Workload-dependent Dynamic Disk I/O Scheduler Selection for Throughput Optimization

Darwin M. Bautista, Jermaine Luis M. Agnasin, Mark Earvin V. Alba, Thomas Neil P. Balauag
Electronics and Electrical Engineering Institute, University of the Philippines Diliman

Abstract—As processors continue to improve in performance, mass storage devices lag behind, becoming the performance bottleneck of the overall system performance. This study aims to implement a dynamic I/O scheduler switcher that maximizes throughput of disks by selecting the appropriate I/O scheduler for the current workload. The I/O scheduler switcher analyzes the current workload by computing for the statistics of the disk. It would then categorize the current workload to either of the four: random read, random write, sequential read, sequential write. And based on that, it would switch to the appropriate I/O scheduler that maximizes the throughput of the disk. Generally, the I/O scheduler switcher has achieved in total a better performance compared to the other schedulers in all of the workloads. The I/O scheduler switcher has a balanced throughput throughout the workloads tested upon it. As compared to other schedulers, all of them performed poorly at some point in the workload tested upon them.

Index Terms—I/O scheduling, workload characterization

I. Introduction

COMPUTER hardware has been generally improving over the past decades. However, I/O-related hardware has not been able to cope with the advances in CPU technology. In a computer, the I/O system is often the performance bottleneck. Thus, being able to utilize the full capacity of a computer's I/O hardware is the key to achieving optimum system performance.

Most Linux users run different types of programs. Some use their computers as web servers or routers and as desktop workstations as well. Other users also run desktop applications but leave their computers while compiling a kernel or a big application. In other words, workload characteristics of computers vary from time to time. With that said, kernels optimized for a specific workload would be suboptimal. Tuning the kernel on-the-fly is what modern users would need.

The I/O scheduler is the part of the kernel that decides which request (read or write) should be performed next. The Linux kernel has a choice of four schedulers, but most distributions use the default Completely Fair Queuing (CFQ) scheduler for their desktop kernels. It has been the default in Linux since version 2.6.18. It is generally a recommendable all-round scheduler which offers suitable performance in sharing drive access between all the different processes that are trying to access the drive at the same time.

Some common I/O scheduler goals are:

- To minimize time wasted by hard disk seeks.
- To prioritize a certain process' I/O requests.
- To give a share of the disk bandwidth to each running process.
- To guarantee that certain requests will be serviced before a particular deadline.

Since no single I/O scheduler can be optimal for all types of workloads, changing the I/O scheduler to suit the current workload would yield better overall system performance.

We aim to develop a dynamic I/O scheduler switcher based on defined workloads. The I/O scheduler selected by the dynamic switcher should deliver the best performance, in terms of throughput, for the current workload.

II. Related Work

This chapter provides a review of prior work related to this study. Both prior research deal with I/O optimization via automatic I/O scheduler selection based on statistics.

A. Automatic I/O Scheduler Selection

Automatic and Dynamic I/O scheduler selection algortihm, called ADIO, automates scheduler selection based on system characteristics and current workload [2].

The main criteria for this proposed selection is bounded latency and maximum throughput. ADIO ensures that the maximum request latency is bounded. If it is, it prioritizes throughput. This optimization is done using feedback-based control, which is called Dynamic and Adaptive Scheduler Selector (DASS).

The DASS consists of two components: a request statistics collection monitor and an I/O scheduler selection controller. The request statistics collection monitor collects the maximum latency for read and write requests and the number of read and write requests serviced . The statistics is recorded and is then used by the controller to determine which scheduler should be active to accommodate latency and bandwidth guarantees. The controller does this every adaptation period W. In the event of process deadline expiration, the controller switches from CFQ to deadline. However, to prevent frequent switching due to I/O burst, a switch is triggered only if at least two process did not meet their deadlines. For deadline to CFQ switch, the requirement is for the maximum latency to be below 50% of the upper bound.

