View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

A NEW I/O SCHEDULER FOR SOLID STATE DEVICES

A Thesis

by

MARCUS PAUL DUNN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2009

Major Subject: Computer Engineering

A NEW I/O SCHEDULER FOR SOLID STATE DEVICES

A Thesis

by

MARCUS PAUL DUNN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Narasimha Annapareddy
Committee Members,	Sunil P. Khatri
	Riccardo Bettati
Head of Department,	Costas N. Georghiades

August 2009

Major Subject: Computer Engineering

ABSTRACT

A New I/O Scheduler for Solid State Devices. (August 2009)

Marcus Paul Dunn, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Narasimha Annapareddy

Since the emergence of solid state devices onto the storage scene, improvements in capacity and price have brought them to the point where they are becoming a viable alternative to traditional magnetic storage for some applications. Current file system and device level I/O scheduler design is optimized for rotational magnetic hard disk drives. Since solid state devices have drastically different properties and structure, we may need to rethink the design of some aspects of the file system and scheduler levels of the I/O subsystem. In this thesis, we consider the current approach to I/O scheduling and show that the current scheduler design may not be ideally suited to solid state devices. We also present a framework for extracting some device parameters of solid state drives. Using the information from the parameter extraction, we present a new I/O scheduler design which utilizes the structure of solid state devices to efficiently schedule writes. The new scheduler, implemented on a 2.6 Linux kernel, shows up to 25% improvement for common workloads.

DEDICATION

To my family

ACKNOWLEDGEMENTS

I am first and foremost extremely grateful to my advisor, Dr. A. L. Narasimha Reddy for allowing me the opportunity to work with him throughout my graduate career. His advice, guidance and encouragement have been a tremendous help to me and without him, none of this work would be possible. I am also grateful to all of the faculty that have taught me over the course of my tenure at A&M.

I would also like to thank my family, especially my parents, for supporting me along my way, through the good times and the bad.

Finally, I am incredibly thankful for my fiancée and her support throughout the entire process of researching and writing this thesis. Her patience and love have encouraged me more than mere words could ever express.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

CHAPTER

I INTRODUCTION 1

II I/O SCHEDULERS 8

- Motivation 8
- Noop Scheduler 9
- Deadline Scheduler 10
- Anticipatory Scheduler 11
- Completely Fair Queuing 12
- Scheduler Comparison 13

III FLASH DRIVE CHARACTERISTICS 17

- Random v. Sequential Performance 17
- Flash-specific Performance Tests 24

IV NEW SCHEDULER DESIGN 31

CHAPTER Page

V RESULTS 37

Iozone Sequential Tests 37
Iozone Mixed Workload Tests 38
Postmark Benchmark 39
Dbench Performance 41
OLTP Benchmark 42
NILFS Performance 43

VI CONCLUSIONS AND FUTURE WORK 45

REFERENCES 47

VITA 51

LIST OF FIGURES

Page

Figure 1 – Internal Flash Architecture 2

Figure 2 – Simplified Linux I/O Stack 5

Figure 3 – Schedulers Reordering I/O Requests 13

Figure 4 - Scaled Comparison of Schedulers on HDD (Lower is Better) 14

Figure 5 - Scaled Comparison of Schedulers on SSD (Lower is Better) 15

Figure 6 - Sequential Read Performance 18

Figure 7 - Random Read Performance 18

Figure 8 - Sequential Write Performance 19

Figure 9 - Sequential Write Performance (Detail) 20

Figure 10 – Random Write Performance (Detail) 20

Figure 11 – Multi-process Performance (Higher is Better) 21

Figure 12 – Delay Experiment (One Process, Higher is Better) 22

Figure 13 – Delay Experiment (Eight Processes, Higher is Better) 23

Figure 14 – 32KB Stride Test 25

Figure 15 – 64KB Stride Test 26

Figure 16 – 512k Stride Test 27

Figure 17 – 1M Stride Test 27

Figure 18 – Striding Test Summary for Transcend 29

Figure 19 – Striding Test for Memoright Drive 29

Page

Figure 20 – Striding Test for Samsung Drive 30

Figure 21 - Four Request Positioning Cases (1-4 from Top) 35

Figure 22 - Iozone Sequential Tests (Higher is Better) 37

Figure 23 - Iozone Mixed Workload Performance (Higher is Better) 39

Figure 24 - Postmark Performance (Lower is Better) 39

Figure 25 - Postmark Summary (Lower is Better) 41

Figure 26 - Dbench Performance (Higher is Better) 42

Figure 27 - OLTP Database Performance (Lower is Better) 43

Figure 28 - Dbench Performance on NILFS2 File System. (Higher is Better) 44

CHAPTER I

INTRODUCTION

In recent years, enormous advances in the area of solid state storage devices have begun to attract wide attention[7,4,17,24,9]. With capacities exceeding 100GB and prices falling drastically, consumers are now able to seriously consider the possibility of using a flash-based drive as their primary storage device in their computer. This is becoming an especially attractive option in many laptops where size, weight, and power consumption are all primary concerns.

As solid state drives become more ubiquitous, it becomes necessary to examine and adapt to the differences between flash drives and more traditional magnetic-based storage devices. For example, while read speeds of flash drives are generally much faster than traditional disks, write speeds, especially random write speeds, may not have such a clear advantage in performance. Since solid state devices have no mechanical parts, they have no seek times in the traditional sense, but there are still many operational characteristics that need to be taken into account. SSDs are organized into multiple banks that can be independently accessed[16,10,15,1]. Figure 1 shows an example, from Texas Instruments, of a possible internal architecture for a flash-based disk[28]. This is just one of many proprietary implementations, but they share similar structures.

The journal model is *IEEE Transactions on Automatic Control*.

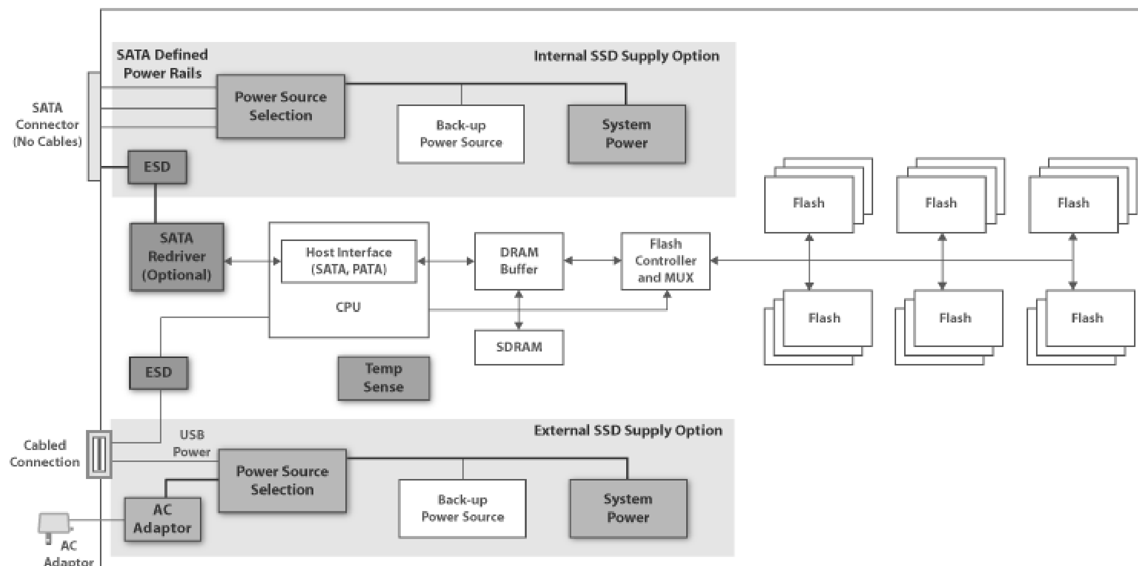

Figure 1 – Internal Flash Architecture

In Figure 1, we can see the internal architecture that most flash drives share. There are numerous banks of flash memory all connected to an FTL and SATA (or other standard) interface by means of a controller chip. The level of parallelization and efficiency has been increasing in many of the drive architectures, leading to greatly improved performance. The number of banks and the number of buses used to connect them determine the amount of parallelism and hence the raw maximum throughput of the devices. Data is organized into blocks that are larger than a typical disk sector of 512 bytes even though the data can be addressed and accessed at 512 byte granularity. The larger blocks enable more efficient erase cycles.

