Evaluation of Linux I/O Schedulers for Big Data Workloads

Abdelmounaam Rezgui
Dept. of Computer Science & Engineering
New Mexico Tech
Socorro, NM, USA
rezgui@cs.nmt.edu

Matthew White
Dept. of Computer Science & Engineering
New Mexico Tech
Socorro, NM, USA
mwhite@nmt.edu

Sami Rezgui
Université d'Alger III
Dely Brahim
Algiers, Algeria
samirezgui@gmail.com

Zaki Malik
Dept. of Computer Science
Wayne State University
Detroit, MI, USA
zaki@wayne.edu

***Abstract*—Big data is receiving more and more attention as an increasingly large number of institutions turn to big data processing for business insights and customer personalization. Most of the research in big data has focused on how to distribute a given workload over a set of computing nodes to achieve good performance. The operating system at each node uses an I/O scheduling algorithm to retrieve disk blocks and load them into main memory. Achieving good performance in big data applications is therefore inherently dependent on the efficiency of the I/O scheduler used in the OS of the nodes. In this paper, we evaluate the impact of different Linux I/O schedulers on a number of big data workloads. The objective of the study is to determine whether specific schedulers (or specific configurations of those schedulers) are superior to others in terms of supporting big data workloads.**

Keywords—Big data, Linux I/O schedulers;

I. Introduction

Big data may be defined as both structured and unstructured data that is so large and complex that it requires a processing capacity not available using conventional database systems or traditional data processing software. Big data is often defined in terms of the "three Vs": volume, velocity (rate at which data arrives and speed at which it must be processed), and variety (heterogeneity of data types, representation, and semantic interpretation.) Big data systems are being adopted in a wide spectrum of applications. For example, they may be used to leverage user data to glean valuable business insights. Given the volumes of data that big data systems manipulate, good levels of I/O performance are crucial to the overall efficiency of big data applications. Even small improvements in performance could have a substantial impact for a company with large scale big data systems such as Google or Amazon.

Research on improving the efficiency of big data systems has focused on two major directions: (i) improving the hardware used in computers running big data workloads and (ii) improving the software that controls that hardware. Some of that research focused on maximizing the use of underutilized hardware, e.g., [1]. In [2], the authors conduct a benchmarking study of a number of popular big data systems. They conclude that different types of hardware are suited to different types of big data applications. In [3], the authors propose hardware acceleration for MapReduce as an approach to overcome the CPU bottleneck.

In parallel with this hardware-focused research, substantial software-focused research is exploring new approaches to efficiently distribute a given big data workload over a set of computing nodes to achieve good performance. The operating system at each node uses an I/O scheduling algorithm to retrieve disk blocks and load them into main memory. Achieving good performance in big data applications is therefore inherently dependent on the efficiency of the I/O scheduler used in the OS of the nodes. In this paper, we evaluate the performance of different Linux I/O schedulers using a number of big data workloads.

Most Linux distributions provide four I/O schedulers by default [4]: (i) Completely Fair Queuing (CFQ), (ii) Deadline, (iv) Noop, and (iii) Anticipatory. We evaluate the performance of these four Linux I/O schedulers when used in big data workloads. The objective of this study is to determine whether specific schedulers (or specific configurations of those schedulers) are better suited to support big data processing. If better efficiency can be achieved by selecting an appropriate scheduler, big data applications could be made faster with no change to the hardware. This, in turn, could translate into substantial cost savings.

This paper is organized as follows. In the next section, we overview the four Linux I/O schedulers. In Section III, we describe our experimental evaluation. In Section IV, we summarize the results of our experiments. In Section V, we conclude and give future research directions.

II. Linux I/O Schedulers

In this section, we review the four Linux I/O schedulers. The description is derived from the Red Hat Enterprise Linux Tuning Guide [4].

A. Completely Fair Queuing

The Completely Fair Queueing (CFQ) scheduler attempts to provide some fairness in I/O scheduling decisions amongst multiple processes. It defines three different scheduling classes: (i) real-time (RT), (ii) best-effort (BE), and (iii) idle. By default, processes are placed in the best-effort scheduling class. Any scheduled real-time I/O is always performed before best-effort or idle I/O. As a result, real-time I/Os can starve out I/Os from both the best-effort and idle classes. I/Os from processes in the idle scheduling class are only serviced when there is no other I/O pending in the system.