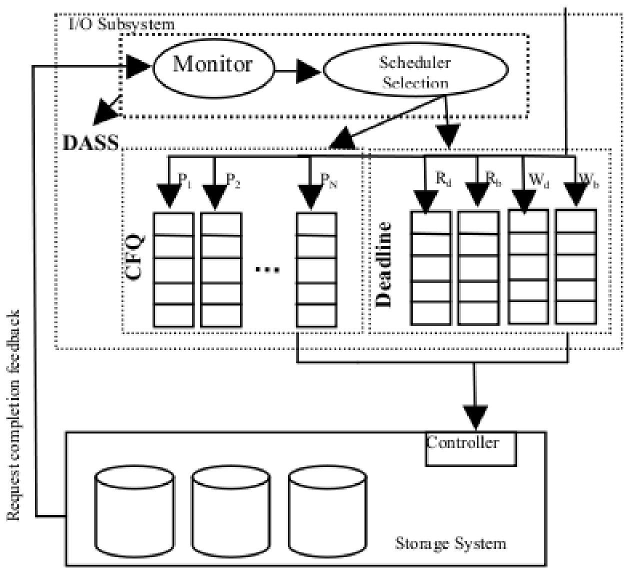


Fig. 1. ADIO Architecture

Their experimental results using tiobench showed that DASS enforces latency bounds, while providing better utilization than the default scheduler, CFQ.

B. Dynamic Disk Scheduling Algorithm

The I/O Analyzer (IOAZ) [5] attempts to examine the pattern of I/O requests in real-time. Based on the pattern examined and the decision matrices, the IOAZ appropriately changes the I/O scheduler in use.

IOAZ saves information of I/O requests from hooks placed within the Disk Request Queuing System to its internal Activity Store. An I/O Analyzer Process periodically analyzes data collected during the last period of time from the Activity Store and purges this data when it is no longer needed. This anlysis first identifies the pattern in disk accesses by comparing it with emprically derived thresholds. And then, it determines the best I/O scheduler for that pattern based on decision matrices derived from their experiments. The IOAZ changes the I/O scheduler if needed.

They defined the patterns as sequential read, sequential write, random read, and random write. Their indicator of a pattern being sequential or random is the number of I/O merge requests, which pools multiple requests for adjacent disk blocks into a single request for multiple blocks. They argue that large number of I/O merged requests is highly indicative of sequential activity and vice versa.

They used MySQL's sql-bench to test the IOAZ's performance. They also conducted disk zeroing experiment, wherein one gigabyte partition was overwritten with zeroes in a sequential fashion in a single pass. In both tests, IOAZ performed better compared to other I/O scheduler.

Lastly, a test using IOZone was carried out. The IOAZ outperformed the Deadline and CFQ scheduler, but had

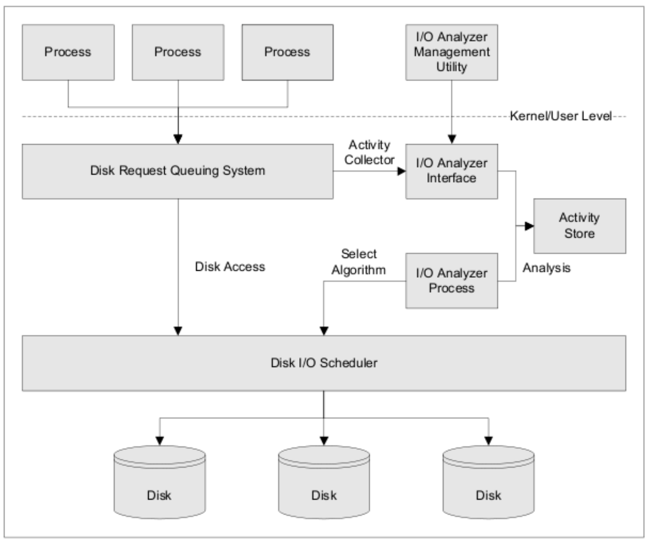


Fig. 2. IOAZ Architecture

difficulty bettering AS in experiments with many processes or FCFS in experiments with a single process. They found out that it was because of the overhead when the IOAZ switches scheduler.