Since each vendor has a specific configuration and construction for their drives, it is important to point out that the internal configuration can have a large effect on how

the drive performs. Since the memory cells in the flash drive are divided into banks with separate I/O paths to each bank, it is possible to have a large amount of parallelization within the drive itself. Often, this is implemented with a DRAM buffer inside the drive which will reorder the writes so that all banks are used to the full extent possible. In general, the more banks the flash memory is divided into, the greater possibilities there are for increased parallelization and greater performance.

Data within the SSDs needs to be erased before the blocks can be used for new writes. As a result, most SSDs employ copy-on-write mechanisms and remap blocks through the Flash Translation Layer (FTL). Different devices employ different policies at the FTL level for remapping blocks, garbage collection and erase operations[16,2,19,8]. The overarching purpose of the FTL is to make the linear flash device appear to the operating system and file system as if it were a normal block disk drive. To accomplish this, it has a number of smaller tasks to perform such as virtual block management, write-in-place emulation, wear leveling, or page merging. To emulate a block device, the FTL causes the flash media to appear to the operating system as a contiguous array of blocks. Once data is written to these virtual blocks, the FTL remaps the virtual block number to the physical location where the data actually needs to be written. While it appears to write the data normally as on a regular block device, the FTL actually writes the data in a new location and merely marks the old data as 'dirty' or 'invalid' and waits for a cleaner process to erase the dirty data. In addition to this copy-on-write, the FTL must manage the fact that flash drives must erase data in large blocks called erase blocks. When the drive needs to erase data, the FTL must move all valid

data out of the erase block which is to be erased and copy it to a new location on the drive without revealing this to the operating system. Finally, the FTL implements wear leveling to ensure that all flash memory cells are written to as close to evenly as possible. More advanced FTLs may add additional functionality, but this core functionality is present in almost all FTLs on storage devices.

As we will show later, while the performance of random I/Os is inferior to sequential I/Os even with SSDs, the random read performance of SSDs is superior to magnetic disk drives. However, the SSDs suffer severe performance degradation with random writes. The poor random write performance is a byproduct of several factors: the long erase cycles needed to erase data in SSDs and the expensive merge costs involved in merging the partial data from the previous version and current version of the block within the FTL. While several design enhancements have been proposed to reduce this random write performance gap[16,10,15], it is expected that random writes will continue to suffer lower performance for various reasons. First, the inherent processes involved in erasing a block of data favor large blocks (for reliable erasure and the stability of the cells). Second, with increasing capacities, larger blocks allow smaller mapping tables required for Copy on Write mechanisms employed in the FTL layer.

Many layers in the I/O subsystem have been designed with explicit assumptions made about the underlying physical storage. Although designs can vary, most assume that at the physical device level, there is a traditional magnetic or optical device which rotates and uses mechanical parts to access the data. In order to take full advantage of

the possibilities of solid state storage devices, we must reconsider some of the design assumptions that have been made and redesign the I/O subsystem for these new devices.

In the Linux operating system, all I/O operations must pass from the originating application down through several layers, all of which can modify or optimize the request as it sees fit. A greatly simplified Linux I/O stack is shown in Figure 2.

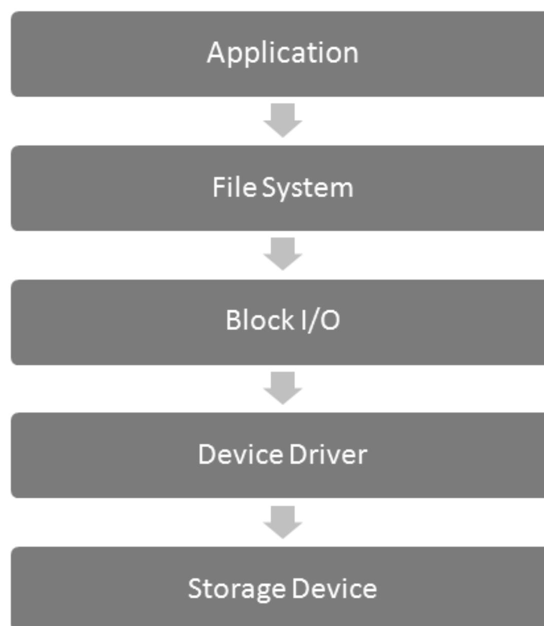

Figure 2 – Simplified Linux I/O Stack

Keep in mind that every distribution of Linux is free to create its own I/O stack and functionality may be different or more intermediate layers may be added. In general though, an application issues a request to the file system which then passes the request on to a block I/O layer which then writes it to the disk via the device driver.

Many file system approaches have been designed to try and take advantage of the specific characteristics of solid state drives[25,35,22,26,23]. Since flash drives are

particularly poor at random writes, the most common approach to improving performance has been to implement a log-structured file system. Initially proposed in the early 90s, log-structured file systems attempt to solve the problem of large seek times by treating storage as a circular log and always writing to the head of the log, resulting in write operations that occur almost exclusively sequentially. While log structured file systems may solve or mitigate the poor random write performance of the SSDs, the random writes may remain an issue when a log structured file system cannot be employed (for legacy/compatibility or other reasons). In addition to log structured file systems such as LogFS and NILFS2, there have been many other file system approaches such as JFFS2, YAFFS, and FFS2 [32,20,3,11]. Some of these seek to work in conjunction with the FTL layer while others incorporate the functions of the FTL into the file system layer. In addition to the many file system solutions that have been offered for flash disks, numerous buffer management strategies have been investigated which can also affect device throughput. Cache management strategies such as ARC or LIRS can have a measurable impact in I/O requests at lower layers of the I/O stack[21,13,30].

In this thesis, we address the differences of SSD characteristics at the I/O scheduler, which resides in the block I/O layer of the stack. Current I/O schedulers take seek time, rotational latency overheads of magnetic disk drives into account in scheduling requests[12,5,27,31,33]. Because of the differences in device characteristics, the current schedulers may not adequately schedule requests for SSDs. In order to design a suitable scheduler for SSDs, we first characterize the performance of these drives to

extract suitable parameters. These parameters are then used in the scheduler for making appropriate scheduling decisions. While there are no hard and fast rules regarding which layers have which responsibilities, we will consider the I/O scheduler to only reorder and merge requests in the outstanding request queue and otherwise service them as quickly as possible. As such, we will leave any buffering or caching strategies to the file system layer. In this thesis, we also propose two methods which we believe will work in tandem to address some of the performance issues of flash drives. First, we propose a framework for extracting important device parameters which will allow us to redesign parts of the I/O subsystem to enhance performance. Second, using the information we gained from our parameter extraction tests, we propose a new I/O scheduler design which will order writes more efficiently on the flash drive and result in increased throughput for many workloads.

In Chapter II we outline the current approach to I/O schedulers and some of the assumptions which are made about the physical storage devices. In Chapter III, we present our framework and tests which we used to model the performance of flash disks. In Chapter IV, we present our approach to scheduler design based on some of the results from Chapter III. Chapter V presents our implementation and benchmark results. Finally, our conclusions and ideas for possible future work are presented in Chapter VI.

CHAPTER II

I/O SCHEDULERS

Motivation

In traditional hard disks, data is addressed using the hard disk's geometry of cylinders, heads and sectors. With logical block addressing, rather than address the hard drive in terms of cylinders, heads and sectors, the hard drives map each sector to a unique block number. When the operating system wants to perform an operation on the hard drive, it issues a block request and the hard drive then accesses the physical location specified by that block number. The important note about logical block addressing is the fact that block addresses tend to map sequentially onto physical addresses. This means that although not technically required, block x tends to be physically adjacent to block x+1 and so on.