CFQ achieves fairness by assigning a time slice to each of the processes performing I/Os. During its time slice, a process may have (by default) up to 8 requests in flight at a time. The scheduler tries to predict whether an application will issue more I/Os in the near future. If it determines that a process is likely to issue more I/Os, then the CFQ scheduler will idle, waiting for those I/Os, even if other I/Os from other processes are waiting to be issued. This idling makes CFQ inadequate for hardware that does not suffer from a large seek penalty, e.g., solid state disks.

CFQ has several tunable parameters:

- **back_seek_max:** This tunable controls the maximum distance in KB the I/O scheduler will allow backward seeks. The default is 16 KB. Backward seeks are bad for performance but are allowable up to the distance set by this variable.
- **back_seek_penalty:** Because backward seeks are inefficient, a penalty is associated with each one. The penalty is a multiplier; for example, assume the I/O queue has two requests equidistant from the current head position (one before and one after.) If the default back seek penalty of 2 is used, the head will move forward because the request at the later position on disk is twice as close as the earlier request.
- **fifo_expire_async:** This parameter controls how long an asynchronous (buffered write) request can go unserviced. After the expiration time, a single starved async request will be moved to the dispatch list.
- **fifo_expire_sync:** This parameter controls how long synchronous requests can go unserviced.
- **group_idle:** If set, CFQ will idle on the last process issuing I/O in a control group (cgroup, [4]). This should be set to 1 when using proportional weight I/O cgroups.
- **group_isolation:** If group isolation is disabled, fairness is provided for sequential workloads only. If it is enabled, fairness is provided for both sequential and random workloads.
- **low_latency:** If low latency is enabled (set to 1), CFQ favors fairness over throughput by attempting to provide a maximum wait time of 300 ms for each process issuing I/O on a device.
- **quantum:** The quantum is the maximum number of I/Os that CFQ will send to the storage at a time, essentially limiting the device queue depth. Increasing this parameter will have a negative impact on latency.
- **slice_async:** This is the time slice allotted to each process issuing asynchronous requests.
- **slice_idle:** This parameter specifies how long CFQ should idle while waiting for further requests.
- **slice_sync:** This tunable specifies the time slice allotted to a process issuing synchronous I/Os.

B. Deadline

The Deadline scheduler aims at providing a guaranteed latency for requests. It assigns an expiration time to each device request. Once a request reaches its expiration time, it is serviced immediately, regardless of its targeted block device. For efficiency, similar requests at nearby disk locations will be serviced as well. By default, reads are given priority over writes, since applications are more likely to block on read I/Os.

Deadline has five tunable parameters:

- **read_expire:** The number of milliseconds before each read I/O request expires.
- **write_expire:** The number of milliseconds before each write I/O request expires.
- **fifo_batch:** When a request expires, it is moved to a dispatch queue for immediate servicing. These requests are moved in batches. The fifo_batch parameter specifies batch size.
- **writes_starved:** This parameter specifies how many read requests should be moved to the dispatch queue before any write requests are moved.
- **front_merges:** Sometimes a new request is contiguous with another already in the queue. The front_merges parameter specifies whether such a new request should be merged to the front or the back of the queue.

C. Noop

Noop is the simplest of the available I/O schedulers. It implements a simple first-in first-out (FIFO) scheduling algorithm and has no tunable parameters. For systems with fast storage and CPU-bound workloads, Noop can be the best I/O scheduler to use.

D. Anticipatory

The Anticipatory I/O scheduler enforces a delay after servicing an I/O request to give the application a window in which to submit another request for the next block. Anticipatory has five tunable parameters:

- **read_expire:** The number of milliseconds before each read I/O request expires. Once a request expires, it is serviced immediately. Generally, this should be half the value of write_expire.
- **write_expire:** The number of milliseconds before each write I/O request expires. Once a request expires, it is serviced immediately.
- **read_batch_expire:** The number of milliseconds that should be spent servicing a read batch of requests before servicing pending write batches. This parameter is typically set as a multiple of read_expire.

- **write_batch_expire:** The number of milliseconds that should be spent servicing a batch of write requests before servicing pending read batches.