C. Self-Learning Disk Scheduling

The self-learning scheduler characterizes I/O workloads by a number of essential attributes, classifies them at runtime, and makes the best I/O scheduling decision. The goal of the self-learning scheduler is to have the maximum performance (highest throughput or the shortest response time), to make a fast decision on which I/O Scheduler to be applied (low overhead), to accurately classify the current workload and to guarantee fairness to all processes[7].

The self-learning scheduling scheme introduces a new type of intelligent disk I/O schedulers, self-learning schedulers, which can learn about the storage system, train themselves automatically, adapt to various types of workloads, and make optimal scheduling decisions. There are four self-learning scheduling schemes, namely, Change-sensing Round-Robin, Feedback Learning, Per-request Learning, and Two-layer Learning.

The Change-sensing Round-Robin algorithm invokes all schedulers in a round-robin fashion for a short time. Then it logs all performance data such as response time and throughput into the log database (data gathering), compares the performance, and selects the best scheduler. However workloads and system configurations may change and frequent switching of the scheduler may impose a high overhead, so the data must be gathered only when a significant change of the workload is detected or when a significantly deteriorated system performance is observed.

The Feedback Learning Algorithm corrects errors in learning models and provides better adaptivity. If a new disk scheduling policy is activated and decreased performance is continuously observed, the feedback mechanism can force the system to switch back to the old disk scheduling policy and self-correct the classification and decision model. Moreover all the performance data are logged into the database, a further comprehensive analysis can be done with real-world data. Therefore, the learning and selection model becomes more accurate.

In Per-request Learning, the self-learning scheduler makes scheduling decisions at the request level instead of the workload level. The decision is based on the analysis of the individual request instead of the workload. They estimate the response time for each request in the waiting queue and schedule the request with the shortest estimated response time.

The Two-layer Learning algorithm combines workload-level and request-level learning algorithms and employs feedback mechanisms. One can implement a self learning core that consists of several regular I/O schedulers and one self-learning scheduler, the Per-request Learning Algorithm. This incorporates Feedback Learning Algorithm and Per-request Learning Algorithm into a two-layer self-learning scheduling scheme.

Machine learning techniques are effectively used in self-learning disk schedulers to automate the scheduling policy selection and optimization processes. The self-learning scheduler automatically creates I/O performance models, gathers system workload information, does both offline and online analysis, and fits into the operating system kernel.

In the performed experiments, they used the kernel I/O schedulers of Linux 2.6.13 and feed the system with real-world and synthetic workloads to collect performance data. They also compare three configuration modes of the self-learning scheduler: online, offline, and combined configurations. They evaluate the performance and overhead for both real-world applications and simulated scenarios. The results show that the Two-layer Learning Scheme has the best performance [7].

III. Methodology

A. Workload Characterization

We define two types of workloads based on I/O access patterns. The first type is the Sequential I/O workload. In Sequential I/O, the physical disks spend most of their time scanning a range of data clustered together in the same part of the disk. This type of access pattern results in high data transfer per I/O request (or request size). The Sequential I/O workload is exhibited mainly by file servers and streaming media servers [3].

The other type is the Random I/O workload. In this type of workload, the physical disks spend a measurable percentage of time seeking data from various different parts of the disk for read or write purposes. Thus, this type of access pattern results in relatively smaller request sizes, compared to Sequential I/O. Random I/O workload is often displayed by database servers, web servers, and mail servers [3]. Since desktop workstations tend to be used for various tasks, desktop workload can be considered as either Sequential or Random.

Since I/O can never be purely sequential or random for any given time, we define Sequential I/O Ratio (SR) as the measure of how much of the I/O requests is sequential in nature. The SR can be obtained by getting the ratio of the average request size to the peak average request size. A purely random I/O would have an SR close to zero. On the other hand, a purely sequential I/O would have an SR equal to one. Since disks vary from one system to another, we consider the I/O pattern which exhibits the highest average request size (peak average request size) to be the purely sequential I/O ($SR = 1$) for a particular system.