When a varied application pool needs to access the data that is stored on a disk, requests tend to come from different locations all over the disk. In traditional hard drives, writing or reading from nearby physical sectors is far less expensive than reading from distant sectors due to the expensive seek operations involved. Because of this, I/O schedulers tend to strive to reduce seek time and handle reads and writes to minimize head distance travelled. In addition, more advanced I/O schedulers attempt to provide other functionality such as starvation prevention, request merging, and inter-process

fairness. By default, the 2.6 Linux kernel includes four I/O schedulers which can be switched at runtime, rather than at compile time. We briefly describe these schedulers below as representative I/O schedulers.

Noop Scheduler

The noop scheduler is so called because it performs a bare minimum of operations on the I/O request queue before dispatching it to the underlying physical device.

The noop scheduler assumes that either the request order will be modified at some other layer of the operating system or that the underlying device is a true random access device. When the noop scheduler receives a request, it first checks to see if it can be merged with any other outstanding request in the queue. If it can be, it merges the two requests and moves onto the next request. If no suitable merge can be found in the queue, the new request is inserted at the end of an unordered first-in-first-out (FIFO) queue and the scheduler moves on to the next request.

Because of its suitability for random access devices and the assumption that solid state devices are random access devices, the noop scheduler has become a popular choice for systems which utilize solid-state devices. However, these assumptions may not hold and so noop may not truly be the most efficient scheduler for solid state devices.

Deadline Scheduler

Due to the simplicity of the noop scheduler, it cannot make any guarantees about service time for any given request. Since different applications may have different performance requirements, service time guarantees may be desireable. To address this shortcoming, the deadline scheduler was created.

Based on the noop scheduler, the deadline scheduler adds two important features. First, the unordered FIFO queue of the noop scheduler is replaced with a sorted queue in an attempt to minimize seek times, and second, it attempts to guarantee a start service time for requests by placing deadlines on each request to be serviced.

To achieve this, the deadline scheduler operates on four queues; one deadline queue and one sector sorted queue for both reads and writes. When a request is to be serviced, the scheduler first checks the deadline queue to see if any requests have exceeded their deadline. If any have, they are serviced immediately. If not, the scheduler services the next request in the sector sorted queue, that is, the request physically closest to the last one which was serviced.

This scheduler generally provides increased performance from the noop scheduler due to the fact that the sorted queues attempt to minimize seek times. In addition, by guaranteeing a start service time, the deadline scheduler can provide service time guarantees.

Anticipatory Scheduler

The anticipatory scheduler was introduced as a modification on the deadline scheduler to solve the problem of "deceptive idleness" that can occur under certain workloads. In general, applications issue their read requests in a synchronous manner. This is because often an application needs information contained in the read to issue the next I/O request. Because of this, a process may appear to be finished reading from the disk when in actuality, it is processing the data from the last operation in preparation for the next operation. Since the process appears to be finished with its requests, the scheduler may move on to working in a different portion of the disk. However, when the time for the next read or write from the original process occurs, the disk head must again seek back and performance degrades as a result of these seeking operations. Most often, this problem manifests itself in the 'writes-starving-reads' problem. Since writes are generally asynchronous, the scheduler will always have outstanding writes to service. If reads are being issued in a synchronous manner, as they usually are, then they will not be serviced in a timely manner since the scheduler will have moved on to one of the outstanding write requests by the time the next read request arrives.

Anticipatory scheduling attempts to compensate for this by pausing for a short time after a read operation, in essence anticipating the next read operation that may occur close by. Even though the anticipation sometimes may guess incorrectly, even if it is moderately successful, it saves numerous expensive seek operations.

Anticipatory scheduling also builds off the deadline scheduler by introducing the concept of a “one-way” elevator, which attempts to only serve requests in front of the disk head, to take advantage of the rotational nature of the disk. This assumption about the rotational nature of the disk will naturally have to be reevaluated when we apply the anticipatory scheduler to a flash disk.

Completely Fair Queuing

The final and default scheduler that is included in the 2.6 Linux kernel is the Completely Fair Queuing scheduler. The CFQ scheduler attempts to provide fair allocation of the available I/O bandwidth to all initiators of I/O requests.

CFQ places all synchronous requests which are submitted into one queue per process, then allocates timeslices for each of the queues to access the disk. The queues are then served in a round robin fashion until each queue’s timeslice has expired. The asynchronous requests are batched together in fewer queues and are served separately. Even though anticipation is not explicitly included in the system, the CFQ scheduler avoids the writes-starving-reads problem due to a natural extension of the fairness mechanism. Since each process is being serviced individually, a process’s synchronous reads will be serviced together without interference from another process’s asynchronous writes.

Another feature of note in the CFQ scheduler is that it also adopts the “one-way” elevator from the anticipatory scheduler and makes some of its scheduling decisions

based on the notion of a rotating disk at the physical layer. Like the anticipatory scheduler, this makes it a prime candidate for modifications which would improve performance on solid state devices.

Scheduler Comparison

To illustrate how the different schedulers can affect performance on the same workload, consider the sample workload given in Figure 3. Although the initial workload is the same, each scheduler reorders it based upon the rules of that scheduler. In this diagram requests 1, 3 and 6 come from the same process and are writes, requests 2, 4 and 7 are writes from a different process and 78, 80 and 81 are reads from a third process. The number represents the sector we wish to access.

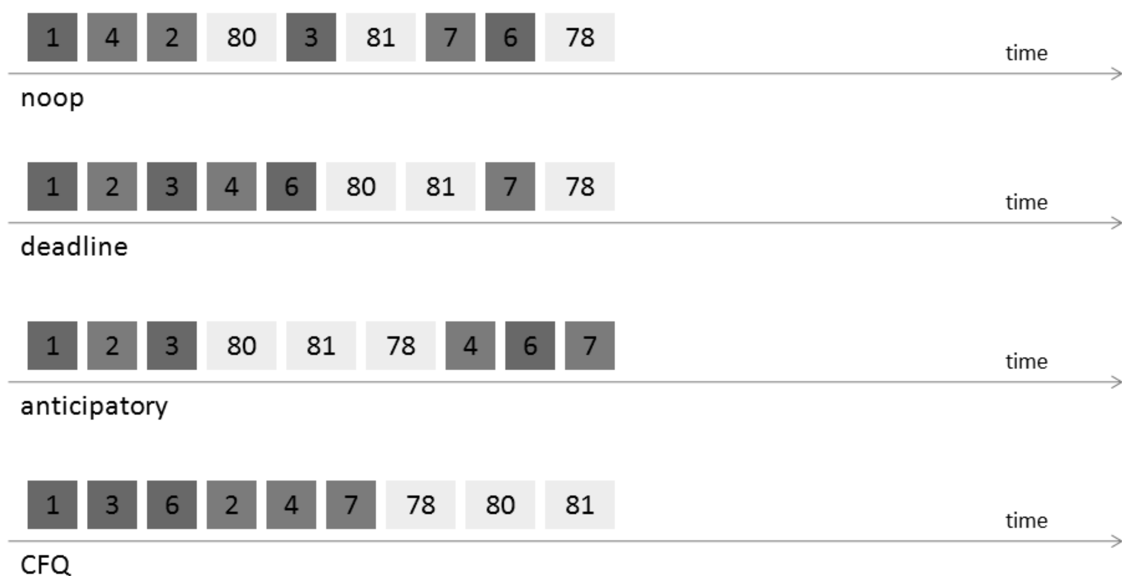

Figure 3 – Schedulers Reordering I/O Requests

In the noop scheduler, the requests are serviced merely in the order in which they are received. Some of the requests may be merged with others, but otherwise, they are issued in a FIFO order. In the deadline scheduler, we see the requests being served in sector order until request 80 hits its deadline and must be serviced. In the anticipatory scheduler, once a read request (80) is served, the scheduler is waiting to see if another read request comes. When it does (81), and it is immediately serviced, so seek time is reduced. In the CFQ scheduler, each process gets fair access to the disk for a certain period of time to service its requests before relinquishing control to another process. This results in inter-process fairness and good locality. The actual ordering of requests will vary depending on tunable parameters and the time between requests, but the same basic rules will apply.

