A Preemptive Fair Scheduler Policy for Disco MapReduce Framework

Augusto Souza[1], **Islene Garcia**[1]

[1] Instituto de Computação
Universidade Estadual de Campinas – Campinas, SP – Brasil

augusto.souza@students.ic.unicamp.br, islene@ic.unicamp.br

Abstract. *Disco is an open source MapReduce framework and an alternative to Hadoop. Preemption of tasks is an important feature which helps organizations relying on the MapReduce paradigm to handle their heterogeneous workload usually constituted of research (long duration and with low priority) and production (short duration and with high priority) applications. The missing preemption in Disco affects the production jobs when these two kinds of jobs need to be executed in parallel: the high priority response is delayed because there aren't resources to compute it. In this paper we describe the implementation of the Preemptive Fair Scheduler Policy which improved largely our experimental production job execution time with a small impact on the research job.*

1. Introduction

MapReduce as proposed by Google in 2004 had the objective of giving developers a computational model in which common distributed systems problems could be addressed by the framework and not by each application [Dean and Ghemawat 2004]. This model is based on a simple abstraction in which programmers need to implement two main functions called Map and Reduce. The first one divides the entry into small groups to be processed in parallel by the cluster's machines, the outputs of the mappers - processes responsible for executing the Map function - are the entry for the reducers. The Reduce function is responsible for combining the data from multiple mappers into a final output. For all of this to work effectively, it is important to have a reliable and distributed file system - e.g. GFS [Ghemawat et al. 2003], HDFS [Shvachko et al. 2010], and DDFS [Disco Distributed Filesystem Website].

Based on Google's MapReduce and File System, some open source projects started to appear and address the same issue in a more accessible way, since the Google's version is only available for their internal use. The main one is Apache Hadoop [Apache Hadoop Website], but there are others like the one we study in this paper, developed by Nokia in 2008 and called Disco [Mundkur et al. 2011]. They both address similar problems but with different architectures and sizes. Hadoop is much larger in terms of code base, investment and complexity than Disco. The last one by being simpler also brings to the table a more flexible solution. To help us compare the complexity in terms of project size we gathered some statistics from their repositories with the help of a tool called *cloc* [Cloc Project at Github.com]. For this, we used the most recent git tags for these projects in the time of this writing: "release-2.7.1" for Hadoop and "0.5.4" for Disco. With much less lines of code than Hadoop (see Table 1), the development of

Table 1. Comparison of projects sizes of Disco and Hadoop.

	Hadoop	**Disco**
Two Most Used Languages	Java and XML	Erlang and Python
Files in Both These Languages	6,474	190
Lines of Code in Both These Languages	1,688,407	19,134

new features in Disco is faster, easier to implement and test, and also has more chances of becoming an improvement integrated in a future project release. Disco is written in Erlang, a language proposed by Ericsson and focused on distributed systems and concurrency, so it can use all the power the language and its virtual machine to support a distributed computation [Armstrong 2003]. Also, Disco MapReduce applications are majorly written in Python, a language with a large history in scientific computing and data processing, also known for being productive and quick to prototype in. In Section 2, we briefly describe how the configuration of Disco can be performed - the cluster administrator is able to do every configuration step through an intuitive web interface - this is another area in which Disco shines when compared to Hadoop and its dozens of XML files of configuration.

The large adoption of the MapReduce model by the industry caused a necessity not yet addressed by Disco, but already addressed by Hadoop: preemption of tasks. Often, clusters are used by different types of applications with different kinds of priorities, and for some uses it is important to be able to suspend the execution of parts of the applications with less priorities as soon as another with more priority arrives. We divide the jobs of this heterogenous workload into two main categories: research (long duration and with low priority) and production (short duration and with high priority) [Cheng et al. 2011]. Nowadays, Disco is only able to schedule applications respecting FIFO (First in First Out) and Fair policies. Both have serious limitations: FIFO needs to wait for a job to complete in order to schedule a new one, while Fair takes time to give to the production (high priority) application its resources. The limitation related to Fair scheduler policy is our concern in this paper. With no preemption of tasks, the production job waits for the tasks of any running research job to finish before being able to share the cluster resources and starts its execution. With the preemption mechanism we propose, the applications submitted to the Preemptive Fair Scheduler should get its resources as soon as possible.