The SR will be used for determining whether an I/O pattern is sequential or random. That is, if the SR is lower than a specified value, say 0.75, then the I/O pattern will be considered random. Otherwise, if the SR is greater than or equal to that value, the I/O pattern will be considered sequential. We call that value the decision point. Based on our tests, a lower decision point results in a more responsive decision-making, albeit susceptible to I/O bursts. On the other hand, a higher decision point results in a slightly less responsive decision-making while being resilient to I/O bursts. For our purposes, we used a decision point of 0.75 to provide a good balance between responsiveness and I/O burst resilience.

B. Block Layer Statistics

Linux provides I/O statistics per disk. In recent 2.6 kernels, the I/O statistics are exported into userspace via sysfs. It can be accessed via /sys/block/<device>/stat. The I/O statistics currently available at the block layer include read I/Os, read merges, read sectors, read ticks, write I/Os, write merges, write sectors, write ticks, in_flight, io_ticks and time_in_queue.

As stated earlier, the two workloads, Sequential I/O and Random I/O, can be differentiated from each other by looking at the I/O request size (ratio of data transferred to the number of I/O requests). Thus, the statistics on the number of I/O requests processed (read I/Os and write I/Os) and the number of sectors accessed (read sectors and write sectors) can be used for determining the current workload. Note that the statistics on number of sectors accessed is independent on the hardware and filesystem used because it is based on the standard 512-byte sector in UNIX. As such, it can be readily used for this project's purposes. Also, for the statistics to be useful, minimal filesystem fragmentation is assumed.

C. I/O Scheduler Switcher Module

To minimize the in-kernel changes and to ease development, the scheduler switcher was implemented as a kernel module. To be able to switch I/O schedulers from within the kernel module, the stock Ubuntu 2.6.32 kernel was slightly modified. We added a new function, elv_switch(), which is based on elv_iosched_store(), to block/elevator.c for changing the I/O scheduler. This function is made available for use in kernel modules using EXPORT_SYMBOL().

We patterned the statistics processing with the CPU load computation of Linux. The CALC_LOAD() macro, which is used to calculate CPU load, employs exponential moving average. The exponential moving average is a type of finite impulse response filter that applies weighting factor that decreases exponentially. The weighting for each older data point decreases exponentially, never reaching zero.

In our implementation, we used a 1-minute time window so that the output of CALC_LOAD() stays responsive yet resilient against I/O bursts. Based on prior research [7], the 1-minute window is the optimum balance between responsiveness and resilience to I/O bursts.

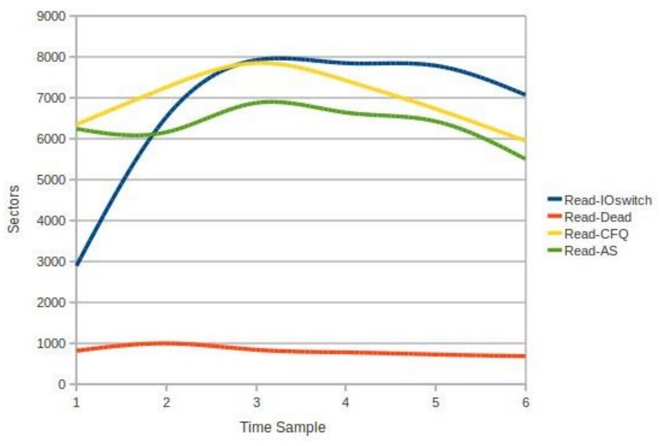
On module load, the switcher spawns a kthread that would look at the I/O statistics periodically, depending on the specified sampling period at compile-time (currently 15 seconds). The workload is first categorized as either being Read or Write. Once the dominant operation is determined, i.e. either Read or Write, the switcher then computes for the current request size based on the dominant operation's statistics. The average request size (1-minute window) is then computed using the current request size. Then, the SR is computed using the average request size and the peak average request size. Finally, the workload is determined by comparing the SR to the decision point, which is specified at compile-time (currently 0.75). Once the current workload is determined, it would then select the appropriate I/O scheduler. Based on related research [4], we have determined that the best I/O scheduler for Sequential I/O is Anticipatory while the best for Random I/O is CFQ for single-disk systems.

