MapReduce “Garbage” Collection

Shady Khalifa, Tianbin Jiang, Patrick Martin
School of Computing
Queen’s University
ON, Canada
{khalifa, jiang, martin}@cs.queensu.ca

Abstract

Recently, Hadoop, an open source implementation of MapReduce, has become very popular due to its characteristics such as simple programming syntax, and its support for distributed computing and fault tolerance. Although Hadoop is able to automatically reschedule failed tasks, it is powerless to deal with tasks with poor performance. Managing such tasks is vital as they lower the whole job's performance. Thus in this work, we design a novel garbage collection technique that identifies and collects “garbage” tasks. Three research questions are addressed in this work. The first, does collecting (shutting down) garbage (slow) tasks help in reducing the total job completion time and resources cost? The second, when is it most efficient to invoke the Garbage Collector? The third, how to identify garbage (slow) tasks and what are the major factors causing a task to slow down?. The proposed Garbage Collector is evaluated on Amazon EC2 using two metrics: (i) the time for a single job completion, and (ii) resource costs. The empirical results using the TeraSort benchmark show that collecting garbage tasks does reduce the job completion time by 16% and resources cost by 27%. The results also show that the Garbage Collector needs to be invoked before the job is 40% completed, otherwise it would be better to leave the slow tasks till the end of the job because at this point the cost of re-executing these slow tasks becomes high. Finally, our results show that CPU utilization is a good indicator of slow tasks.

1 Introduction

There are numerous recent examples of the benefits obtained from analyzing the large data sets produced by today’s applications. Internet services such as e-commerce websites and social networks generate click-stream data from millions of users every day. Understanding patterns in these enormous volumes of data means increased advertising revenue. Also, analyzing the huge amount of logs generated by users’ actions in a timely manner indicates that developers and operators can operate more efficiently in diagnosing problems in production. In the Big Data era, the volume of data that needs to be processed and analyzed has led to an increasing interest in parallel processing on commodity clusters.

The MapReduce framework, which was originally developed at Google [1], is the pioneer in parallel processing for enormous data sets. Features such as elastic scalability and fine-grained fault tolerance contributed to its wide adoption. Google for instance uses its MapReduce framework to process 20 petabytes of data per day [1]. Hadoop [2], an open source implementation of MapReduce developed by Yahoo!, is in use by many organizations like Facebook, Twitter and Yahoo! [11].

While Hadoop offers elastic scalability and fine-grained fault tolerance, it lacks strength in dealing with slow tasks, which has become a stumbling block to its performance. Its strategy for dealing with slow tasks is to activate several speculative

tasks when detecting some tasks running slower than others in the job. Hadoop provides an option to launch duplicates of slow tasks on different nodes, with the hope that they will finish faster than the original slow tasks. The motivation for this feature is that it has been found that every job has stragglers- a small percentage of tasks that are significantly slower than the rest [2]. These slow tasks increase the overall execution time of the job. These stragglers typically appear due to hardware problems. The speculative task mechanism trades resources for speed which might not always be the best option.

In order to speed up Hadoop without the increased resource costs, we propose a "Garbage Collector", which is based on the idea of "garbage collection" in the Java Virtual Engine [3]. In the world of Java, a garbage collector is used for automatic memory space management. But, in this paper, "garbage collection" refers to reaping Hadoop tasks with poor performance automatically. These collected tasks are then moved on faster nodes. Generally speaking, the proposed "Garbage Collector" acts as an assistant to the Hadoop scheduler to make more efficient use of the computing resources.

Two contributions are made in this work. First of all, we expand the meaning of "Garbage Collection" and use it to identify "garbage" tasks. Second, we propose using Garbage Collection to help Hadoop maintain better performance through collecting garbage tasks before they affect the overall job's efficiency.

The remainder of this paper is organized as follows. After introducing MapReduce and Hadoop in Section 2, we define the downfalls of the current implementation in Section 3. Section 4 covers the Garbage Collector design. Our experimental environment and results are presented in Section 5. In Sections 6 and 7, we discuss related work, and present our conclusions and future work, respectively.

