadi.ibrahim, gabriel.antoniu}@inria.fr

***Abstract*—Hadoop emerged as the *de facto* state-of-the-art system for MapReduce-based data analytics. The reliability of Hadoop systems depends in part on how well they handle failures. Currently, Hadoop handles machine failures by re-executing all the tasks of the failed machines (i.e., executing recovery tasks). Unfortunately, this elegant solution is entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. The unawareness of failures therefore may prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of MapReduce applications. This paper presents Chronos, a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while still operating within a specific scheduler objective. Upon failure detection, rather than waiting an uncertain amount of time to get resources for recovery tasks, Chronos leverages a lightweight preemption technique to carefully allocate these resources. In addition, Chronos considers data locality when scheduling recovery tasks to further improve the performance. We demonstrate the utility of Chronos by combining it with Fifo and Fair schedulers. The experimental results show that Chronos recovers to a correct scheduling behavior within a couple of seconds only and reduces the job completion times by up to *55%* compared to state-of-the-art schedulers.**

***Keywords*-Data Management; Scheduling; Failure; MapReduce; Hadoop, Preemption**

## I. Introduction

Due to its simplicity, fault tolerance, and scalability, MapReduce [1], [2] is by far the most powerful programming model for data intensive applications. The popular open source implementation of MapReduce, Hadoop [3], was developed primarily by Yahoo!, where it processes hundreds of terabytes of data on at least *10,000 cores*, and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [4].

Failures are part of everyday life, especially in today's data-centers, which comprise thousands of commodity hardware and software devices. For instance, Dean [5] reported that in the first year of the usage of a cluster at Google there were around a thousand individual machine failures and thousands of hard drive failures. Consequently, MapReduce was designed with hardware failures in mind. In particular, Hadoop handles machine failures (i.e., fail-stop failure) by re-executing all the tasks of the failed machines (i.e., executing recovery tasks), by leveraging data replication.

Hadoop has not only been used for running single batch jobs, it has also recently been optimized to simultaneously support the execution of multiple diverse jobs (both batch and interactive jobs) belonging to multiple concurrent users. Several built-in schedulers (i.e., Fifo, Fair and Capacity schedulers) have been introduced in Hadoop to operate shared Hadoop clusters towards a certain objective (i.e., prioritizing jobs according to their submission times in Fifo scheduler; favoring fairness among jobs in Fair and Capacity schedulers) while ensuring a high performance of the system, mainly by accommodating these schedulers with locality-oriented strategies [6], [7]. 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.

While failure handling and recovery has long been an important goal in Hadoop clusters, previous efforts to handle failures have entirely been entrusted to the core of Hadoop and hidden from Hadoop schedulers. The unawareness of failures may therefore prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of MapReduce applications. When failure is detected, in order to launch recovery tasks, empty slots are necessary. If the cluster is running with the full capacity, then Hadoop has to wait until "free" slots appear. However, this waiting time (i.e., time from when failure is detected until all the recovery tasks start) can be long, depending on the duration of current running tasks. As a result, a violation of scheduling objectives is likely to occur (e.g., high priority jobs may have waiting tasks while lower priority jobs are running) and the performance may significantly degrade. Moreover, when launching recovery tasks, locality is totally ignored. This in turn can further increase the job completion time due to the extra cost of transferring a task's input data through network, a well-known source of overhead in today's data-centers.

Adding failure-awareness to Hadoop schedulers is not straightforward; it requires the developer to carefully deal with challenging yet appealing issues including an appropriate selection of slots to be freed, an effective preemption mechanism with low overhead and enforcing data-aware execution of recovery tasks. *To the best of our knowledge, no scheduler explicitly coping with failures has been proposed.* To achieve

these goals, this paper makes the following contributions:

- We propose Chronos[1], a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while operating within a specific scheduler objective. Chronos takes an early action rather than waiting an uncertain amount of time to get a free slot (thanks to our preemption technique). Chronos embraces a smart selection algorithm that returns a list of tasks that need to be preempted in order to free the necessary slots to launch recovery tasks immediately. This selection considers three criteria: the progress scores of running tasks, the scheduling objectives, and the recovery tasks input data locations.
- In order to make room for recovery tasks rather than waiting an uncertain amount of time, a natural solution is to kill running tasks in order to create free slots. Although killing tasks can free the slots easily, it wastes the work performed by the killed tasks. Therefore, we present the design and implementation of a novel work-conserving preemption technique that allows pausing and resuming both map and reduce tasks without resource wasting and with little overhead.
- We demonstrate the utility of Chronos by combining it with two state-of-the-art Hadoop schedulers: Fifo and Fair schedulers. The experimental results show that Chronos achieves almost optimal data locality for the recovery tasks and reduces the job completion times by up to *55%* over state-of-the-art schedulers. Moreover, Chronos recovers to a correct scheduling behavior after failure detection within only a couple of seconds.