IV. Results and Analysis

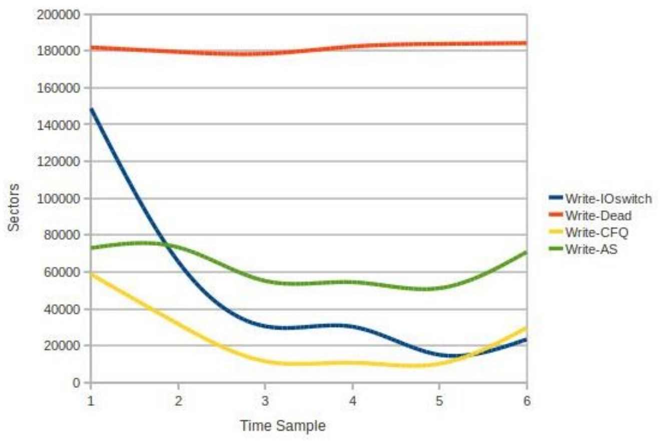
We conducted all our experiments on an Intel Core i5 650 (dual-core processor 3.20GHz) with 500GB 7200 rpm Western Digital hard disk. We devised a module that prints in /var/log/messages the throughput of the disk being monitored. This module is used for testing only; it is not part of the system. The throughput measures the sectors written to per second and the sectors read from per second.

With the module described above, we conducted a series of test using the fio benchmarking tool [8]. The test is composed of six fio files namely: file-server.fio, oltp.fio, rand-read.fio, rand-write.fio, seq-read.fio and seq-write.fio. These fio files determine the workloads we tested our system with. These six workloads were run successively and repeated five times. The average of the five runs at a sampling period of 5 seconds are shown in Figures 3 to 8. We presented the data for each workload in separate plots due to horizontal space restrictions.

As seen in the graphs, the I/O scheduler switcher has achieved in total a better performance compared to the other schedulers in all of the workloads. The I/O scheduler switcher has a balanced throughput throughout the workloads tested upon it. As compared to other schedulers, all of them performed poorly at some point in the workload tested upon them.