2 Background

Before defining the problem, we first present a brief introduction to the basic MapReduce and Hadoop frameworks. MapReduce is a runtime system that allows parallel execution of tasks over a cluster of nodes. MapReduce takes two input functions (*Map* and *Reduce*) written by the programmer. The *Map* function processes input data to generate intermediate data in the form of <*key, value*> tuples. The *Reduce* function then merges the values associated with a key. Each of the *Map* and *Reduce* functions are then executed in parallel over a set of distributed data files in a Single Program Multiple Data (SPMD) paradigm. Data files are stored on a distributed file system such as Google File System (GFS) [12] or Hadoop Distributed File System (HDFS) [13]. MapReduce is constructed of three phases, *Map*, *Shuffle*, and *Reduce*, where the phases are executed semi-sequentially and each phase utilizes all the nodes in the MapReduce cluster.

In the *Map* phase, the programmer-provided Map function is executed in parallel over the cluster. Input data is divided into chunks and stored in a distributed file system. Each Map task reads some number of chunks and generates intermediate data which is used as an input to the *Reduce* phase. Intermediate data is then partitioned into a number of chunks and stored locally on the nodes that generated them.

In the *Shuffle* phase, intermediate data chunks are moved from the local storage of the Map nodes to the appropriate Reduce node. In this phase, tuples are grouped on the key field and then all tuples for a particular key are sent to a single Reduce task. A Reduce task may process more than one key since usually the number of keys is much more than the number of Reduce tasks. This phase requires an all-Map-to-all-Reduce communication pattern, thus it heavily utilizes the network.

In the *Reduce* phase the programmer-provided Reduce function is executed in parallel over the cluster. The output of this phase is then stored on the distributed file system. In case of multi-stage MapReduce, this output is used as an input for the next MapReduce stage.

The MapReduce runtime system automatically partitions the input data, schedules the Map tasks across the nodes, runs the Shuffle, schedules the Reduce tasks, and re-executes computations when nodes fail. MapReduce tracks the execution status of nodes and assigns new tasks to free nodes considering data locality. MapReduce speculatively creates backup copies of straggler tasks and terminates all outstanding copies when one copy finishes. Straggler tasks are defined as tasks that take an unusually long time to execute and delay the completion of a phase.

Hadoop, the MapReduce implementation, has a single JobTracker (Master) managing a number of TaskTrackers (Slaves). The JobTracker is responsible for scheduling and monitoring jobs across the cluster. JobTracker tasks include detecting and tagging failed or slow jobs, duplicating slow executions, and restarting failed ones. Input data which resides on a HDFS is split into even-sized chunks.

Hadoop divides each *job* into a set of *tasks*. Each chunk of input is first processed by a Map task, which outputs a list of key-value pairs generated by a programmer-defined Map function. Map outputs are split into buckets based on key. When Maps are finished, Reduce tasks apply a reduce function to the list of Map outputs with each key. Hadoop runs several Maps and Reduces concurrently on each TaskTracker – two of each by default – to overlap computation and I/O [4]. Each TaskTracker tells the JobTracker when it has empty task slots. The scheduler then assigns it more tasks.

Hadoop's scheduler makes several implicit assumptions [4]. The first assumption is that nodes can perform work at the same rate and tasks progress at a constant rate throughout time. This assumption is not valid in a virtualized environment as other VMs running on the same physical machine can affect the performance of nodes and tasks. The second assumption is that a task's progress is approximately equal to its percent completion. Tasks whose progress deviates from the normal progress for a phase by more than a static threshold are considered to be slow tasks. This assumption is also not valid because a task consists of a number of phases and some of these phases (ex. processing data) execute faster than others (ex. reading data from storage). So with these assumptions, Hadoop will incorrectly identify a task in the reading phase as a slow task if the other tasks have started working on the processing phase. This is not necessarily correct since the slow task could catch up with the other tasks once it finishes the read phase.

3 Problem Definition

In this section, we illustrate the downfalls of the current implementation of Hadoop and their effect on the overall job performance.