- **antic_expire:** The number of milliseconds to wait for an application to issue another I/O request before moving on to a new request.

III. Experimental Evaluation

To evaluate the four I/O schedulers, we ran several big data workloads on Hadoop clusters built using the Global Environment for Network Innovations (GENI) [5]. GENI is an NSF-supported nationwide suite of infrastructure supporting "at scale" research in networking, distributed systems, security, and novel applications. To use GENI, a researcher can acquire computational resources from a variety of locations, connect the resources together in a custom network topology, and control the resources via interfaces such as SSH [6]. In our case, it was critical that we have complete control of the software running on the cluster's nodes and GENI allows near-total control of resource software.

A. Hadoop's Built-in Benchmarks

In order to assess the relative performance of the various I/O scheduling options, we had to have some objective standard of comparison. There are a variety of benchmarking systems available for big data. One of the benchmarks that we initially considered was BigDataBench, a benchmark suite for big data [7]. It is meant to simulate a variety of real-world workloads including search engines, social networks, and e-commerce sites. We downloaded the suite and tried to make some progress with it. However, we quickly discovered that BigDataBench requires a lot of configuration and custom code modification. As a result, we opted to use Hadoop's built-in benchmarking tools.

Apache Hadoop comes with a selection of benchmarking tools and sample programs [8]. Among these tools, there was a subset that is suitable for the purpose of our evaluation:

1) TestDFSIO

TestDFSIO is the most obvious benchmark choice for determining the efficiency of the I/O schedulers. TestDFSIO contains simple read and write tests for Hadoop's HDFS [9] [10]. Being the most I/O-intensive benchmark, it is uniquely positioned as the most direct test for I/Os. Due to replication, the write test will end up doing more work and generating more network traffic than the read test [10].

2) TeraSort

TeraSort is another benchmark for Hadoop, consisting of three phases: generation, sorting, and validation [9] [10]. TeraGen is the first step in the process, generating the data for TeraSort to sort. Due to data writing, TeraGen will have a similar I/O impact as TestDFSIO write does, but may be bottle-necked by the CPU, since the data TeraGen creates requires complex computations [10].

TeraSort sorts the output from TeraGen (a file full of 100-byte rows of data). The sorted result from TeraSort is stored on HDFS (but bypasses replication) [10].

TeraValidate reads in the output from TeraSort and ensures that all of the data is in the proper order [9] [10]. The map tasks ensure order for any individual files while the reduce tasks ensures that the individual files are ordered compared to one another.

3) NameNode Benchmark

The NameNode benchmark (NNBench) is a benchmark focused on testing the capacity of the name node in a Hadoop cluster, generating a lot of HDFS requests in order to put a high strain on the management capabilities of the name node [9].

4) MapReduce Benchmark

The MapReduce benchmark (MRBench) is a benchmark that can loop a small job many times in sequence [9].

5) Wordcount

Wordcount is not one of Hadoop's built-in benchmarks, but is found in the example map reduce programs. It reads through its input and outputs how frequently words occur in the input.

B. Cluster Configuration

We conducted two sets of experiments using two different Hadoop cluster configurations deployed on GENI:

1) Cluster I

The first cluster included one name node and three worker nodes. All nodes were co-located and had the maximum hard disk size allowable on GENI's virtual machines. The nodes were placed into a mesh network with every node connecting directly to every other node.

1) Cluster II

The second cluster included one name node and four worker nodes. The purpose of adding a fourth worker node was to add some more disk space. Here again, all nodes were co-located and had the maximum hard disk size allowable on GENI's virtual machines. The nodes were placed into a mesh network with every node connecting directly to every other node.

IV. Results

In this section, we present the results of our experiments. For the sake of brevity, we will omit some cluster-benchmark combinations. Specifically, we will first present the results related to two benchmarks run on Cluster I, namely, TeraSort and WordCount. Then, we will present the results of four benchmarks run on Cluster II, namely, TestDFSIO, TeraSort, the NameNode Benchmark, and the MapReduce Benchmark

A. Cluster I

1) TeraSort

The first benchmark we applied to Cluster I was TeraSort. We first used TeraGen to generate four different sets of data.