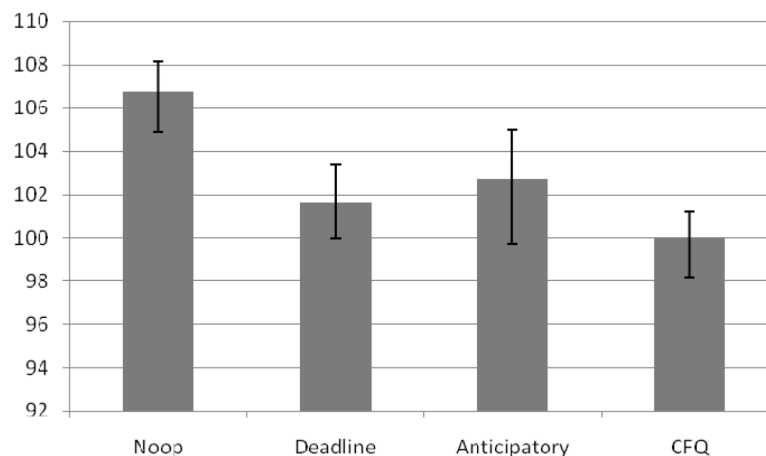

Figure 4 - Scaled Comparison of Schedulers on HDD (Lower is Better)

Figure 4 shows a comparison of the four default Linux schedulers on a sample Postmark workload [14] on a magnetic disk. The Postmark test was run on a dataset of

10000 individual files, with 50000 transactions being run over the span of the files. Read and write operations were equally weighted. It is observed that CFQ performs the best of the schedulers while noop performs the poorest.

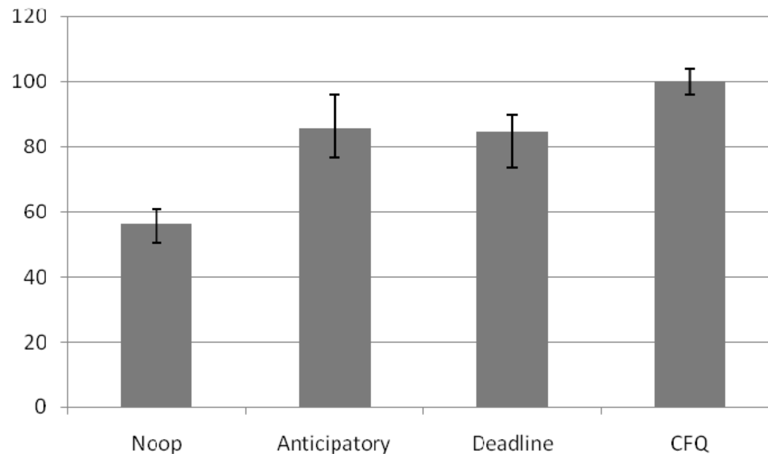

Figure 5 - Scaled Comparison of Schedulers on SSD (Lower is Better)

Figure 5 shows the performance of the schedulers on the same workload on a SSD. Figure 5 serves as an illustration highlighting the differences that exist between traditional hard drives and SSDs. On a similar workload, the CFQ scheduler actually performs the poorest on the SSD while the noop scheduler performs the best. This is quite the opposite of what we have seen in Figure 4. It is also observed that the performance differences between the schedulers are much more significant for a SSD than for a traditional hard drive.

While this test employs only one sample workload, it serves to illustrate the potential impact of the scheduler decisions on the performance and motivates us to investigate if the schedulers can be improved for SSDs.

We study the characteristics of SSDs such that these characteristics could be factored into the design of an appropriate I/O scheduler for these devices.

CHAPTER III

FLASH DRIVE CHARACTERISTICS

Random v. Sequential Performance

In order to effectively design scheduling algorithms for solid state devices, it is important to gain an understanding of how flash drives perform under different workloads. All of the data in this section was gathered using a modified version of the IOZone benchmark[6]. We employed three different SSDs in our evaluations. These drives are from three different manufacturers and were roughly available in market six months apart from each other. These drives have considerable differences in raw performance numbers. While these three drives may not completely represent all the diversity of the available SSDs, they provide a good representative sample of the SSDs in the market. The three drives we studied are: Transcend TS16GSSD25-S 16GB drive, Memoright GT 32GB drive and the Samsung MCBQE32G8MPP 32GB drive.

We will present the results from the Transcend drive on the Linux 2.6.25.15 kernel. We will also highlight the results from the other drives below.

First we will examine both the sequential and random read and write performance of the drive under varying record and file sizes.

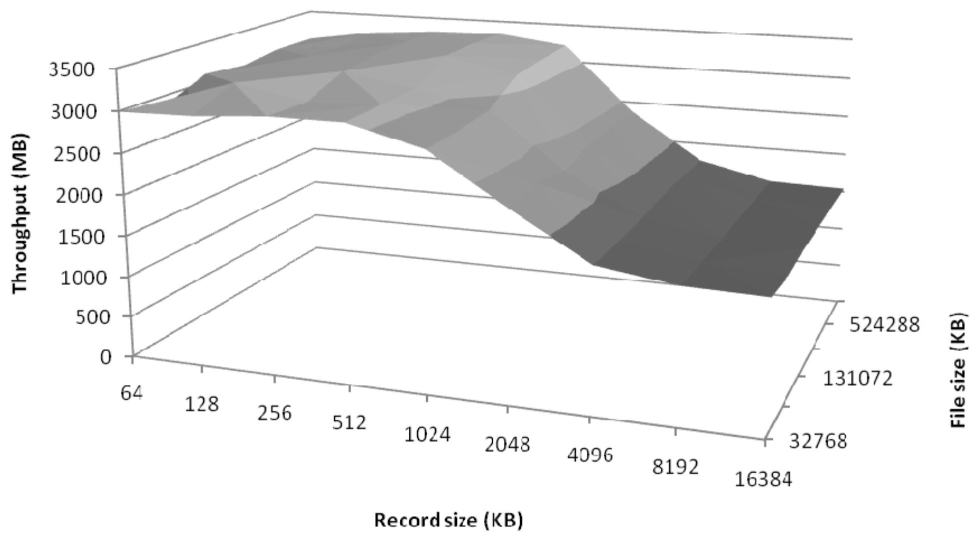

Figure 6 - Sequential Read Performance

Figure 6 shows a surface curve of the sequential read performance of the flash drive. Read performance appears to be unaffected by file size, but appears to degrade slightly as more data is requested to be read.

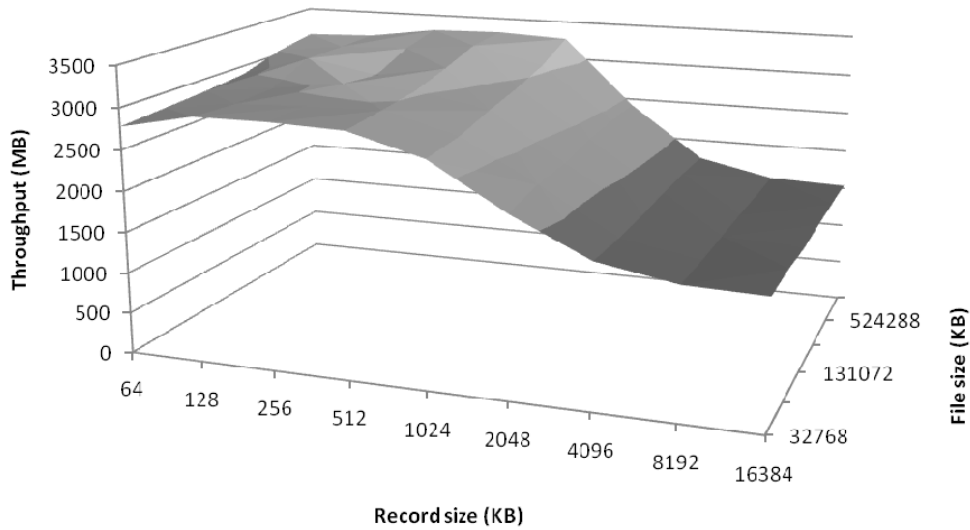

Figure 7 - Random Read Performance

Figure 7 shows a surface curve of the random read performance of the drive. A comparison of the two graphs in Figure 6 and Figure 7 shows that for all intents and purposes, performance appears to be nearly the same. Traditional thinking about flash drives says that this should be no surprise at all. With no moving parts, there should be no seek time, and thus, no difference between sequential and random operations.