- A *Reduce* task can only start after a set of *Map* tasks finish and generate the intermediate data. Thus, if some *Map* tasks in this set suffer poor performance, then this will delay the *Reduce* task execution which will affect the total job completion time.
- In Hadoop, speculative execution is used to support fault tolerance. The JobTracker keeps track of all scheduled tasks. The speculative tasks, which are launched when Hadoop finds the task is unusually slow compared with others of the same job, will process the same input data with the hope that it will complete much earlier than the original one. Speculative task mechanism trades resources for speed. The number of Speculative tasks should be minimized to maximize efficiency. The current default mechanism fails to pay attention to the number of speculative tasks which increases the cost.
- Speculative tasks are activated only when certain conditions are met. For instance, there should be no previously running speculative tasks, most of the job's tasks are finished, and a task must be less than 20% completed after running for 60 seconds. As far as we know, this algorithm suffers from many problems. One of the problems is that there is a possibility that some task might run fast at the beginning but then slow down or even get blocked when it is almost finished. This problem is beyond the scope of the above algorithm. Another problem is that the speculative task mechanism does not allow clients to configure the speculative time gap and percentage of job complication, making the mechanism inflexible to the different jobs needs.
- With the current Hadoop's scheduler assumptions, tasks can be wrongly identified as slow tasks due to the method used for calculating tasks' progress and the use of static threshold.
- Since *Map* tasks access the input data chunks over the distributed file system, replicating slow tasks (running a speculative copy of this task on another node) will increase the network usage, which can lead to the network becoming a bottleneck.

The goals of this work are to determine if shutting down slow tasks instead of replicating them (speculative tasks) can be more efficient. We then determine the deadline for shutting down these slow tasks. We only address the problem of slow tasks and leave faulty tasks to Hadoop's scheduler.

4 Garbage Collector Design

The Java garbage collector is used to manage memory space automatically. Before releasing an object's memory space, the garbage collection thread invokes a specific method to perform any sort of cleanup required. In normal operation, the garbage collector runs concurrently with the applications. It is able to do most of its work while the application is paused and completes when all the application threads have stopped.

In this paper, we propose to use the idea of the "Garbage Collector" to reap the slow tasks in Hadoop and allocate them seamlessly on faster nodes.

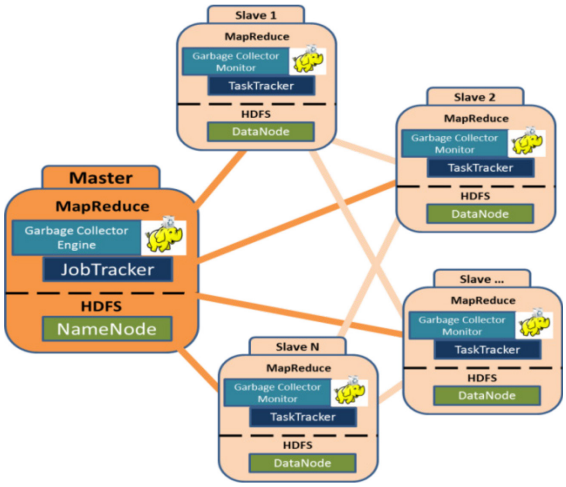

Figure 1. Garbage Collector Design

In fig.1, we illustrate the high level design of the proposed Garbage Collector. The Garbage Collector consists of a main engine which resides on the Master node and a set of monitors on the Slave nodes. Having the engine on the Master node allows it to collect the information reported to the JobTracker from the TaskTrackers and also allows it to command the JobTracker to take actions towards the slow tasks. The Garbage Collector monitors on the slave nodes collect the CPU, Memory and I/O utilization and report back to the Garbage Collector engine on the Master node.

The Garbage Collector can be either implemented as a separate application that communicates with Hadoop using APIs or it can be implemented as a part of the Hadoop scheduler. Each method has its own advantages and disadvantages. The separate application approach is easier and faster to implement. However, it is limited to the functionalities and performance reports provided by Hadoop APIs and presents an extra deployment step for the Hadoop users. On the other hand, implementing the Garbage Collector as a part of Hadoop scheduler is more difficult but makes it easier to deploy and provides the Garbage Collector with more performance information, allowing it to make more informed decisions.

In the next section, we run a set of experiments to identify the key factors the Garbage Collector must consider while making decisions to collect a garbage node.

5 Evaluation and Analysis

We used Amazon EC2 and Hadoop 1.0.4 stable release running on Ubuntu 12.04.1 LTS to analyze the effectiveness of the proposed Garbage Collector. Two Hadoop virtual clusters were built to make sure that the results are not affected by the state of the cloud hardware. Each cluster consists of 6 "small"-size VMs. One VM (Master VM) acts as the JobTracker/NameNode while the rest (Slave VMs) are TaskTrackers/DataNodes. The Master VM is also running a TaskTracker/DataNode, so in total we have 6 TaskTrackers/DataNodes. Each data block is replicated on 2 other nodes. "Small"-size EC2 VMs were used running with 1.7 GB of memory, 1 virtual core 1.0-1.2 GHz, and 8GB of disk space.