Each of the four data sets was generated using a different scheduler. Table I and Figure 1 give the run time of TeraGen using the four schedulers. We can see some noticeable differences in the execution times. The difference, however, was not significant. One observation is that the Anticipatory I/O scheduler performed the worst in both map time and total time.

	CFQ	Deadline	Noop	Anticip.
Map time (s)	102	109	102	113
Total time (s)	112	117	113	123

Table 1. TeraGen run time

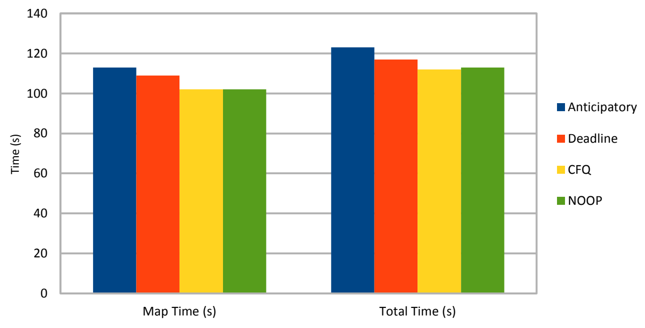

Figure 1: Performance of TeraGen using Cluster I

After the four data sets were generated, we ran TeraSort a total of 16 times – one time per data set per scheduler. We then took the average performance of each scheduler across all four data sets. Figure 2 shows the performance of TeraSort in this scenario.

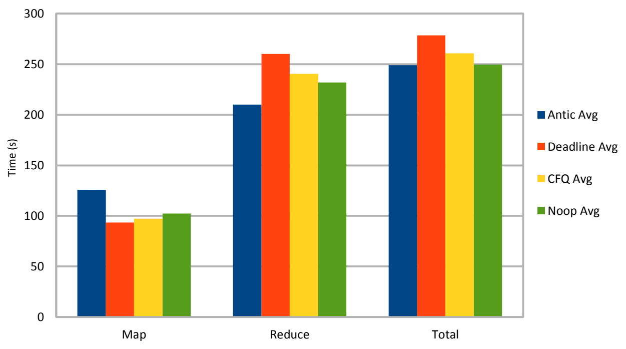

Figure 2. Performance of TeraSort on Cluster I

These runs showed some difference among the schedulers with the Anticipatory scheduler leading the way with the fastest total time. In this case, the Deadline scheduler was the slowest among the four schedulers.

2) WordCount

To conduct the WordCount experiment, we downloaded a large data set from Wikipedia – the set of current revisions of all articles that were available on February 13, 2014. Uncompressed, the size of this data set was approximately 44 gigabytes. Unfortunately, we soon found that there was no way the complete data set could be used. The job was so large that, invariably, some errors would occur during execution that would end up resetting the reduction step (e. g., a job might go from 78% reduced back down to 0% reduced in an instant). The jobs would eventually succeed despite the errors but the errors ensured that any run time statistics we got from the job would not be reflective of the scheduler's performance.

With these issues in mind, we split the Wikipedia data set up into 4 gigabyte-sized-chunks. We used three of these chunks running each scheduler on each of the chunks for a total of 12 WordCount runs. Figure 3 shows the performance of WordCount using the four schedulers on Cluster I. Here, we see similar results to those found in TeraSort. The Anticipatory scheduler is no longer the fastest since the Noop scheduler beats it by a small margin but both are relatively close. The Deadline scheduler was the slowest as in the TeraSort experiment.

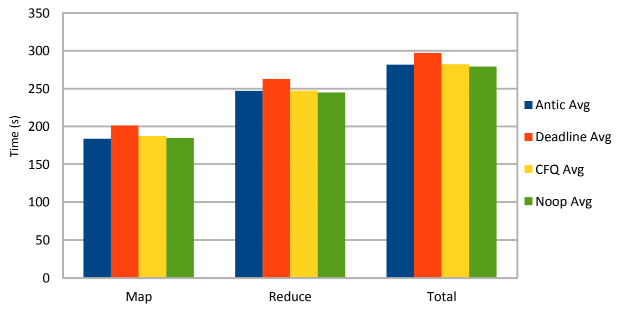

Figure 3. Performance of WordCount on Cluster I

B. *Cluster II*

We used Cluster II to conduct more extensive experiments. In particular, we used it to run a greater variety of tests and with a greater number of runs to average. Outliers (e. g., a throughput of 360 Mb/s when the typical is 36 Mb/s) were removed from consideration.