Besides the submission of tasks, another critical period for scheduling tasks in Disco environment occurs between transition phases of the jobs. These are the phases between the end of the map and the begin of the shuffle and the end of the shuffle and the begin of the reduce. It is during shuffle that reducers copy the mappers output to start their computation. The Preemptive Fair Scheduler waits the transition phases to finish in order to get more accurate information about the running jobs and allocate the resources in accordance to its policy.

The rest of the paper is organized as follows. We present Disco architecture and its schedulers in detail in Section 2. In Section 3, we explain the proposed Preemptive Fair Scheduler Policy and advocate for its necessity. In Section 4, we provide some results obtained by our implementation. In Section 5, we discuss related work. We analyze future work and conclude in Section 6.

2. Disco's Architecture

The architecture we study is the one of version 0.5.4 of Disco. In this version, just like the Hadoop pre-YARN (Yet Another Resource Negotiator) [Vavilapalli et al. 2013], Disco works with a simple master-slave architecture. It is able to achieve its objective of running on a large cluster of networked commodity machines with this simple model. The master is responsible for two key activities:

- initialization of every other component of Disco, like the logging mechanism, web interface, and workers which run in the slaves nodes;
- scheduling, monitoring and allocating resources for tasks from new job submissions of the clients.

The Disco's master needs few configurations to start its slaves: just the number of workers each node will run and their hostnames. The usual is to give to the hosts the same number of workers as they have of CPU cores, since workers are used to run job tasks. After this setup, the master uses the hostnames to launch a single slave Erlang node in these hosts. The slave nodes are responsible for launching and monitoring the execution of tasks specified by the submitted jobs. Also, the slave nodes are the units of storage used by Disco Distributed File System (DDFS) [Disco Distributed Filesystem Website]. Administrators can add and remove slave nodes in real time.

The decision of using the Erlang Virtual Machine for Disco's core shines when we take a look at how nodes interact with each other. The connections between master and slaves are monitored and maintained by Erlang's monitor node. Since the language and its virtual machine have been designed especially for distributed systems, there are useful features provided out of the box for handling things like communication, fault tolerance, reliability and scalability. This way when a node fails, the master is notified and can reschedule the task in another host. Also, all the clustering setup is not responsibility of Disco, it can rely on the Erlang Virtual Machine for communication between cluster machines.

A summary of Disco's master-slave architecture and the communication between these nodes is illustrated by Figure 1. One of the objectives in this architecture is to let Disco MapReduce applications to be written in a range of programming language. In order to achieve this design goal, there is a protocol for the communication of the slave nodes and its workers, nowadays besides the default client in Python there are others in OCaml [OCaml Worker at Github.com], Golang [Go Worker at Github.com], LFE [LFE Worker at Github.com], and Haskell [Haskell Worker at Github.com], giving the programmers the ability to use the programming language they feel more comfortable with. The protocol uses the standard error file descriptor (stderr) for messages from the worker to Disco slave and the standard input file descriptor (stdin) for messages from the Disco slave to the worker. The message flow is always initiated by the worker and there are messages to notify about a bunch of events, such as the startup, task information request, errors while retrieving inputs, and others.

Another interesting piece of the Disco's architecture is its Pipeline. Like Hadoop developed YARN to support a more vast range of applications than just MapReduce, Disco developed the Pipeline for similar purpose. The idea is to define jobs as a linear sequence of stages constituted of a computation and a grouping function. The output