In order to simulate slow nodes, we ran CPU, memory and disk intensive processes on one of the nodes [4]. The Ubuntu Stress package [10] was used to create the workload to slow down one of the cluster nodes. The Stress workload we used is *1000 workers* spinning on `sqrt()` to consume CPU, *50 workers* spinning on `malloc()/free()` to consume Memory and *50 workers* spinning on `write()/unlink()` to consume disk.

As for the workload benchmark, we used the TeraSort benchmark bundled with the Hadoop distribution. TeraSort benchmark is the main benchmark used for evaluating Hadoop at Yahoo! and it combines testing of both the HDFS and MapReduce layers of the Hadoop cluster. Input data of 1GB was generated and used in the evaluation phase. The job was split to 400 Map tasks and 50 Reduce tasks. All results reported here present the mean and the 95% Confidence Interval values of running the benchmark 5 times. Resource cost is calculated according to the following formula:

$$C = P \times (\sum_{N_f} T_f + \sum_{N_s} T_s) \quad (1)$$

C is the resource cost required to complete the job. P is the VM price per hour. N_f and N_s are the number of fast and slow nodes respectively. T_f and T_s are the running time of fast and slow nodes respectively. In all scenarios in our experiments, we either had one or no slow nodes, so N_s is either 0 or 1.

Experiment 1

Objective: How is Hadoop's performance affected by enabling or disabling speculative tasks? Do speculative tasks improve Hadoop's speed?

In a cloud computing environment, tasks may be slow for various reasons, including hardware degradation, software misconfiguration, etc. In this experiment, we slow down one of the five slave VMs by consuming CPU resources to simulate a "busy" node.

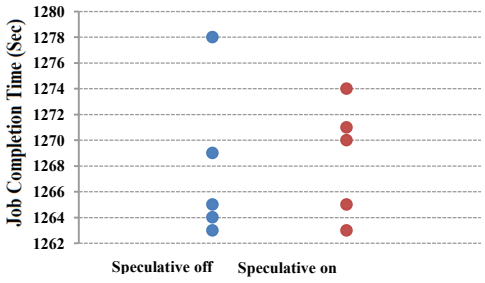

Figure 2: Time to finish the job with speculative on and off.

Speculative execution can be enabled or disabled independently for Map tasks and Reduce tasks, on a cluster-wide basis, or on a per-job basis. These configurations are related to two variables: `mapred.map.tasks.speculative.execution` and `mapred.reduce.tasks.speculative.exeuction` in the `mapred-site.xml` configuration file.

Judging from fig. 2, it is hard to tell if the speculative configuration is helpful in speeding up Hadoop. This situation could be due to several factors. First, our experiment is done with 6 small size VMs with 1GB of data to sort, so the advantage of the speculative mechanism might be hidden in such a configuration. Second, the goal of speculative execution is to reduce job execution time, but this comes at the cost of cluster efficiency. On a busy cluster, speculative execution can reduce overall throughput. However, executing redundant tasks does bring down the execution time for a single job.

Experiment 2

Objective: Does collecting (shutting down) garbage (slow) tasks help in reducing the total job completion time and resource cost?

In this experiment, we created three scenarios:

1. **Baseline Scenario:** In this scenario, we run Hadoop and do not run the Stress workload on any of the slaves. Thus, this scenario shows Hadoop's performance in the case of no slow tasks and presents the best case scenario where all tasks run as fast as they can.
2. **Slow Node Scenario:** In this scenario, we run Hadoop and run the Stress workload on one of the slaves to slow it down. Studying the effects of having multiple slow nodes is left for future research. In this scenario, we do not run the Garbage Collector and allow Hadoop to use its default speculative execution model to handle the slow tasks.
3. **Garbage Collector Scenario:** In this scenario, we run Hadoop, run the Stress workload on one of the slaves to slow it down and run the Garbage Collector to collect the slow tasks. The Garbage Collector is configured to collect the slow tasks after the job runs for 2 minutes.