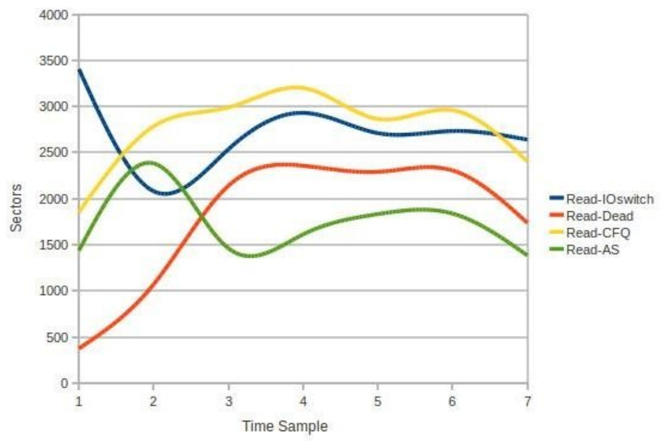
(a) Read Statistics



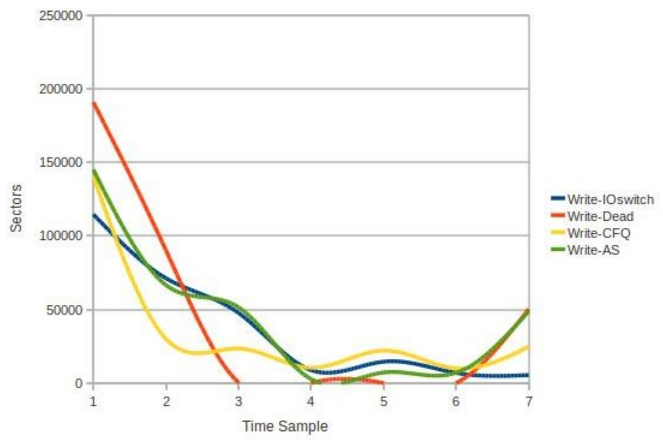
(b) Write Statistics

Fig. 3. File Server Workload

Also, we can see that the I/O scheduler switcher follows the form of the better scheduler between the Anticipatory Scheduler and the CFQ Scheduler during a certain workload. This only proves that the I/O scheduler switcher successfully mimicked the better of the two depending on the workload.

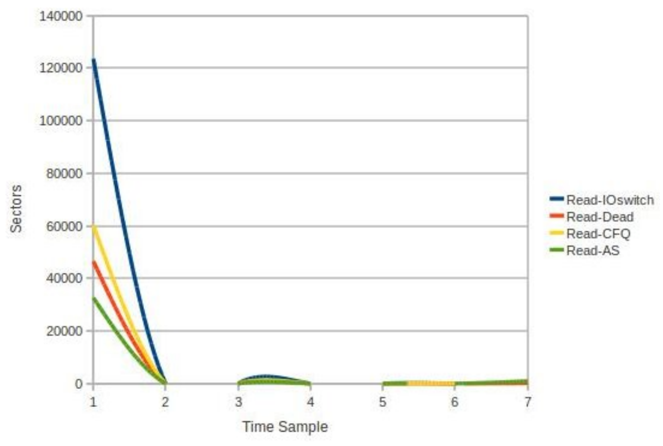


(a) Read Statistics

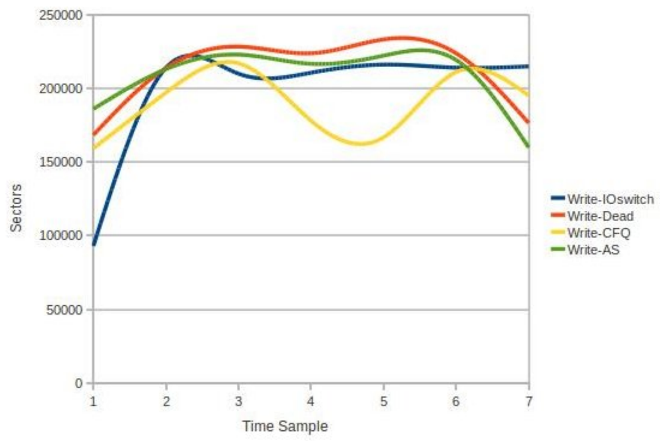


(b) Write Statistics

Fig. 4. OLTP Workload



(a) Read Statistics



(b) Write Statistics

Fig. 8. Sequential Write Workload

V. Conclusions and Recommendations

In this paper, we were able to design and implement a dynamic disk I/O scheduler in a stock Ubuntu 2.6.32 kernel. With its simplistic design of computing the statistics, the dynamic disk I/O scheduler has a small memory footprint. But it still can determine succesfully the current workload based on tests. Also, we were able to determine that the decision point of the system should be high enough to stay responsive and resilient against I/O bursts.

Our focus was to maximize the throughput of the disk. This was achieved by switching to Anticipatory, when the workload is sequential in nature, and to CFQ, otherwise. Based on the results above, the dynamic disk I/O scheduler was able to achieve a better performance throughout the workloads tested upon it.

Further experimentation is necessary to determine the best scheduling algorithm for a specific workload. Validation of the previous paper [4] should be done to justify the use of certain scheduling algorithm for a specific workload. This will result to a more reliable selection of the scheduling algorithms, capable of delivering improved performance.