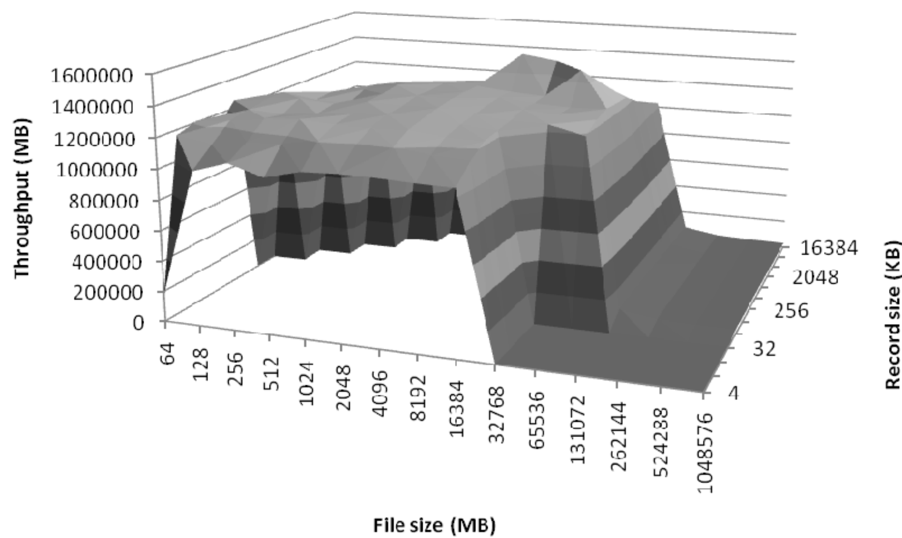

Figure 8 - Sequential Write Performance

Figure 8 shows the sequential write performance of the drive. The first thing that stands out about the write performance of the flash drive is the precipitous drop that occurs once the file size exceeds 512MB. At this point, the machine's RAM can no longer buffer all of the writes that are being attempted and so the drive's performance becomes a major bottleneck. We focus on the write performance in the region where it is determined by the SSD's performance, rather than that of the buffer.

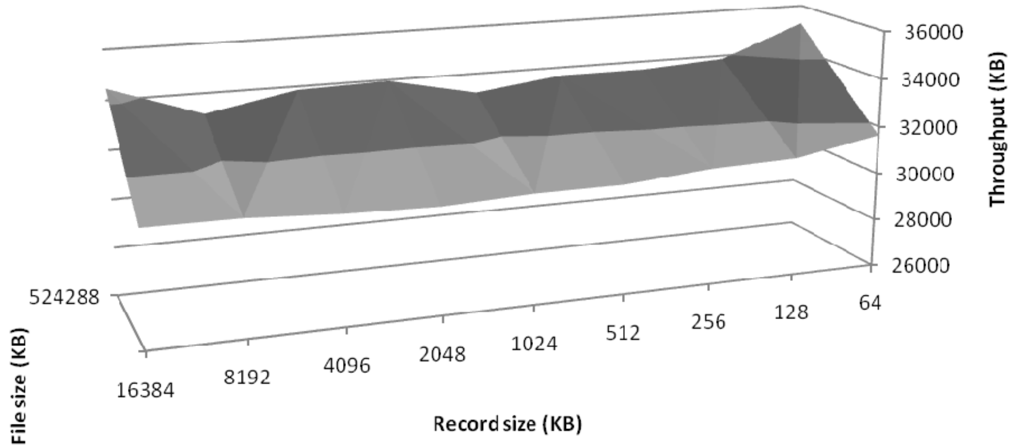

Figure 9 - Sequential Write Performance (Detail)

In Figure 9, we see that once the RAM caching effect has been nullified, the drive performs its writes in the 30MB/s range across all record sizes.

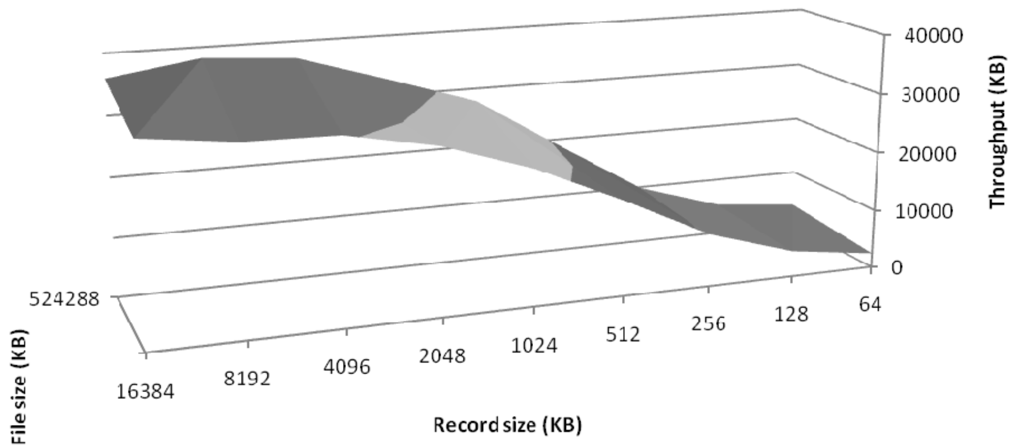

Figure 10 – Random Write Performance (Detail)

Figure 10 shows the same region for random write operations. In Figure 10, we notice that unlike the comparison between sequential and random reads, random writes experience a severe drop in performance as the record size gets smaller. In essence, as the writes become more random, the performance of the drives becomes poorer and poorer. If there truly is no seek time, we would expect the random write curve to look like the sequential write curve, much like it did with the read curves.

Tests were also performed for multiple processes writing at the same time. For each scheduler included in the Linux operating system, sequential write tests were performed with between 1 and 8 processes trying to write. The results are summarized in Figure 11.

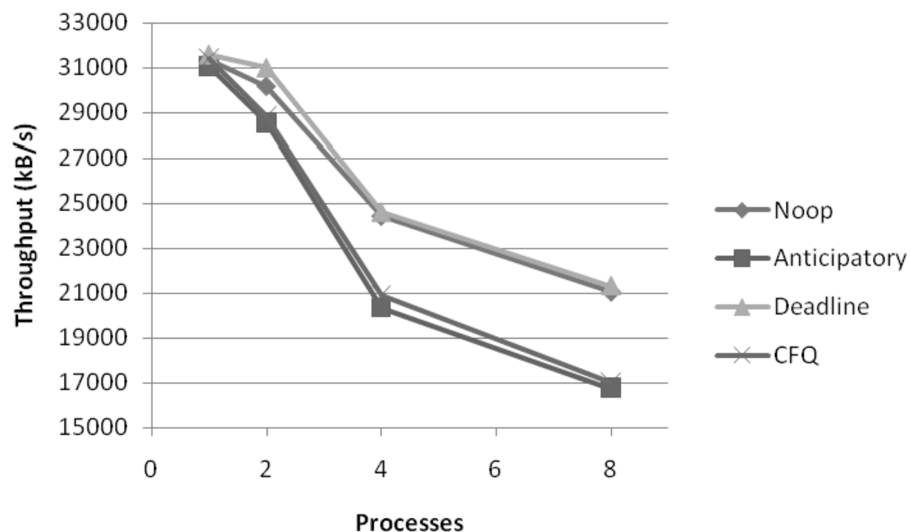

Figure 11 – Multi-process Performance (Higher is Better)

As the number of processes increases, the throughput of the device decreases drastically. Originally, it was considered that too many writing processes may be

overloading the scheduling queue or the device buffer. To test for this, a simple back-off/delay test was created. Every time a write was issued, the benchmark had a chance to delay the next operation for a period of time. The results for a 10-50% chance of delay and delay values between 1-500ms are shown in Figure 12 for one process.