The paper is organized as follows: Section II briefly discusses fault-tolerance in Hadoop. We present the Chronos scheduling strategy in Section III. We then present our experimental methodology in Section IV which is followed by the experimental results in Section V. Finally, Section VI reviews the related work and Section VII concludes the paper.

## II. Fault-tolerance in 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. 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 empty slots, the one with the data on it will be chosen for the map task).

[1]From Greek philosophy, the god of time.

When the master is unable to receive heartbeat messages from a node for a certain amount of time (i.e., failure detection timeout), it will declare this node as failed. Then, currently running tasks on this node will be reported as failed. Moreover, completed map tasks also will be reported as failed since these outputs were stored on that failed node, not in the distributed file system as reducer outputs. For a better recovery from the failures, Hadoop will try to execute the recovery tasks on any healthy node as soon as possible.

## III. Chronos

### A. Design Principles

We designed Chronos with the following goals in mind:

- **Enabling an early action upon failure**: Hadoop handles failures by scheduling recovery tasks to any available slots. However, available slots might not be freed up as quickly as expected. Thus, recovery tasks will be waiting an uncertain amount of time which depends on the status of running tasks (i.e., current progress and processing speed) when failure is detected. Furthermore, during this time, scheduling objectives are violated. Chronos thus takes immediate action to make room for recovery tasks upon failure detection rather than waiting an uncertain amount of time.
- **Minimal overhead**: For the early action, a natural solution is to kill the running tasks in order to free slots for recovery tasks. Although the killing technique can free the slots easily, it results in a huge waste of resources: it discards all of the work performed by the killed tasks. Therefore, Chronos leverages a work-conserving task preemption technique (Section III-C) that allows it to stop and resume tasks with almost zero overhead.
- **Data-aware task execution**: Although data locality is a major focus during failure-free periods, locality is totally ignored by Hadoop schedulers when launching recovery tasks (e.g., our experimental result reveals that Hadoop achieves only *12.5%* data locality for recovery tasks, more details are given in Section V-A2). Chronos thus strongly considers local execution of recovery tasks.
- **Performance improvement**: Failures can severely impact Hadoop's performance. Through eliminating the waiting time to launch recovery tasks and efficiently exposing data-locality, Chronos not only corrects the scheduling behavior in Hadoop after failure but also improves the performance.
- **Schedulers independent**: Chronos targets to make Hadoop schedulers failure-aware and is not limited to Fifo or Fair schedulers. Taken as a general failure-aware scheduling strategy, Chronos can be easily integrated with other scheduling policies (e.g., priority scheduling with respect to the duration of jobs).

Hereafter, we will explain how Chronos achieves the above goals. We will discuss how Chronos allocates the necessary slots to launch recovery tasks, thanks to the tasks-to-preempt selection algorithm and then present our work-conserving preemption technique.

### B. Smart slots allocation

Chronos tracks the progress of all running tasks using the cost-free real-time progress reports extracted from the heartbeats. When failure is detected, Chronos consults the JobTracker to retrieve the list of failed tasks and the nodes that host their input data. Chronos then extracts the list of candidate tasks (running tasks) that belong to nodes where the input data of failed tasks reside. This list is then fed to the tasks-to-preempt selection algorithm (*Algorithm 1*) which first sorts the tasks according to the job priority. After sorting the tasks for preemption, the next step is to decide whether a recovery task can preempt any of these tasks in the sorted list. To respect scheduling objectives, we first compare the priorities of the recovery task and candidate tasks for preemption. If this condition holds, the recovery task can preempt the candidate task. For example, recovery tasks with higher priority (e.g., tasks belonging to earlier submitted jobs for Fifo or belonging to a job with a lower number of running tasks than its fair share for Fair scheduler) would preempt the selected tasks with less priority. Consequently, Chronos enforces priority levels even under failures. The list is then returned to Chronos, which in turn triggers the preemption technique in order to free slots to launch recovery tasks. If the scheduler behavior is corrected, Chronos stops preempting new tasks.