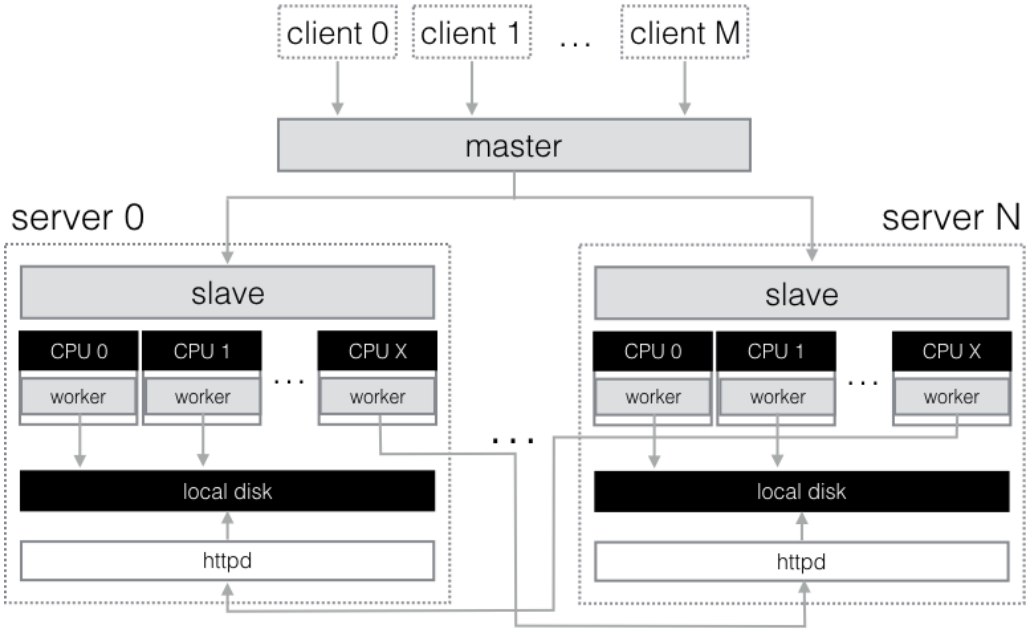

Figure 1. Disco's architecture overview. Gray scale squares are the main components (master, slave and workers) [Mundkur et al. 2011].

of a stage is the input of the next one and there are a bunch of pre-defined grouping possibilities to help the programmers of Disco applications.

Disco has two possible scheduling policies by default: FIFO and Fair. The first one works as a simple queue in which the running job gets all cluster resources it needs and executes up to its end while the others wait. In FIFO policy, when there are available resources the first job in the queue is always selected by the scheduler, if it has no more tasks ready to be executed the next job in queue is chosen. The Fair scheduler tries to divide the cluster in equal shares giving the running jobs the same amount of cluster resources in average to run their tasks. In order to achieve this behavior the Fair scheduler policy cycles through the running jobs when asked to select a task from one of them to give an available resource to. Also, it calculates a deficit that is increased by the amount of time a job had to wait or had fewer resources than the other jobs. The default fair policy always try not to have unused resources while there is work to be done.

Figure 2 shows a case in which four jobs have been submitted to a cluster. The left side of the figure, shows a scheduler configured with FIFO policy, while the right side shows the case for Fair policy. The scheduler makes three of these jobs wait in a queue for the FIFO policy scenario. In the Fair policy case, the scheduler does not need a queue, instead it uses a pool to store the information related to the running jobs and without prioritization it tries to schedule every job with the same amount of resources.

Both these policies have problems when trying to work with a workload composed of research and production jobs. FIFO does not divide the cluster resources within the running jobs, so the production job starts only after a long time, because the research job takes every resource of the cluster for a long time. Fair policy is not ready to preempt running tasks of the research job to give the resources needed by the production job faster. Also, Fair policy isn't ready to avoid the research job from taking every resource of the