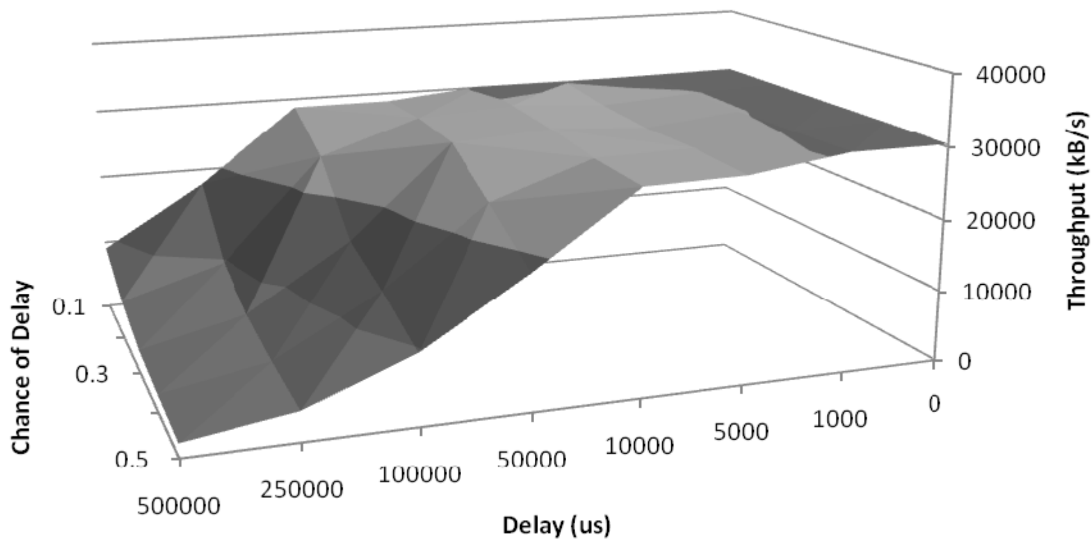

Figure 12 – Delay Experiment (One Process, Higher is Better)

For one process, the results are unsurprising. As the process has a higher chance to issue the next request with a higher delay, the throughput decreases. The results for the case when the same test is applied across 8 processes are shown in Figure 13.

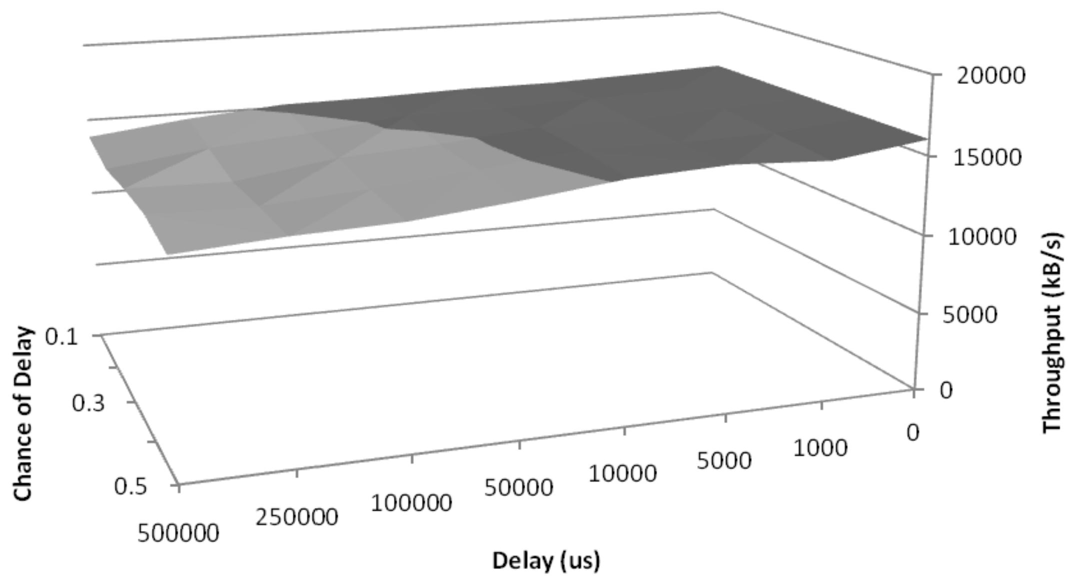

Figure 13 – Delay Experiment (Eight Processes, Higher is Better)

As shown in Figure 13, there is little performance difference even when the processes are aggressively backing off of their writing duties. In this case, it appears that more processes do not necessarily lead to more efficient utilization of parallelization and increased performance. The performance has actually decreased from 30MB/s with one process to about 16MB/s with 8 processes.

Clearly, while there may not be seek time in the traditional sense, clearly there is some cost during writes inherent to the SSD's structure which should be considered during I/O scheduling. In order to understand the characteristics that may determine these costs in random writes, we have devised a number of tests.

We expect that the random write costs are determined by the internal block sizes and the policies involved during writes and erase cycles. A full block write is efficient since the older version of that block can be efficiently garbage collected. When data in a

block is partially updated, the updated data needs to be coalesced with the still valid data from the older version of the block, increasing garbage collection overhead. While the page remapping algorithms in the FTL determine the exact merge costs, it is sufficient for the I/O scheduler to know the size of the block. The I/O scheduler can then try to avoid paying the block crossing costs during writes in order to increase the efficiency of writes.

Flash-specific Performance Tests

In order to accurately determine the size of the flash blocks, we have developed a series of strided writing tests which will tell us when merge costs occur, therefore telling us the size of the flash blocks.

The first thing one notices when looking at a trace of latencies for a sequential write on a flash disk is that there are large peaks in write latencies occurring frequently. It is these large peaks which we believe are primarily responsible for decreased write performance in these drives.

For our tests, we needed to determine exactly when these peaks occurred to determine their cause and if there was any regularity associated with them. The strided write tests are structured so that the process will write a small amount of data (4k in our case), seek forward a fixed amount, write another small amount of data, seek forward a fixed amount, write again, and so forth. The purpose of this test is twofold. By varying the stride size, we can accurately determine if there is a bottleneck caused by traversing

blocks or if there is a bottleneck caused by some secondary component such as an on disk cache which is being periodically flushed. If the latency peaks occur at fixed intervals independently of the amount of data written, we can assume that there is some boundary mechanism that triggers when a specific offset is crossed. However, if, for varying strides, the latency peaks occur only after a set amount of data has been written, we can conclude that it is a buffering or caching bottleneck. This test, hence allows, us to simultaneously determine the internal cache or buffer employed in the SSD along with its block size.

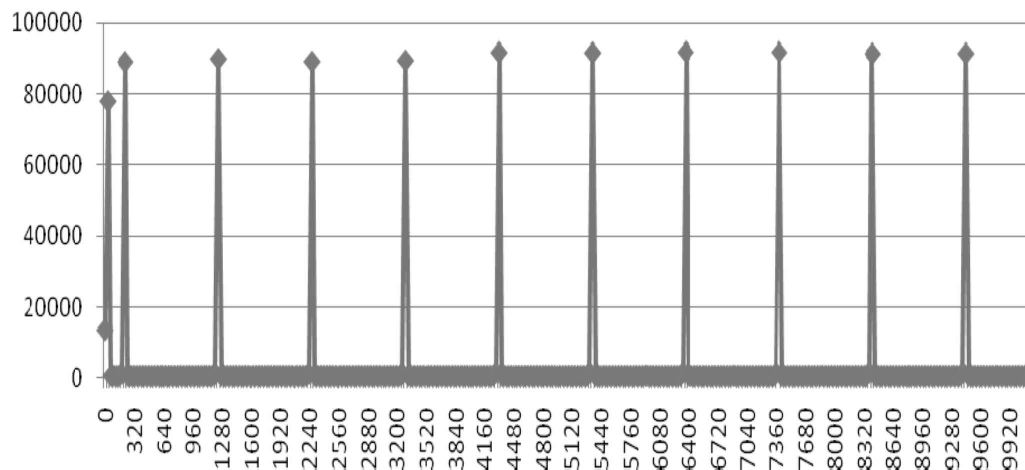

Figure 14 – 32KB Stride Test

Figure 14 illustrates the output of the 32K striding test. As described above, the process in question will write 4K, skip ahead to the next 32K boundary, then write 4K, etc. The 32K striding test gives us numerous important insights into the structure of the flash drive. First, the peaks occur with regularity, so it is safe to assume that there is some function of the drive which is causing these writing delays. In this particular test,

the latency peaks occur regularly every 1024KB offset, or after 128KB of data has been written in total. From this information, we can make one of the two conclusions. Either there is a characteristic of the drive that causes a peak after each 1MB page has been written or there is a 128KB buffer at some point in the pipeline that is being filled and must be flushed periodically. To determine which of these to be the case, we can perform striding writes of other sizes.