For improved overall performance, fine tuning of scheduling algorithm should be sought after. In our I/O scheduler switcher, we used default parameters for the schedulers used, resulting to suboptimal performance compared to finely tuned schedulers.

References

[1] C.H Lunde, H. Espeland, H. Stensland, A. Petlund, and P. Halvorsen. “Improving Disk I/O Performance on Linux,” in Proc. Linux-Kongress and OpenSolaris Developer Conference, 2009, pp. 61-70.

[2] S. Seelam, J. S. Babu, and P. Teller. “Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization,” in Proc. Workshop on Operating System Interference in High Performance Applications, 2005.

[3] D. Hoch. “Extreme Linux Performance Monitoring Part II.” Internet: http://www.ufsdump.org/papers/io-tuning.pdf, Jan. 15, 2007 [Sep. 16, 2010].

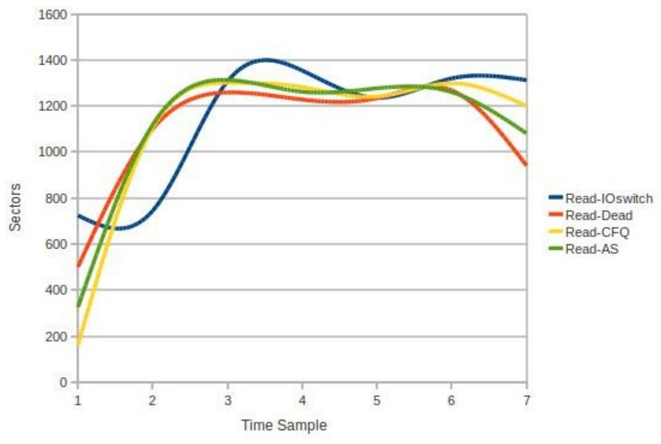
[4] S. Pratt and D. Heger. “Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers,” in Proc. Ottawa Linux Symposium, 2004, pp. 425-448.

[5] D. L. Martens and M. J. Katchabaw. “Optimizing System Performance through Dynamic Disk Scheduling Algorithm Selection,” in Proc. WSEAS Trans. Information Science and Applications, 2006.

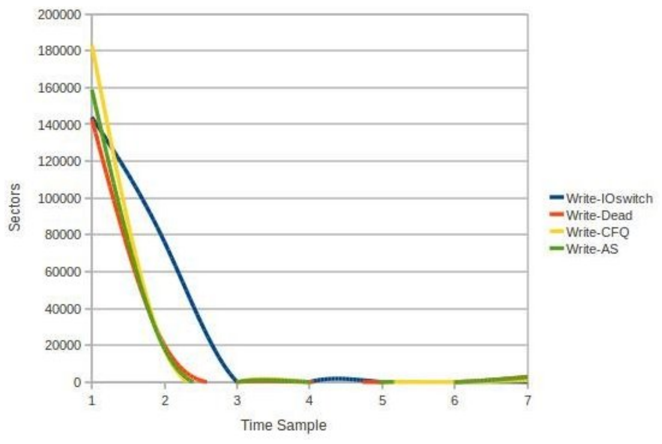
[6] K. Brandt. “Interpreting iostat Output.” Internet: http://blog.serverfault.com/post/777852755/interpreting-iostat-output, Jul. 6, 2010 [Sep. 16, 2010].

[7] Y. Zhang and B. Bhargava. “Self-learning Disk Scheduling,” in Proc. IEEE Transactions on Knowledge and Data Engineering, 2009, pp. 50-65.

[8] B. Martin. “Inspecting disk IO performance with fio.” Internet: http://www.linux.com/archive/feature/131063, Apr. 9, 2008 [Oct 8, 2010].