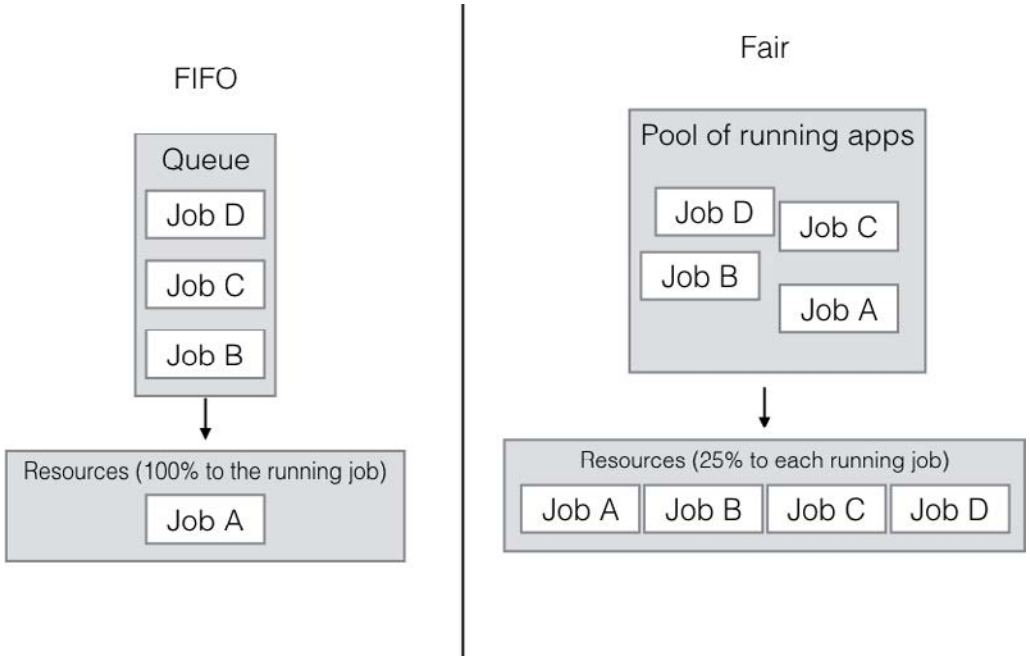

Figure 2. Cluster resources distributions comparison for both FIFO and Fair scheduler policies of Disco.

job during transition phases. These are special periods in a job lifetime for Disco and if the scheduler policy does not monitor that the production job is in this period and is still calculating the amount of remaining work, it can make a mistake by giving the resources for the research job. It greedily gets and locks more resources than it should. The Preemptive Fair Scheduler Policy we propose in this work addresses these limitations.

3. Preemptive Fair Scheduler Policy

Disco's scheduler has a pluggable policy architecture to rely on when some events occur, e.g. a new job submission, the end of a job's execution or an new available slot for task execution. Respecting this pluggable architecture, there are already FIFO and Fair policies as options for the cluster administrator as we described in the previous section. We decided to implement a new policy to add preemption of tasks to the scheduler. The Preemptive Fair Scheduler Policy acts when one of the following events occur:

New job submission:
In order to be fair with the new job as soon as possible, the policy calculates the fair share for it and kill tasks evenly from the running jobs making room for it;

New slot is available:
There is room in the cluster for a new task to be scheduled in, but the Preemptive Fair Scheduler policy must first check if there is a running job in a transition phase. If there is one, then it is better to wait by not scheduling any job in the available slot, because the amount of remaining work is still unknown by Disco at this moment. If there is no job in a transition phase, then the policy checks if a recently submitted job needs to get its first resources, if so it gives them as soon as the job is able to be executed; if not, it makes a list of the jobs sorted by their

running tasks quantity, then filter only those with the number of running tasks smaller than the fair share, the most unfavored job is the first one in this list and is the one that is chosen to use the available slot;

Execution of job finished:
The policy removes this job from every internal data structure it has to maintain to help with the scheduling decisions;

This policy focus on being fair by giving the new job resources for its execution as soon as possible using preemption. It also makes sure the fair number of resources will be available for the desiring job even if some waiting needs to be done during transition phases.

4. Experiments

Our experiments have been done with the sponsorship of Digital Ocean [Digital Ocean Website], we configured a 5 nodes cluster (connected as a private network) based in their New York data center. Each node has 2 cores, 2 GB of RAM memory, 40 GB of SSD Disk, and 3 TB transfer. They are running Disco on top of Ubuntu 14.04.3 LTS, Erlang/OTP R16B03, and Python 2.7.6. We compared two scheduler policies of Disco, for this we used a setup with the original version 0.5.4 got from the Github repository [Disco at Github.com] configure with the original Fair Scheduler Policy and a setup of our own adapted to run the Preemptive Fair Scheduler Policy [Disco Fork at Github.com].

In order to simulate the heterogenous workload of research and production jobs we use the canonical Word Count MapReduce example, in which large files of plain text are the inputs and the application calculates the number of times each word appears in these texts. Our research job has 25 mappers and a sleep interval of 1 millisecond per processed word during map phase. The objective of this sleep is to turn its tasks into long time consuming activities, while our production job is a Word Count with the same number of mappers but with no sleep time. The sleep time is the simplest yet effective way for us to experiment with long running computations. The time a task is running is what will affect the scheduler decisions, the kind of the computation being performed does not matter. The second job is submitted 20 seconds after the first job starts. Charts in Figure 3 compare the total execution time for the production job after 10 runs of this experiment in both policies.