1) TestDFSIO

The first benchmark we ran on Cluster II was TestDFSIO. Each write job was set to generate 1 gigabyte of data and each read job was set to read in that same volume of data. On every scheduler, we ran write and read 5 times, for a total of 20 write jobs and 20 read jobs. Figure 4 shows the performance of TestDFSIO on Cluster II. The results show nearly identical write capacity across the board. With reads, we got the opposite of what we would expect based on our experiments on Cluster I. In this case, the Deadline scheduler had the highest throughput and the shortest execution time.

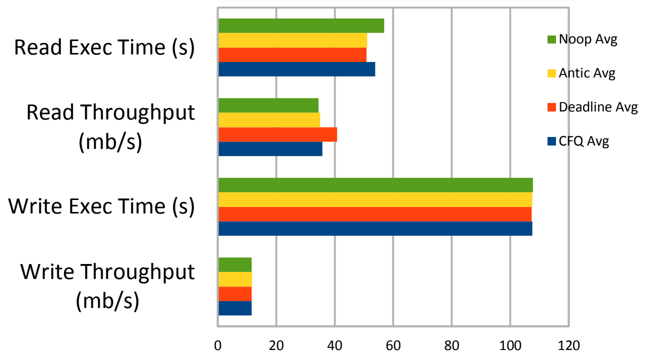

Figure 4. Performance of TestDFSIO on Cluster II

2) TeraSort

In this experiment, we repeated the TeraSort benchmark but left off the TeraGen statistics. Rather than running TeraGen repeatedly and having a variety of input (as we did on Cluster I), we only generated one set of 1 gigabyte of data and ran TeraSort against it for every test. This was to increase test consistency. In addition to TeraSort, we also kept track of TeraValidate (which we omitted in the experiments on Cluster I). For consistency, we only ran TeraValidate on a single set of TeraSort output. TeraSort and TeraValidate were run 5 times for each scheduler. Figure 5 gives the results of this set of experiments.

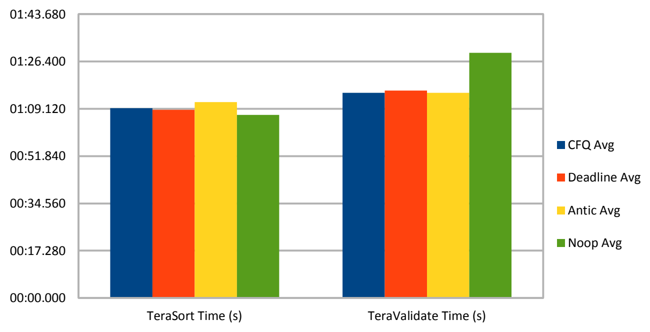

Figure 5: Performance of TeraSort and TeraValidate on Cluster II

In this set of experiments, the results were not as clear-cut as they seemed to be when using Cluster I. The fastest scheduler on this set of TeraSort runs turned out to be Noop. Conversely, on TeraValidate, Noop was the slowest by a wide margin.

3) NameNode Benchmark

We ran NNBench on each scheduler three times for a total of 12 runs. Figure 6 shows the results of these experiments. There was barely any discernible difference between the four schedulers for this benchmark. CFQ and Anticipatory did the worst while Noop and Deadline did the best. However, the averages were all within the same four seconds.

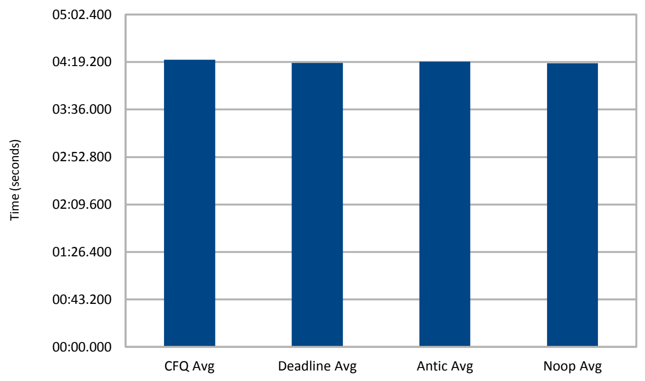

Figure 6: Performance of the four I/O schedulers with NNBench (Cluster II)

4) MapReduce Benchmark

In this set of experiments, we ran the same job 50 times for each scheduler and collected the data for all the runs. However, the percent difference was not large enough to be significant, with the averages all being within 1% of each other. Figure 7 shows the total execution time and Figure 8 shows the average execution time for the individual jobs.