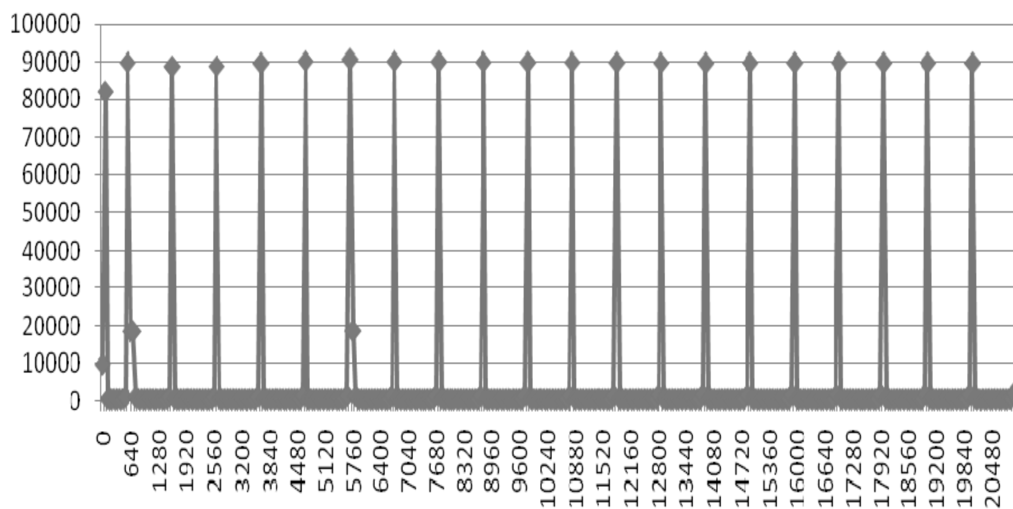

Figure 15 – 64KB Stride Test

Figure 15 is the resulting graph from a 64K stride test. The important result from this test is that the latency peaks still occur regularly at 1MB intervals. However, compared to the 32KB stride test, we have only written half of the total amount of data as before when the latency peaks occur. This leads us to believe that the latency peaks are occurring independently of the total amount of data being written and instead are a function of the offset, not due to any internal buffering mechanism.

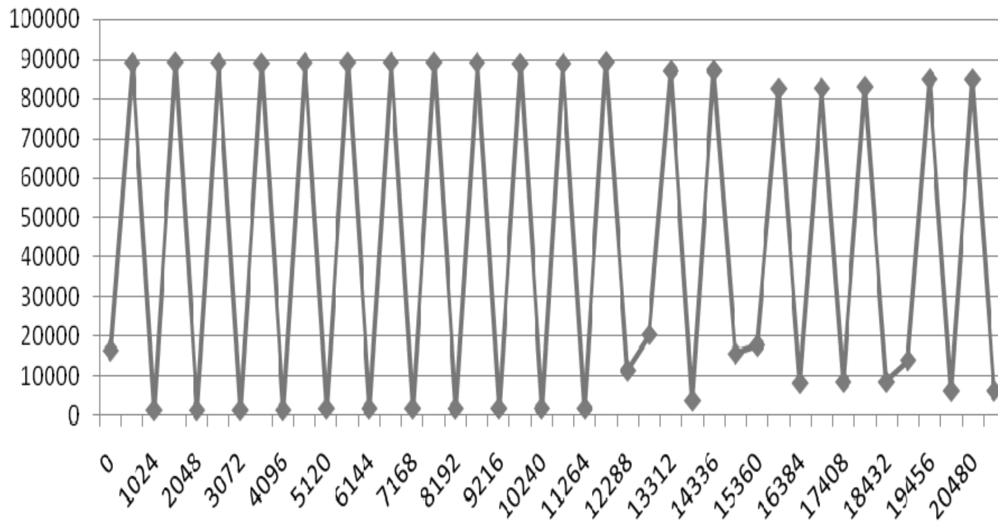

Figure 16 – 512k Stride Test

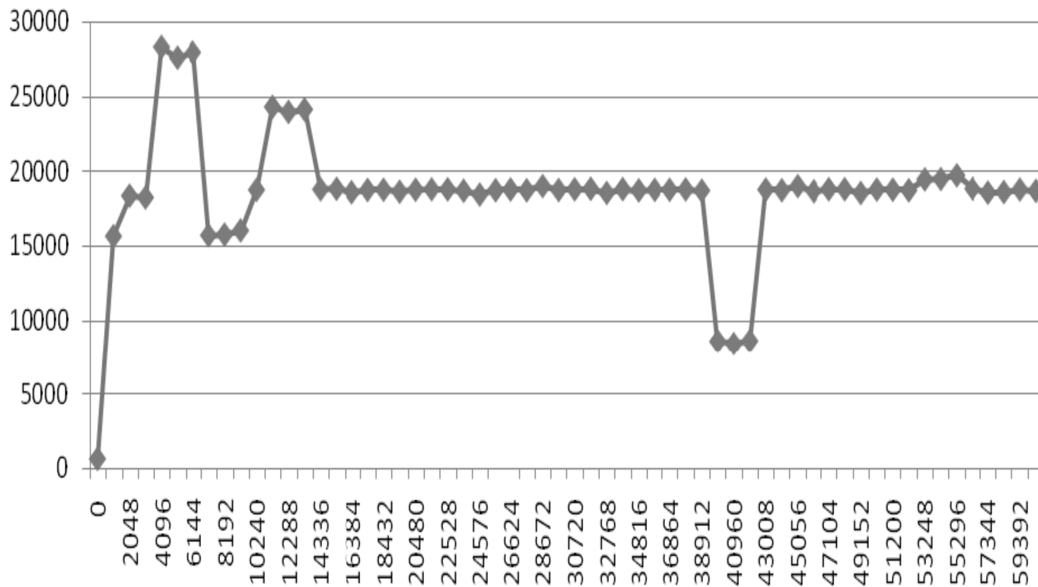

Figure 17 – 1M Stride Test

Figure 16 and Figure 17 show the latency information for larger stride sizes of 512K and 1M, respectively. For stride sizes of 512K, we see peaks occurring at every other disk access. This confirms the data presented at the smaller stride sizes, where the peaks occurred whenever a write was made in a different 1M block than the previous

write. The same peaks and valleys pattern is very evident. However, when we run identical tests with a 1M stride size, the peaks and valleys pattern that exists at smaller stride sizes disappears. Instead, there is a somewhat uniform slow access pattern, indicating that there are no 'quick' writes taking place anymore.

The write performance of the drive is shown in Figure 18. As we increase the stride size from 4K up to 2M, we see a precipitous drop in total throughput until the curve levels off at the 1MB stride size.

Based on these results, we can say that the peaks do not occur due to the total amount of data written, but rather they occur when we cross into another 1MB block. It is likely that this is due to this particular drive using 1MB block. The data we've gathered can be used to design a new scheduling algorithm which takes advantage of the architectural characteristics we've determined.

The implications for this 1MB boundary penalty for writes are explored further below during the design of a new scheduler.