**Algorithm 1:** Tasks-to-preempt Selection Algorithm

```
Data: L_tasks, a list of running tasks of increasing
      priority;
      T_r, a list of recovery tasks t_r of decreasing
priority
Result: T_p, a list of selected tasks to preempt
for Task t_r:T_r do
    for Task t:L_tasks do
        if t belongs to job with less priority compared to
        t_r;
         AND !T_p.contains(t) then
            T_p.add(t);
        end
    end
end
```

### C. Work-conserving Preemption

Preemption has been widely studied and applied for many different use cases in the area of computing. Similarly, Hadoop can also benefit from the preemption technique in several cases (e.g., achieving fairness, better resource utilization or better energy efficiency). In this paper, Chronos leverages it for better handling of failures by pausing the running tasks to make room for the recovery tasks, and resuming them from their paused state when there is an available slot. In this section, we introduce our lightweight preemption technique for map and reduce task preemption.

*1) Map Task Preemption:* During the map phase, TaskTracker executes the map tasks that are assigned to it by the JobTracker. Each map task processes a chunk of data (input block) by looping through every key-value pair and applying the user defined map function. When all the key-value pairs have been processed, JobTracker will be notified as the map task is completed and the intermediate map output will be stored locally to serve the reducers.

For the map task preemption, we introduce an *earlyEnd* action for map tasks. The map task listens for the preemption signal from Chronos in order to stop at any time. Upon receiving the preemption request, this action will stop the looping procedure and split the current map task into two subtasks. The former subtask covers all the key-value pairs that have been processed before the preemption request comes. This subtask will be reported back to the JobTracker as completed as in the normal map task execution. On the other hand, the second subtask contains the key-value pairs that have not been processed yet. This subtask will be added to the map pool for later execution when there is an available slot, as for new map tasks. Full parallelism of map tasks by having independent key-value pairs gives us the opportunity to have fast and lightweight map preemption and also ensures the correctness of our preemption mechanism.

*2) Reduce Task Preemption:* In Hadoop, reduce task execution consists of three phases: shuffle, sort and reduce. During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. The sort phase performs a sort operation on all the fetched data (i.e., intermediate map outputs in the form of key-value pairs). Later, the reduce phase produces the final output of the MapReduce job by applying the user defined reduce function on these key-value pairs.

For the reduce preemption, the splitting approach (as in the map preemption) would not be feasible due to the different characteristics of map and reduce tasks. Full parallelism of map execution and having map inputs on the distributed file system enables us to apply a splitting approach for map preemption. However, the three different phases of the reducer are not fully independent of each other. Therefore, we opt for a pause and resume approach for the reduce preemption. In brief, we store the necessary data on the local storage for preserving the state of the reduce task with pause and we restore back this information upon resume.

Our reduce preemption mechanism can preempt a reduce task at any time during the shuffle phase and at the boundary of other phases. The reason behind this choice is that usually the shuffle phase covers a big part of the reduce task execution, while the sort and reduce phases are much shorter. In particular, the sort phase is usually very short due to the fact that Hadoop launches a separate thread to merge the data as soon as it becomes available.

During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. Then, these intermediate results are stored either on the memory or local disk depending on the memory capacity of the node and also the size of the fetched intermediate results. Upon receiving

a preemption request, the pause action takes place and first stops the threads that fetch the intermediate results by allowing them to finish the last unit of work (i.e., one segment of the intermediate map output). Then, it stores all the in-memory data (i.e., number of copied segments, number of sorted segments) to local disk. This information is kept in files that are stored in each task attempt's specific folder, which can be accessed later by the resumed reduce task.

Preemption at the boundary of the phases follows the same procedure as above. The data necessary to preserve the state of the reduce task is stored on the local disk and then the reduce task will release the slot by preempting itself. The task notifies the JobTracker with a status of *suspended*. Suspended tasks will be added to the reduce pool for later execution when there is an available slot.