The preemption mechanism and the special treatment we give to the transition phases of the job produced an interesting result: our production job now runs 8.4 times faster in average. But this improvement comes with a cost: an impact in the research job execution time as shown in Figure 4. In average, with the Preemptive Fair Scheduler policy the research job is 4% slower. It had the execution time increased in 9 seconds in our experiments.

Figure 5 shows the cluster resources division between research and production jobs during their lifetime for the original Fair Scheduler Policy. It is possible to observe two large waiting times to execute the production job. The first one is at the beginning of the production job, just after it is submitted: it only starts to run after 1 minute waiting

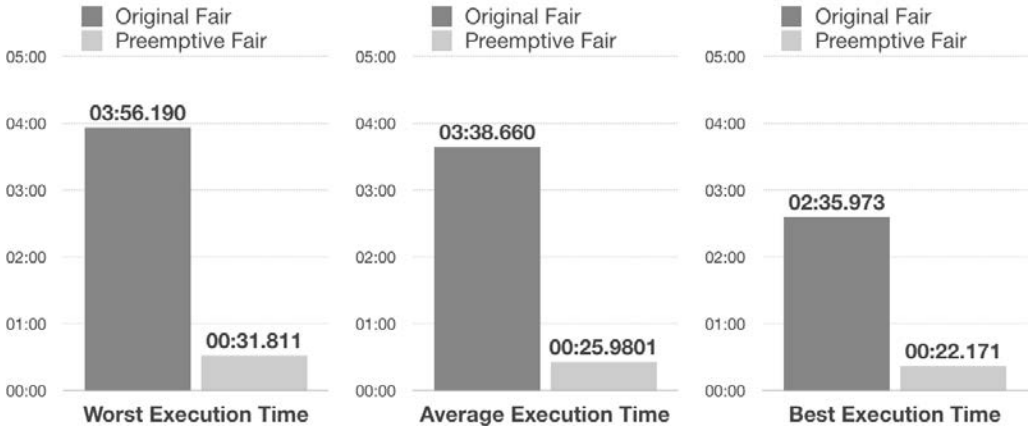

Figure 3. Production job execution times.

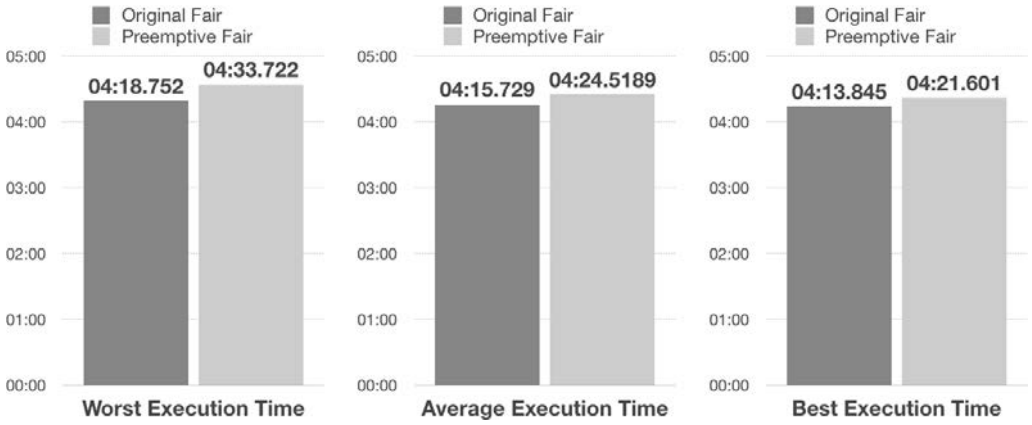

Figure 4. Research job execution times.

(01:20s in the chart, remembering that it was submitted 20 seconds after the research job). The second waiting happens between the end of the map phase and the start of shuffle phase: the production job has no tasks running between 01:41s and 02:42s.

The Preemptive Scheduler policy tries to address this two waiting periods as seen in Figure 6. So the production job gets its resources right away and there is almost no waiting after its submission. The amount of work lost because of the preemption is proportional to the fair share of the resources and the interval before the job submission in our experiments, so 100 seconds of computation is lost because we have 5 slots for each job and 20 seconds of interval. But, in a general case, the lost work could vary, since each task could be in a different stage of the computation.