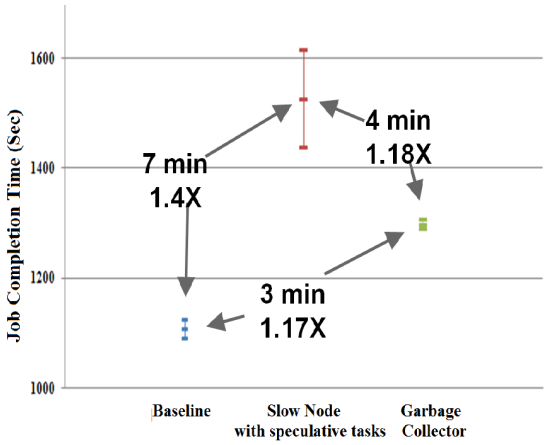

Figure 3. Job Completion Time for Baseline, Slow Node and Garbage Collector Scenarios.

From fig.3, we can see that, when dealing with slow tasks, the current Hadoop implementation (slow node with speculative tasks scenario) can cause a 40% increase in the job completion time compared to the baseline scenario. This increase was 7 minutes in our experiment representing 1.4 folds of the execution time of the baseline scenario. On the other hand, collecting (shutting down) slow tasks in the first 2 minutes (Garbage collector scenario) causes only a 16.7% increase in the job completion time - only added 3 minutes - to the baseline scenario execution time. Thus, the Garbage Collector is actually reducing the job completion time in the presence of slow node compared to the speculative tasks mechanism. With the Garbage Collector, instead of waiting for the slow tasks to complete, the Garbage Collector kills these tasks and forces Hadoop's scheduler to re-schedule them on faster nodes.

Shutting down the slow node reduces the number of VMs being used and thus reduces the total resource cost making collecting slow nodes more cost efficient than keeping them running slowly. For the job cost, as calculated with Equation 1, fig. 4 shows that using the Garbage Collector would save 40% of the cost compared to leaving the slow tasks running till the end of the job. Using the Garbage Collector keeps the cost almost the same as the baseline case in which there are no slow nodes.

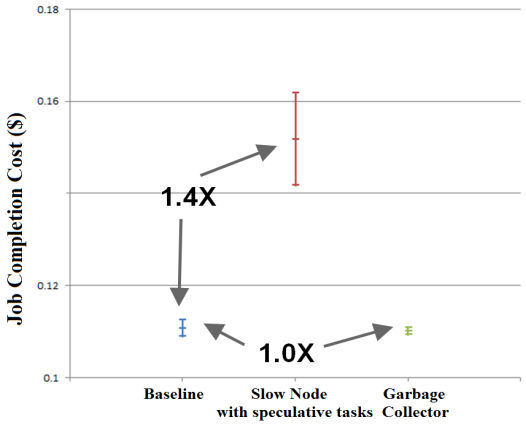

Figure 4. Job Completion Cost for Baseline, Slow Node and Garbage Collector Scenarios.

In fig. 5, we repeat experiment 2 but in a larger cluster of one master and 12 slaves using 1GB of data to sort and one slow node. As illustrated in the fig. 5, the Garbage Collector still yields better performance than leaving Hadoop to handle the slow node. Comparing the results of a 12 node cluster with that of 6 node cluster, we find that the effect of a slow node on the job completion time in the large cluster is worse than that in the small cluster. In the larger case, the effect of a slow node on the job completion time increased from 40% to 100%. The reason behind this increase is that more nodes need to wait for the slow *Map* tasks to finish so that the *Shuffle* phase can complete and the *Reduce* phase can start. In the large cluster, the effect of the Garbage Collector on the job completion time also increased and it helped to reduce the job completion time by 19% compared to 16% in the small cluster.

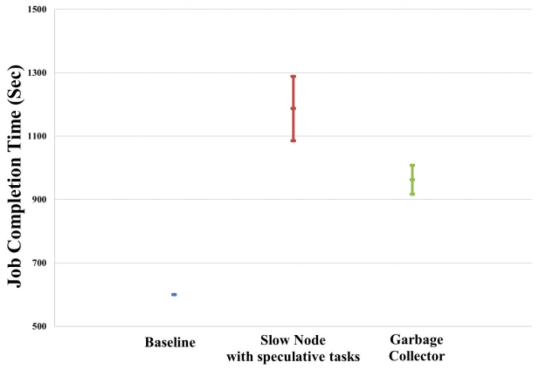

Figure 5. Job Completion Time for Baseline, Slow Node and Garbage Collector Scenarios with 12 nodes and 1 slow node.