## IV. Experimental Methodology

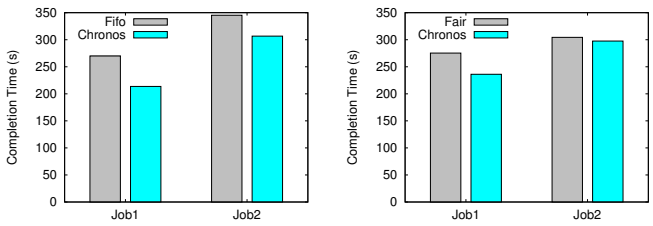
We implemented Chronos in Hadoop-1.2.1. We evaluated Chronos performance on the Grid'5000 [8] testbed, more specifically we employed nodes belonging to the Rennes site of Grid'5000. These nodes are outfitted with *12-core* AMD *1.7 GHz* CPUs and *48 GB* of RAM. Intra-cluster communication is done through a *1 Gbps* Ethernet network.

**Hadoop deployment.** We configured and deployed a Hadoop cluster using *9 nodes*. The Hadoop instance consists of the NameNode and the JobTracker, both deployed on a dedicated machine, leaving *8 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. As suggested in several studies in the literature [9], we set the failure detection timeout to a smaller value (i.e., *25 seconds*) compared to the default timeout of *600 seconds*, since the default timeout is too big compared to the likely completion time of our workloads in failure-free periods.

**Workloads.** We evaluated Chronos using two representative MapReduce applications (i.e., wordcount and sort benchmarks) with different input data sizes from the PUMA datasets [10]. Wordcount is a Map-heavy workload with a light reduce phase, which accounts for about *70%* of the jobs in Facebook clusters [11]. On the other hand, sort produces a large amount of intermediate data which leads to a heavy reduce phase, therefore representing Reduce-heavy workloads, which accounts for about *30%* of the jobs in Facebook clusters [11].

**Failure injection.** To mimic the failures, we simply killed the TaskTracker and DataNode processes of a random node. We can only inject one machine failure since Hadoop cannot tolerate more failures due to the replication factor of *2* for HDFS.

**Chronos implementation.** We implemented Chronos with two state-of-the-art Hadoop schedulers: Fifo (i.e., priority scheduler with respect to job submission time) and Fair schedulers. We compare Chronos to these baselines. The Fifo scheduler is the default scheduler in Hadoop and is widely used by many companies due to its simplicity, especially when the performance of the jobs is the main goal. On the other hand, the Fair scheduler is designed to provide fair allocation of resources between different users of a Hadoop cluster. Due to the increasing numbers of shared Hadoop clusters, the Fair scheduler also has been exploited recently by many companies [7], [12].




(a) Chronos *vs* Fifo scheduler (b) Chronos *vs* Fair scheduler

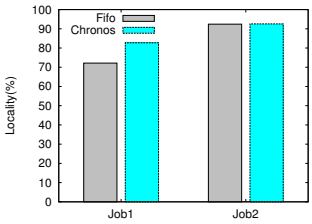
Fig. 1. **Performance comparison for Map-Heavy jobs**

## V. Experimental Results

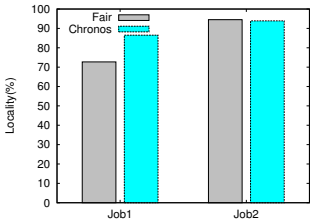
### A. The Effectiveness of Chronos

*1) Reducing job completion time:* **Fifo Scheduler.** We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The input data sizes result in fairly long execution times of both jobs, which allowed us to thoroughly monitor how both Hadoop and Chronos handle machine failures. More importantly, this mimics a very common scenario when small and long jobs concurrently share a Hadoop cluster [7], [13]. Also, we tried to ensure that the cluster capacity (64 map slots in our experiments) is completely filled. After submitting the jobs, we have injected the failure before the reduce phase starts in order to have only map tasks as failed tasks. In contrast to Fifo, Chronos reduces the completion time of the first job and the second one by *20%* and *10%*, respectively. Most of the failed tasks belong to the first job and therefore Chronos achieves better performance for the first job compared to the second one. The main reason for the performance improvement is the fact that Fifo waits until there is a free slot before launching the recovery tasks, while Chronos launches recovery tasks shortly after failure detection. The waiting time for recovery tasks is *51 seconds* (15% of the total job execution) in the Fifo scheduler and only *1.5 seconds* in Chronos (Chronos waited *1.5 seconds* until new heartbeats arrived). Moreover, during this waiting time, recovery tasks from the first submitted job (high priority job) are waiting while tasks belonging to the second job (low priority) are running tasks. This obviously violates the Fifo scheduler rule. Therefore, the significant reduction in the waiting time not only improves the performance but also ensures that Fifo operates correctly towards its objective.