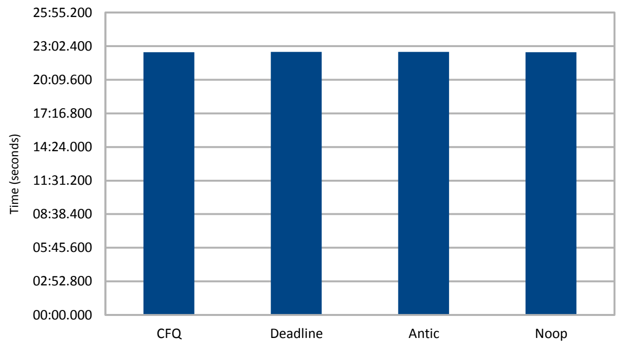

Figure 7. Total time for the MapReduce Benchmark (Cluster II)

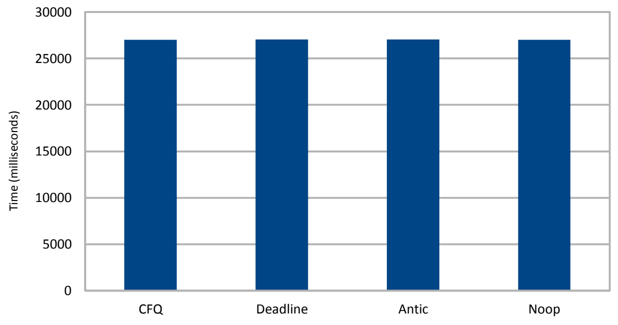

Figure 8. Average time for individual jobs in the MapReduce Benchmark (Cluster II)

5) TestDFSIO Read Revisited

After having tried all of the above benchmarks, we found that the largest percent difference seemed to lie in the TestDFSIO read benchmark. We created a new set of data with TestDFSIO write. This time the data was 10 gigabytes in size (10 times larger than before). We once again ran TestDFSIO write on every scheduler five times.

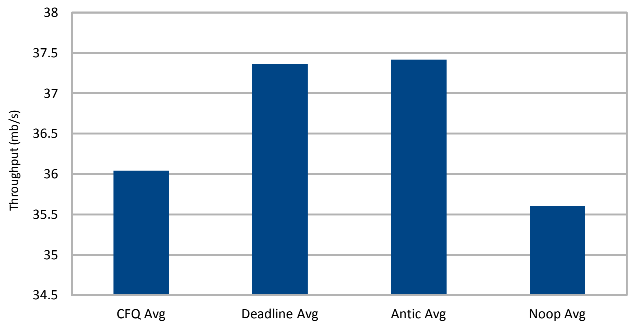

Figure 9. Throughput for TestDFSIO read on 10 GB

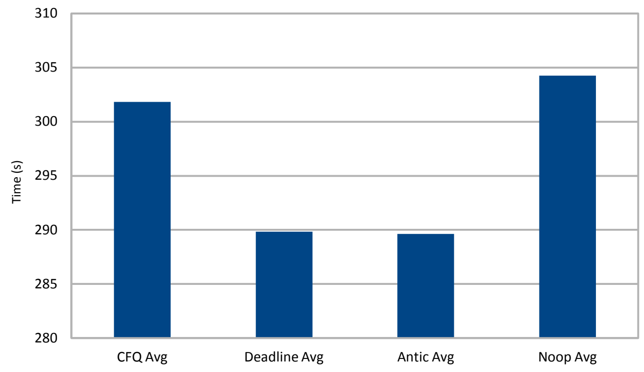

Figure 10. Execution time for TestDFSIO read on 10 GB

Figure 9 shows the throughput for TestDFSIO read on 10 GB. Figure 10 shows the execution time for TestDFSIO read on 10 GB. Here, we see a more pronounced difference between schedulers. In particular, Deadline and Anticipatory both outperform CFQ and Noop by a decent margin.