We conclude from this experiment that collecting (shutting down) slow tasks - and thus nodes- is more efficient than leaving them running slowly till the end of the job both in terms of job completion time and job resources cost.

Experiment 3

Objective: When is it most effective to invoke the Garbage Collector to shut down slow tasks in order to minimize the job completion time and resource cost?

In this experiment, we run the Stress workload on one of the slaves to slow it down and evaluate the effectiveness of the Garbage Collector when invoked at different job completion percentages. We used the average of the job completion times from the three scenarios reported in experiment 2 as the estimated job completion time to calculate the job completion percentage in this experiment. We ran a set of scenarios, invoking the Garbage Collector from when the job was 10% completed to when it was 90% completed. The effect of invoking the Garbage Collector on the Job completion time and job resources cost compared to not invoking the Garbage Collector at all (Y=0) are reported in fig. 6 and fig. 7, respectively.

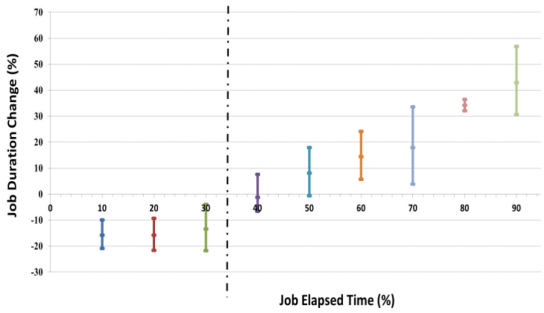

Figure 6. Impact of Garbage Collector Invocation Threshold on Job Completion Time.

In terms of the job completion time, fig. 6 illustrates that the Garbage Collector needs to be invoked before the job is 40% completed. Invoking the Garbage Collector after that caused an increase in the job completion time because at this point the Garbage Collector is collecting tasks that are almost finished and thus the generated intermediate data is lost. From the figure we can deduce that if the Garbage Collector was not invoked before the job is 40% completed, then to minimize the job completion time, it is better to keep the slow tasks running till the job is completed (100%). Otherwise, the job takes longer to finish.

In terms of the resource cost required to complete the job, fig. 7 illustrates that the Garbage Collector needs to be invoked before the job is 50% completed. Invoking the Garbage Collector after that caused an increase in the resource cost.

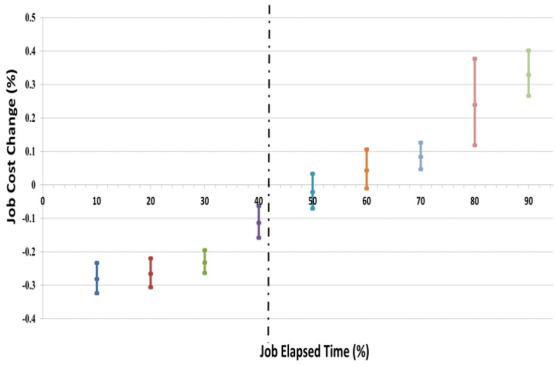

Figure 7. Impact of Garbage Collector Invocation Threshold on Job Completion Cost.

We conclude from this experiment that, in general, there is a point at which it is essential to collect (shut down) the slow tasks in order to save time. In our experiments using the TeraSort benchmark, that point is before the job is 40% completed. If that is not possible, then it is better to keep the slow tasks running till the job is completed. Similarly, in general there is another point at which the slow jobs should be collected if the main objective is to reduce the resources cost and a slower job completion can be tolerated. In our experiments that point is before the job is 50% completed, thus giving more time to do the analysis to identify slow tasks before starting to collect them. To generalize these results, we are planning on repeating this experiment using a set of different workload benchmarks to determine how the workload type affects the collection point and to deduce guidelines for selecting the collection points.

Experiment 4

Objective: How to identify garbage tasks? What is the perfect time to say that a task is being slow and what are the major factors causing a task to slow down?