**Fair scheduler.** We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The failure is injected before the reduce phase starts. Figure 1(b)




(a) Data locality in Chronos and Fifo scheduler




(b) Data locality in Chronos and Fair scheduler

Fig. 2. **Data locality for Map-Heavy jobs under Chronos, Fifo and Fair Schedulers**

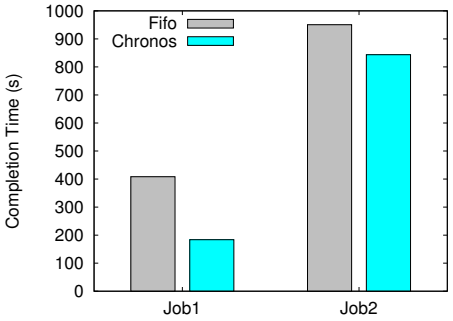



Fig. 3. **Job Completion times for Reduce-Heavy jobs under Chronos and Fifo scheduler**

demonstrates that Chronos improves the job completion time by *2%* to *14%*, compared to Fair scheduler. This behavior stems from eliminating the waiting time for recovery tasks besides launching them locally. We observe that failure results in a serious fairness problem between jobs with Hadoop's Fair scheduler: this fairness problem (violation) lasts for almost *48 seconds* (16% of the total execution time) in the Fair scheduler, while Chronos restores fairness within about *2 seconds* by preempting the tasks from the jobs which exceed their fair share.

*2) Improving data locality:* Besides the preemptive action, Chronos also tries to launch recovery tasks locally. Figure 2 shows the data locality of each job from previous experiments with Fifo (Figure 2(a)) and Fair (Figure 2(b)) schedulers. While the second job has a similar data locality, we can clearly observe that Chronos significantly improves the data locality for the first job for both scenarios (i.e., *15%* and *22%* data locality improvement compared to Fifo and Fair schedulers, respectively). This improvement is due to the almost optimal locality execution of recovery tasks with Chronos (all the recovery tasks which are launched through Chronos are executed locally). Only *12.5%* of the recovery tasks were executed locally in Hadoop. The improved locality brings better resource utilization by eliminating the need for remote transfer of input blocks for recovery tasks and further improves the performance.

*Summary.* The aforementioned results demonstrate the effectiveness of Chronos in reducing the violation time of the scheduler (i.e., priority based on job submission time and fairness) to a couple of seconds. More importantly, Chronos reduces the completion time of the first job (the job was affected by the machine failure) due to the reduction in the waiting time and optimal locality for recovery tasks. This in turn allows the second job to utilize all available resources of the cluster and therefore improves the performance.

### B. Impact of Reduce-Heavy Workloads

We also evaluated Chronos with Reduce-Heavy workloads. We ran two sort applications with input data sizes of *17 GB* and *56 GB*, respectively. Both jobs have *32 reduce* tasks. We injected the failure during the reduce phase in order to have failed reduce tasks from the first job. Figure 3 details the job completion time with Chronos and Fifo. Chronos achieves a *55%* performance improvement for the first job and *11%* for the second one. The improvement in the Reduce-Heavy benchmark is higher compared to the Map-Heavy benchmark because reduce tasks take a longer time until they are completed and therefore the recovery (reduce) tasks have to wait almost *325 seconds* in Fifo. Chronos successfully launches recovery tasks within *2 seconds*.

*Summary.* Chronos achieves higher improvement when the failure injection is in the reduce phase which clearly states that the main improvement is due to the reduction in the waiting time. Here it is important to mention that other running reduce tasks will be also affected by the waiting time as they need to re-fetch the lost map outputs (produced by the completed map tasks on the failed machine).

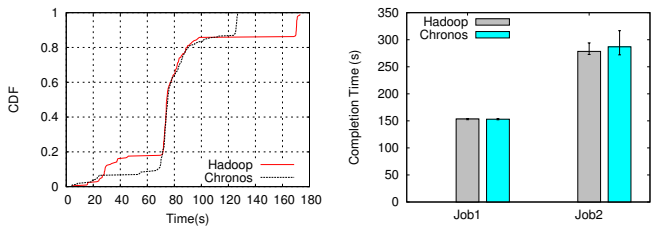
### C. Overhead of Chronos

The overhead of Chronos may be caused by two factors: first, due to the collection of the useful information (i.e., real-time progress reports) that is fed later to our smart slot allocation strategy, and second, due to the overhead of the preemption technique. With respect to the slot allocation strategy, the overhead of Chronos is very little because Chronos leverages the information already provided by heartbeat messages. We have studied the overhead of the preemption technique by repeating the same experiment as in Section V-A1. Figure 4(a) shows the completion times of each successful task with Chronos and Hadoop, we can see that they both have a similar trend. Thus, we conclude that the preemption technique does not add any noticeable overhead to cluster performance in general.