Also, while running the scheduler with the Preemptive Fair Policy there is a period in which the cluster is underutilized. This happens during the shuffle and reduce phases of the production job, because the policy decides it is better to wait than to allocate resources for the greed research job. In Figure 6 check that between 00:20s and 00:30s in the chart there is a period in which the sum of running tasks for both jobs is less than the available resources (10 nodes). This corresponds to the period in which the amount of remaining work is still unknown by Disco because there is a job in a transition phase, as we described in section 3. Like Figure 5 shows, the research job with its long running tasks would take

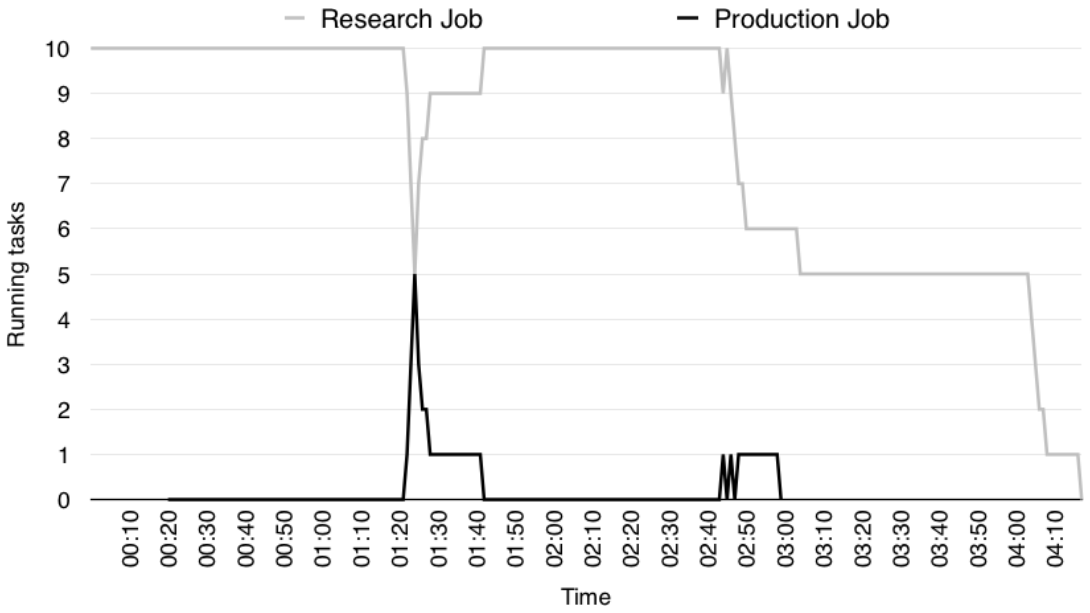

Figure 5. Cluster utilization for Disco configured with the original Fair scheduler policy.

every resource of the cluster and make the production job wait for a long time.

5. Related work

The closest work and one of the main inspirations for this is the Hadoop YARN preemption mechanism [Vavilapalli et al. 2013]. It works with the Hadoop's Capacity Scheduler, a feature enhanced version of the Fair Scheduler based on being Fair with the jobs between the same queue while each queue gets a percentage of the cluster's resources. Also, it preserves the work of the tasks to be preempted with a checkpointing service (which is something we intend to look as a future work). Finally, it tries to preempt tasks that made only little progress not any task like ours. These changes are still open issues in the Hadoop's source code changes control system (JIRA) [MAPREDUCE-5269 , MAPREDUCE-4584], so even they could be tested for the time the paper was published they are not available for the majority of Hadoop's users.

There is competing work regarding YARN preemption that was not accepted by Hadoop's community [Cho et al. 2013]. It is based in an interesting approach of eviction policies for both jobs and tasks, separating the logic necessary for the scheduler to chose the job that will be penalized with preemption and then the task from this job that will be preempted. This two level preemption policy is simple and powerful. Our work nowadays does not look in policies with this separation, our policy only choses the job to preempt a task from and do not chose a task wisely (we only chose randomly from the set of running tasks). The separation of policies could be a future work.