Instead of monitoring individual tasks, we monitor the whole node's utilization. Node utilization presents a faster, less complex metric with available history that can be checked before even submitting the tasks. In this experiment, we consider the virtual machine CPU, Memory and Disk utilizations. To evaluate the effect of the utilization of the different components on the job completion time, we consider 4 scenarios:

1. **No Stress Workload Scenario:** In this scenario, no Stress workloads are running (i.e. no slow nodes). This scenario presents the baseline for comparison.
2. **CPU Stress Workload Scenario:** In this scenario, we run 10,000 CPU workers on one of the slaves to cause the CPU to become a bottleneck.
3. **Memory Stress Workload Scenario:** In this scenario, we run 50 Memory workers on one of the slaves to cause Memory to become a bottleneck.
4. **Disk Stress Workload Scenario:** In this scenario, we run 10,000 Disk workers on one of the slaves to cause Disk access to become a bottleneck.

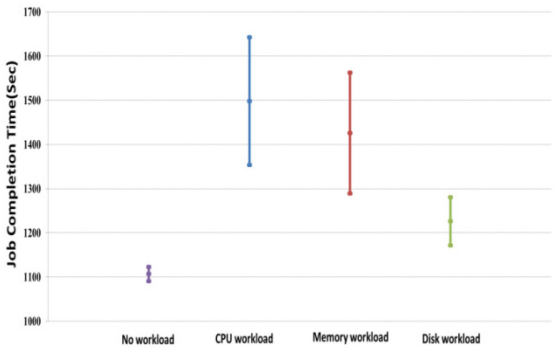

Figure 8. Effect of the different types of Stress workloads on the Job Completion Time.

In this experiment, we work on identifying the bottleneck resource for a job and then use it to determine the slow tasks. The results of this experiment, represented in fig.8, show that CPU contention had the largest effect on slowing the tasks down. CPU contention can be caused by other applications using the CPU or be due to memory thrashing.

Using the ratio between the CPU usage of the task and the VM CPU utilization can be a good indicator of a task running slow. A high ratio would indicate that the task is running as fast as possible and consuming the CPU, in this case, the garbage collector will not consider this as a slow task. On the other hand, a low ratio would indicate that the CPU time is being consumed by something other than the task, making the task run slow. In this case, the garbage collector will flag this task as a slow task and collects it. More investigation still needed in this point to determine the threshold that differentiates between high and low ratios.

The effects of CPU and memory were expected; however, what came as a surprise was that Disk contention only causes 10% increase in the job completion time. This might be a result of having a large memory in our experiments that increased the memory hit rate, thus there was no much need to load data from disk.

As a conclusion from the three last experiments, we deduce that, for our experimental scenarios, allowing Hadoop to monitor the CPU utilization ratio on the TaskTracker nodes and using a CPU utilization ratio threshold to identify garbage tasks before the job is completed with a certain percentage can indeed save time and reduce cost.

6 Related Work

Hadoop's performance is closely tied to its task stragglers. Thus, a number of studies have investigated how to minimize the effect of poor performance nodes and tasks on the overall job performance. However, the problem of finding the optimal way to identify slow tasks and efficiently migrate the processed data in a cost effective manner is still unsolved.

Hadoop assumes that cluster nodes are homogeneous and tasks are executed linearly. This assumption is then being used to decide when to active speculatively tasks for the stragglers. However, homogeneity might not be true in practice because of the use of different hardware and virtualized resources. *LATE* [4] is the first work to point out and address the short-comings of MapReduce in heterogeneous environments (virtualized data centers included). *LATE*, which stands for Longest Approximate Time to End, focuses specifically on managing slow tasks. The study showed that the current slow task

management techniques lead to poor performance in heterogeneous clusters and proposed a new robust scheduling algorithm for identifying, prioritizing, and scheduling backup copies for slow tasks. *LATE* focuses on estimate time left rather than progress rate. It is supposed to execute only tasks that will improve job response time, rather than any slow tasks. *LATE* claims to have reduced Hadoop's response time by a factor of two.

SAMR [5], Self-Adaptive MapReduce, extends LATE by calculating the progress of tasks dynamically and adapting to the continuously varying environment automatically. It deals with the Map and Reduce phases separately and specifies several new parameters for self-adaption. *Tarazu* [9] enhances the performance by implementing a communication-aware scheduling and load balancing to reduce network traffic when replicating slow tasks.