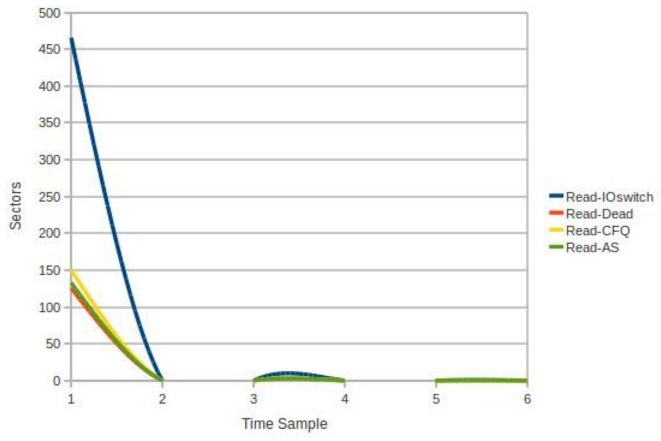


(a) Read Statistics

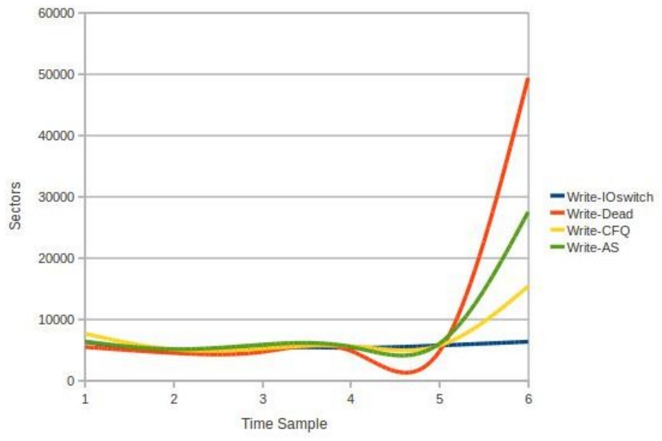


(b) Write Statistics

Fig. 5. Random Read Workload

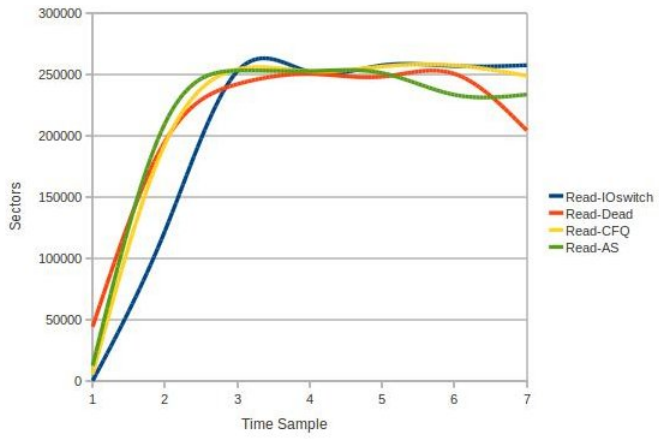


(a) Read Statistics

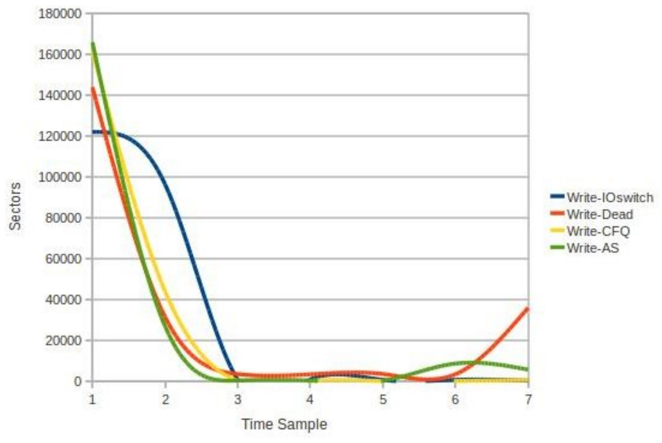


(b) Write Statistics

Fig. 6. Random Write Workload



(a) Read Statistics



(b) Write Statistics

Fig. 7. Sequential Read Workload