The idea of waiting a little before taking a scheduling decision have been explored for Hadoop [Zaharia et al. 2010]. This is similar to what we have done during job's transition phases. Differently than our scheduling decision that waits to have more information about the remaining work, it prefers to wait for some tasks to complete and use their vacant slots to schedule other task that will be benefitted by the data-locality

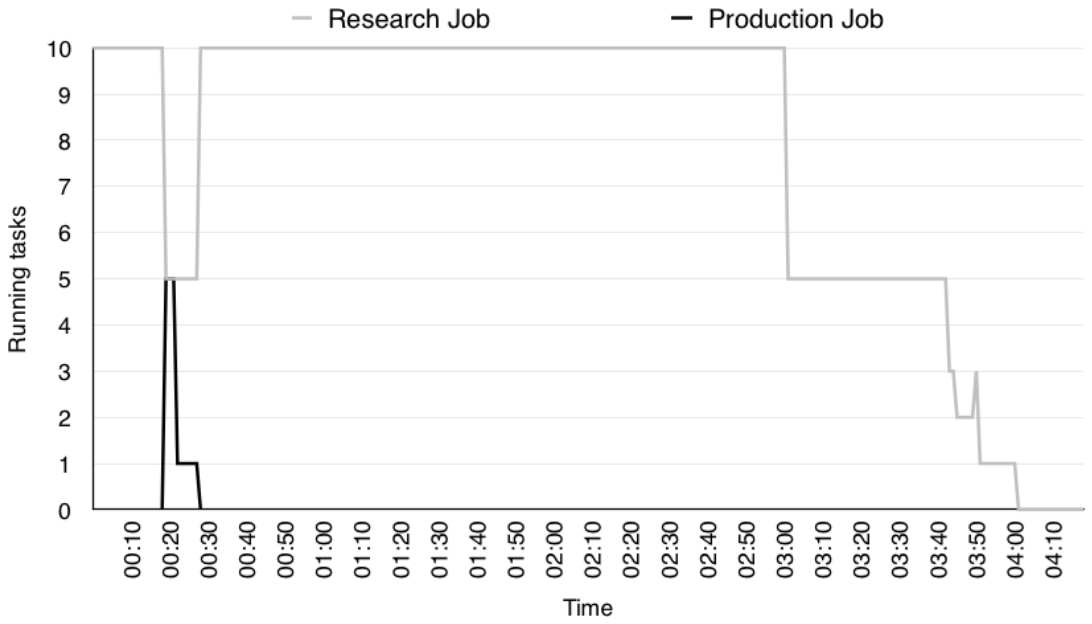

Figure 6. Cluster utilization for Disco configured with the Preemptive Fair Scheduler policy.

of the inputs it needs to use. Even Disco being location aware of its Data, we decided not to look at data-locality before preempting tasks at this moment. Another interesting information that improves Hadoop's jobs is an estimation of the end time of the jobs before deciding if a backup task can improve its execution time [Zaharia et al. 2008]. This is interesting for Hadoop, but Disco doesn't work with the concept of stragglers and backup tasks. Neither these works support preemption of tasks.

The problem we have with schedulers that greedily allocate resources for long running jobs also occurs in Hadoop schedulers [Wang et al. 2013]. Preemption of tasks in a work-preserving manner for the long running reduce tasks can help solve this problem after a proven not so good scheduling decision is made. Our work shows that this issue can happen in Disco too, and introduces both a preemption mechanism and a delay scheduling during transitory phases to improve production job's execution time. But, we underutilize the cluster during the reduce phase of the production job to avoid the greedily behavior of the research job.

Finally, the heterogeneous workload scheduling problem in which research and production jobs compete for cluster's resources and commonly the scheduling policies can't be fair by their defaults implementations exists for Hadoop too. The Global Preemption (GP) technique [Cheng et al. 2011] acts mainly in the tasks to be preempted decision of the scheduler. In Hadoop's default implementation the most recent tasks that overflow the running jobs share are preempted to give room for the tasks of the just submitted job. GP proposes not to look at tasks from each job separately, but to check every running task in the cluster and kill the newly launched ones, even if fairness can be lost during this process for a small amount of time. Our work prefers to stick with the default implementation since we doesn't look at tasks running time before preempting yet.