These techniques relies on scheduling backup copies of the slow tasks while keeping the slow tasks running, the Garbage Collector on the other hand, relies on shutting down the slow tasks and allowing the default Hadoop scheduler to reschedule them. By shutting down slow tasks, the slave becomes free to serve other tasks instead of being locked to a slow task. Also, by not creating backup tasks, the load (number of tasks to serve) is reduced, thus decreasing the job cost. An empirical comparison is needed to compare these different techniques and is left for future work.

The Garbage Collector still can utilize the advanced task progress calculations provided by these techniques for a better estimation of the time left for a task to complete. This can help the Garbage Collector to better decide when to be invoked to collect the slow tasks.

ISS [7] protects data generated by Map tasks (intermediate data) against node failures by replicating locally-consumed data online. *Mantri* [6] extends the *ISS* idea and provides a broader solution that replicates task output based on the probability of data loss and the recursive cost of re-computing inputs. *Mantri* is also considered the first study on a large production MapReduce cluster of thousands of servers as it was deployed by Microsoft Bing search engine. *RAFT* [8] proposed using piggy-backs checkpoints to persist intermediate results at several points in time to deal with multiple node failures. Thereby, *RAFT* can re-compute intermediate data instead of re-executing the task.

The Garbage Collector would benefit in terms of reducing the re-execution time of slow tasks, if integrated with one of these techniques to preserve the data processed before the slow node was collected. This data can then be excluded from being processed again when the slow task is re-executed.

7 Conclusions and Future Work

In this paper, we proposed a Garbage Collector to collect garbage (slow) tasks in Hadoop. We evaluated the hypothesis that collecting (shutting down) these garbage tasks can save time and reduce cost. We used Amazon EC2 to build two Hadoop virtual clusters to evaluate our hypothesis. The empirical results using the TeraSort benchmark show that collecting garbage tasks does indeed save time and reduce cost but only if the Garbage Collector is invoked before the job is 40% completed. Otherwise, it is more efficient - both in terms of time and cost- to keep these garbage tasks running till the job is completed. The results also show that a CPU utilization ratio threshold on the TaskTracker nodes is a good method to detect if tasks are going to run slowly.

The design of our initial experiments is coarse grained. More in-depth investigation with a larger set of workload benchmarks is still needed to detect accurately the appropriate time for invoking the "Garbage Collector". Our current results indicate that it is worthwhile improving Hadoop's speculative task mechanism and "Garbage Collector" might be the cure. In our experiment, we only used Hadoop clusters of 6 and 12 small size VMs. However, we are looking forward to extend our experiment on a larger cluster.

For the future work, we plan to integrate the Garbage Collector engine and monitor with the JobTracker and TaskTracker respectively. We also plan to study the effects of having multiple slow nodes per cluster on the Garbage Collector performance. Finally, we plan to combine the Garbage Collector with one of the intermediate data reservation techniques (*ISS* [7], *Mantri* [6] or *RAFT* [8]) introduced in the related work section to see if this would reduce the time and cost further.

References

[1] J. Dean and S.Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (January 2008), 107-113. 2008.

[2] T. White. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.. 2009

[3] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc.1996.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (OSDI'08). 29-42. 2008.

[5] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology* (CIT '10), 2736-2743. 2010.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*(OSDI'10), 1-16. 2010.

[7] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing* (SoCC '10). 181-192. 2010.

[8] J. Quiané-Ruiz, Ch. Pinkel, J. Schad, and J. Dittrich. RAFT at work: speeding-up mapreduce applications under task and node failures. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (SIGMOD '11). 1225-1228. 2011.

[9] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing MapReduce on heterogeneous clusters. *SIGARCH Comput. Archit. News* 40(1): 61-74. 2012.

[10] A. Waterland. 2009. Stress the deliberately simple workload generator for POSIX. Online [last visit April 2nd, 2013]: http://weather.ou.edu/~apw/projects/stress/

[11] Applications and organizations using Hadoop. Online [last visit June 22nd, 2013]: http://wiki.apache.org/hadoop/PoweredBy

[12] S. Ghemawat, H. Gobioff, and Sh. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5): 29-43. 2003.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. *In Proc. of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 1-10. 2010.