What's more, we studied the overhead of Chronos during the normal operation of the Hadoop cluster. We ran the same experiment as in Section V-A1 five times without any failures and Figure 4(b) shows that Chronos incurs negligible performance overhead during the normal operation.

## VI. Related Work

**Scheduling in MapReduce.** There exists a large body of studies on exploring new objectives (e.g., fairness, job priority) when scheduling multiple jobs in MapReduce and improving their performance. *Isard et al.* introduced Quincy [12], a fair




(a) CDFs of completion times of successful tasks under Chronos and Fifo scheduler

(b) Overhead of Chronos during normal operation

Fig. 4. **Overhead of Chronos**

scheduler for Dryad, which treats scheduling as an optimization problem and uses min-cost flow algorithm to achieve the solution. *Zaharia et al.* introduced a delay scheduler [7], a simple delay algorithm on top of the default Hadoop Fair scheduler. Delay scheduling leverages the fact that the majority of the jobs in production clusters are short, therefore when a scheduled job is not able to launch a local task, it can wait for some time until it finds the chance to be launched locally. Although these scheduling policies can improve the MapReduce performance, none of them is failure-aware, leaving the fault tolerance mechanism to the MapReduce system itself, and thus are vulnerable to incurring uncertain performance degradations in case of failures.

**Failure recovery in MapReduce.** Several studies have been dedicated to improve the performance of MapReduce systems under failures. *Ruiz et al.* have proposed RAFT [14], a family of fast recovery algorithms upon failures. RAFT introduces checkpointing algorithms to preserve the work upon failures. However, an experimental study is performed only with a single job scenario. *Dinu et al.* have proposed RCMP as a first-order failure resilience strategy instead of data replication [15]. RCMP performs efficient job recomputation upon failures by only recomputing the necessary tasks. However, RCMP only focuses on I/O intensive pipelined jobs, which makes their contribution valid for a small subset of MapReduce workloads. Our work is different in the targeting environment, as we focus on shared Hadoop clusters with multiple concurrent jobs.

## VII. Conclusion

In this research work we present Chronos, a failure-aware scheduling strategy for Hadoop. Chronos is conductive to improving the performance of MapReduce applications by enabling an early action upon failure detection. Chronos tries to launch recovery tasks immediately by preempting tasks belonging to low priority jobs, thus avoiding the uncertain time until slots are freed. Moreover, Chronos strongly considers the local execution of recovery tasks. The experimental results indicate that Chronos results in almost optimal locality execution of recovery tasks and improves the overall performance of MapReduce jobs by up to *55%*. Chronos achieves that while introducing very little overhead.

Thanks to these encouraging results, we plan to further investigate the potential benefits of the work-conserving preemption technique. In particular, Hadoop schedulers are still relying on wait or kill primitive to ensure the QoS requirements of several users; thus an interesting direction to explore is how to ensure QoS requirements without wasting the cluster resources.

## VIII. Acknowledgment

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

## References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

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

[3] "The Apache Hadoop Project," http://www.hadoop.org, Accessed on Sep 2015.

[4] "Powered By Hadoop," http://wiki.apache.org/hadoop/PoweredBy, Accessed on Sep 2015.

[5] J. Dean, "Large-scale distributed systems at google: Current systems and future directions," in *Keynote speach at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky, MT, USA, 2009.

[6] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content pogridpularity in mapreduce clusters," in *Proceedings of the sixth ACM European Conference on Computer Systems (EuroSys'11)*, 2011, pp. 287–300.

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

[8] "Grid'5000 Home page," https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

[9] F. Dinu and T. E. 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)*, 2012, pp. 187–198.

[10] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue Mapreduce benchmarks suite," *ECE Technical Reports. Paper 437*, 2012.

[11] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, 2009, pp. 261–276.

[13] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: An analysis of hadoop usage in scientific workloads," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, Aug. 2013.

[14] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE'11)*, 2011, pp. 589–600.

[15] F. Dinu and T. Ng, "Rcmp: Enabling efficient recomputation based failure resilience for big data analytics," in *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS'14)*, May 2014, pp. 962–971.