We calculated the percentage differences between the other schedulers and Anticipatory, which was performing the best. These percentage differences are given in Figure 11.

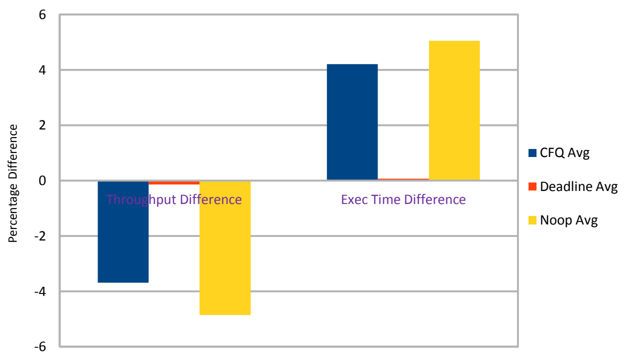

Figure 11. Percentage difference between Anticipatory and the other schedulers - TestDFSIO Read 10GB

6) Combined Results

To better understand the results of our experiments, we put all the average values (of every benchmark result) into a table and calculated the percent difference between each scheduler's average and the overall average. We then took the overall average of the percentages to get a clear view of how each scheduler performed in relation to one another. A higher percentage indicates better performance. Figure 12 and Figure 13 give the average percent difference for Cluster I and Cluster II respectively.

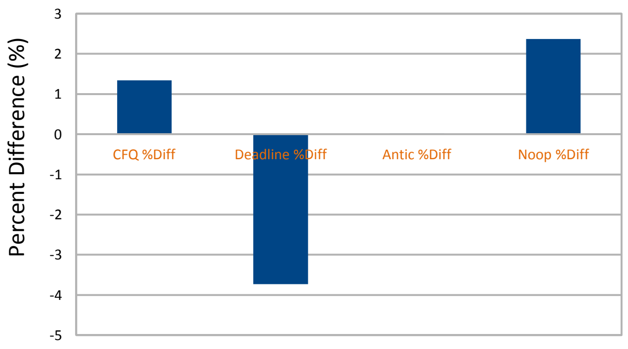

Figure 12. Average percent difference for Cluster I

As we can see, on Cluster I (Figure 12), Noop was the best performer, followed by CFQ. Anticipatory straddled the average while Deadline was the worst.

For Cluster II (Figure 13), we see a result that is almost the exact opposite of the result obtained using Cluster I. We ascend in the same order that Cluster I descended: Noop, CFQ, Anticipatory, and Deadline.

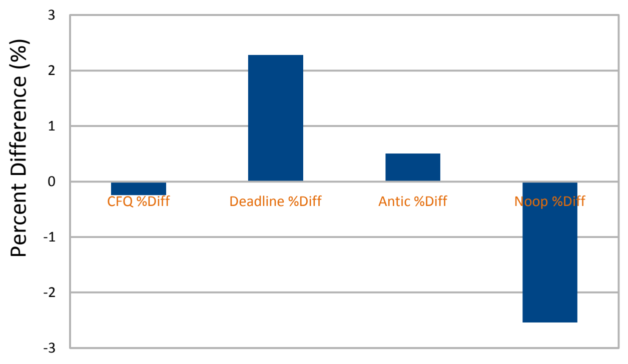

Figure 13: Average percent difference for Cluster II

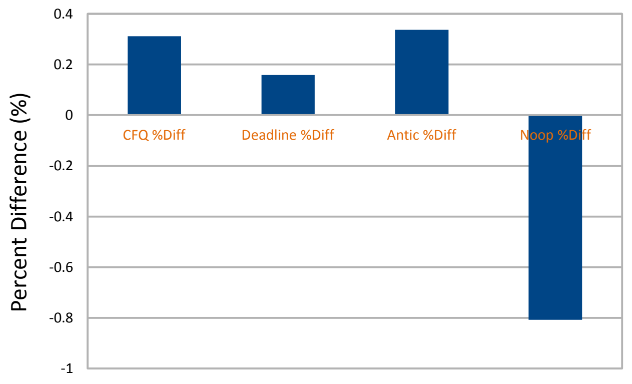

Figure 14. Average percent difference overall (both clusters)

For the overall average (Figure 14), the results on the two clusters more or less canceled each other out, dropping the largest difference to less than 1%. Anticipatory is the best performer overall, but with a performance improvement of less than 1%. Based on these results, we conclude that it may not be worth changing the I/O scheduler at all.