Every related work described in this section have been implemented for Hadoop, our work intends to take some of the good mechanisms Hadoop have to a less explored system: Disco. We also think Disco is a great project for MapReduce related research, since it is open source, simple, robust, mature, well architected and uses the Erlang Virtual Machine to address most of the distributed systems problems related to network communication, giving the researchers the freedom to focus on new features instead of configuration.

6. Conclusion

This paper describes a work performed in a robust, elegant and not so explored MapReduce system called Disco. Using the power of Erlang in a well architected pluggable scheduler policy environment, we have been able to address the heterogenous workload problem in this system by creating the Preemptive Fair Scheduler Policy. In order to implement this policy only a small quantity of lines of code were changed and added to Disco's source code. This way it can continue to be a lean alternative to Hadoop.

Usually, MapReduce jobs have different priorities and running times and can be divided into two main categories: research (long duration and with low priority) and production (short duration and with high priority). The default schedulers provided by Disco doesn't support preemption of tasks and are not able to address this kind of workload respecting the job's priority and staying fair. Using previous researches from the well known Hadoop framework as basis, we could adapt best practices related to preemption and to delayed scheduling decisions to a new Fair scheduler policy, achieving great results in our experimental environment. The application with more priority runs 8.4 times faster in average with the Preemptive Fair Scheduler Policy when compared to the default Fair Scheduler Policy. But this optimization comes with a small price: underutilization of the cluster during the production job final phases and the low priority research job has a 4% increase in its execution time.

For future work, we want to preserve the preempted tasks state with a checkpointing mechanism. We intend to avoid the cluster underutilization during the shuffle and reduce phases of the production job: in our experiments, the research job could have taken more resources than those the scheduler gave during this period. Another great improvement would be to only preempt when jobs with more priority are submitted, to achieve this we think in giving the clients the power to send jobs to Disco with a parameter indicating if preemption of tasks must be enabled for this submission. Finally, we want to take these ideas to Disco's community and work more closely to them in the most important features.

Acknowledgements

Special thanks to Digital Ocean [Digital Ocean Website] for the sponsorship for our experiments.

References

Apache Hadoop Website. `http://hadoop.apache.org/`. Accessed: 2016-04-07.

Armstrong, J. (2003). Making reliable distributed systems in the presence of software errors.

Cheng, L., Zhang, Q., and Boutaba, R. (2011). Mitigating the negative impact of preemption on heterogeneous mapreduce workloads. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 189–197, Laxenburg, Austria, Austria. International Federation for Information Processing.

Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., and Lin, P. (2013). Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 6:1–6:17, New York, NY, USA. ACM.

Cloc Project at Github.com. `https://github.com/AlDanial/cloc`. Accessed: 2016-04-07.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

Digital Ocean Website. `https://www.digitalocean.com`. Accessed: 2016-04-07.

Disco at Github.com. `https://github.com/discoproject/disco`. Accessed: 2016-04-07.

Disco Distributed Filesystem Website. `http://disco.readthedocs.org/en/latest/howto/ddfs.html`. Accessed: 2016-04-07.

Disco Fork at Github.com. `https://github.com/discoproject/disco`. Accessed: 2016-04-07.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA. ACM.

Go Worker at Github.com. `https://github.com/discoproject/goworker`. Accessed: 2016-04-07.

Haskell Worker at Github.com. `https://github.com/zuzia/haskell_worker`. Accessed: 2016-04-07.

LFE Worker at Github.com. `https://github.com/oubiwann/lfe-disco`. Accessed: 2016-04-07.

MAPREDUCE-4584. Umbrella: Preemption and restart of MapReduce tasks - JIRA. `https://issues.apache.org/jira/browse/MAPREDUCE-4584`. Accessed: 2016-04-07.

MAPREDUCE-5269. Preemption of Reducer (and Shuffle) via checkpointing - JIRA. `https://issues.apache.org/jira/browse/MAPREDUCE-5269`. Accessed: 2016-04-07.

Mundkur, P., Tuulos, V., and Flatow, J. (2011). Disco: A computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 84–89, New York, NY, USA. ACM.

OCaml Worker at Github.com. `https://github.com/discoproject/odisco`. Accessed: 2016-04-07.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA. ACM.

Wang, Y., Tan, J., Yu, W., Zhang, L., Meng, X., and Li, X. (2013). Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 279–289, San Jose, CA. USENIX.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010). Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA. ACM.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. (2008). Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA. USENIX Association.