C. *Scheduler Parameters*

Since switching between the various scheduler choices did not seem to make much of a difference, we decided to see what would happen when we varied some of the schedulers' tunable parameters. For this, we used the TestDFSIO read benchmark on the same 10GB of data that we generated in Section IV.B.5.

It should be noted that the values quoted here were not averaged like the results in the earlier experiment. It takes 5 minutes (300 seconds) to do a single TestDFSIO read on 10 gigabytes. Therefore, it would take 25 minutes to get the average of five runs for a single parameter configuration. Rather than spend the time to get a single parameter averaged, we selected promising parameters and tried a very large range of values to see if any apparent difference appeared, the idea being that extreme parameter values should show some kind of noticeable difference in a single run.

1) CFQ

a) low_latency

By default, low latency is turned on. This places fairness above throughput [4]. We turned the setting off and ran TestDFSIO read twice. We found no appreciable difference. The throughput fell within the 36.6-37.1 Mb/s range, which is totally ordinary.

b) quantum

This effectively limits the device queue depth, controlling the number of I/Os sent to storage at a time [4]. Changing the quantum had little effect. The default value is 4. We tried values of 1, 4, 32, and 128, all with no apparent difference.

2) Deadline

a) read_expire

This is the primary parameter for the Deadline scheduler [4]. It sets the time before each read request expires and is therefore directly applicable to the TestDFSIO read benchmark (write_expire would only be worth exploring if read_expire had an effect.) We ran the benchmark with read_expire values of 8 (the minimum), 500 (the default), 1000, 2000, and 4000. There was no noticeable difference.

3) Anticipatory

a) antic_expire

This is the amount of time to wait for another I/O request before moving on to a new one [4]. The default value is 4 (and the OS will not let you set it lower). We tried values of 4, 8, 32, 64, and 256, all without apparent changes in throughput.

V. Conclusion

We conducted an experimental evaluation of four Linux I/O schedulers to assess their suitability for big data workloads. Our experiments demonstrate little differences amongst the four schedulers. Moreover, for the configurations we used, there was no appreciable difference in performance when changing the key tunable parameters of each scheduler. This work gave us valuable insights useful in our ongoing research towards developing efficient I/O schedulers for big data workloads.

Acknowledgment

This work was supported, in part, through a grant from NM Space Grant Consortium (NMSGC) and NM NASA Experimental Program to Stimulate Competitive Research (NM EPSCoR).

References

[1] Y. Zhai, E. Mbarushimana, W. Li, J. Zhang and Y. Guo, "Lit: A High Performance Massive Data Computing Framework Based on CPU/GPU Cluster," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, 2013.

[2] J. Quan, Y. Shi, M. Zhau and Y. W., "The Implications from Benchmarking Three Big Data Systems," in *IEEE*

International Conference on Big Data, 2013.

[3] T. Honjo and K. Oikawa, " Hardware Acceleration of Hadoop MapReduce," in *IEEE International Conference on Big Data*, 2013.

[4] Red Hat Inc., "Red Hat Enterprise Linux 6, Performance Tuning Guide," Technical White Paper.

[5] GENI, "https://portal.geni.net".

[6] GENI, "About GENI," http://www.geni.net/?page_id=2.

[7] Chinese Academy of Science, "BigDataBench," http://prof.ict.ac.cn/BigDataBench.

[8] Apache, "Hadoop," http://hadoop.apache.org.

[9] M. G. Noll, "Benchmarking and Stress Testing an Hadoop Cluster with TeraSort, TestDFSIO & Co.," http://www.michael-noll.com, 2011.

[10] VMware, "A Benchmarking Case Study of Virtualized Hadoop Performance on VMware vSphere 5," 2011.

[11] J. J. Quan, Y. Shi, M. Zhau and W. Yang, "The Implications from Benchmarking Three Big Data Systems," in *IEEE International Conference on Big Data*, 2013.