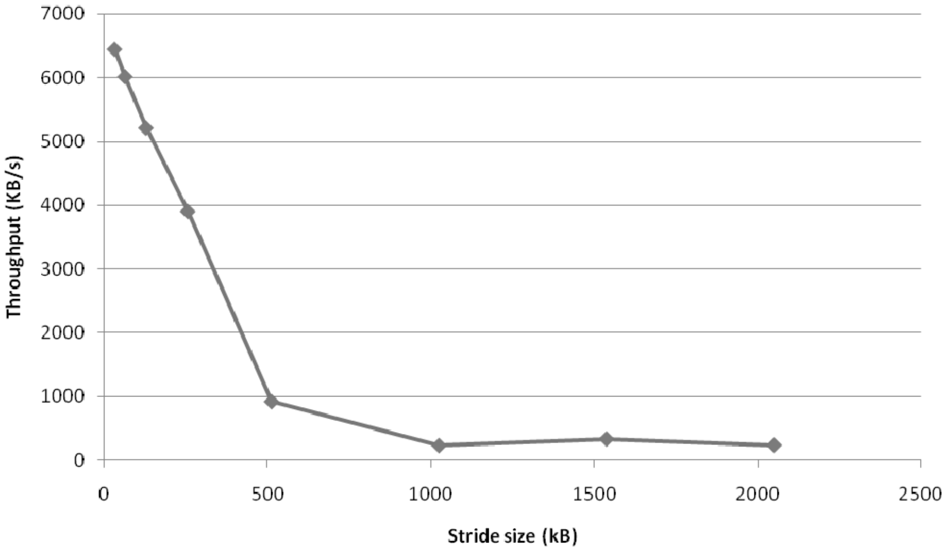

Figure 18 – Striding Test Summary for Transcend

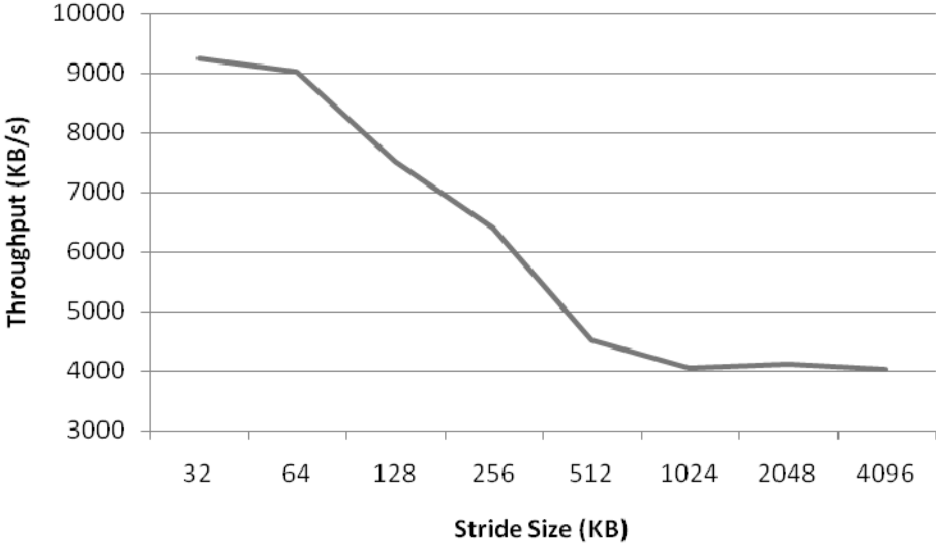

Figure 19 – Striding Test for Memoright Drive

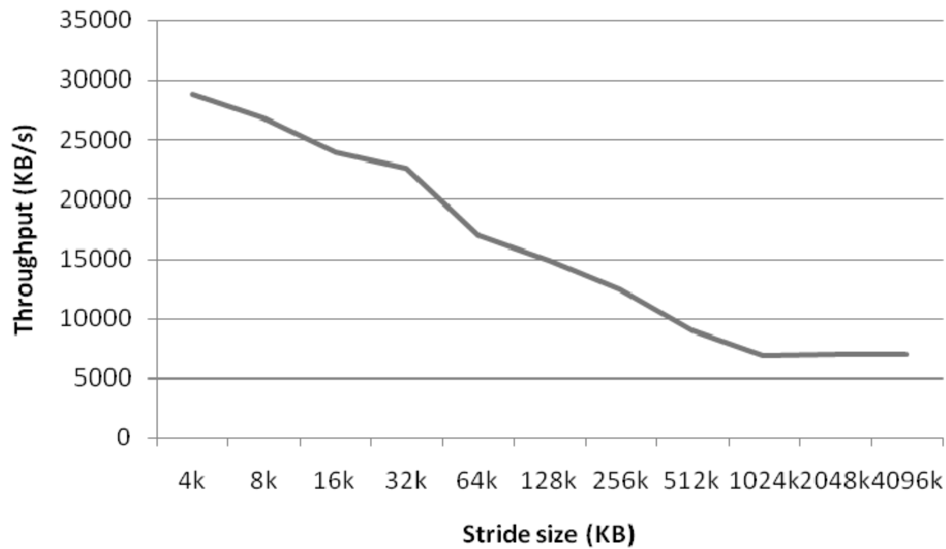

Figure 20 – Striding Test for Samsung Drive

We have carried out similar tests on the other two drives from Memoright and Samsung. The results from these tests are presented in Figure 19 and Figure 20. These tests confirm that these two drives also have a block crossing penalty for random writes at 1MB block sizes for the Memoright and Samsung drive. In addition, we also determined that the Samsung drive employs a significant internal cache, much like the arm buffers employed in traditional magnetic disk drives. While the internal buffer mitigates the impact of random writes up to a certain point, the applications may decide to turn off internal drive buffering in order to make sure that the writes have been recorded (as is done with most database applications and traditional magnetic disk drives).

CHAPTER IV

NEW SCHEDULER DESIGN

From the framework of tests we created in the previous section, it is clear that a significant penalty is imposed when new blocks in flash are written to. Because of this result, the most effective scheduler for a flash device would take into account the block size and would attempt to minimize the number of times that the penalty for writing a new block is imposed. To accomplish this, we design a scheduler which will always service requests that are in the same block as the previous request. This 'block preferential' scheduler seeks to improve performance in solid state drives by changing the notion of locality in disk devices. Traditionally, since seek time has been directly related to sector distance in drives, schedulers have attempted to minimize the seek distance since this has the most direct impact on device latency. However, in flash drives, that paradigm no longer holds. Even though some writes may be closer by raw sector count, if the writes are occurring at or near a block boundary, it may be more advantageous to service requests in an order not strictly based on sector distance. In conjunction with this write scheduling mechanism, our block-preferential scheduler has no directional seeking preference, like traditional schedulers do. While traditional schedulers implement 'one-way' elevators which are designed to take advantage of the rotational characteristics of disk drives, this feature does not apply to flash drives and in

some cases, may be disadvantageous. One-way elevators also help in preventing service starvation of requests. However, since we are basing our new schedulers on the anticipatory and CFQ schedulers, the deadline and fairness mechanisms inherent in both of the original schedulers will make sure that starvation does not occur. The block-preferential scheduler assigns no weight to requests based on their direction from the previous request.

We have implemented the block-preferential scheduler by modifying both the anticipatory scheduler and the CFQ scheduler and how they choose in what order to service requests. In later sections and results, our scheduler based on the anticipatory scheduler core will be referred to as block-preferential-anticipatory and our scheduler based on the CFQ core will be referred to as block-preferential-CFQ.

When deciding how to schedule a write and using the block boundary assumption from the previous sections, there are four basic cases of write ordering for which we need to plan.

Even though there is no head and no real direction in flash drives, for purposes of discussion we need to make some distinctions about terminology. When we say a request x is ‘in front’ of request y, we mean that the starting sector of request x is larger than the ending sector of request y. Correspondingly, for request x to be ‘behind’ request y, the starting sector number of request y must be greater than the ending request number of request x.

In our scheduler, we do not distinguish between backward seeks and forward seeks as the schedulers do for rotational disks. Without a rotational component, forward and backward seeks are operationally the same, and so no distinction is made between them in our scheduler.

In the first case, the scheduler will query the next two requests in the ordered queue, one which is in front of the previously served request, and one which is behind. In the case that both of the requests lie in the same block as the most recent request, the scheduler will service the largest of the two requests first.

In the second case, we have one request which lies in the same block as the most recent request and one which lies in a different block. To avoid the boundary crossing penalty which was discussed previously, the scheduler will always service requests in the same block as the request which was most recently serviced.

In the third case, when both the ‘front’ request and the ‘back’ request are in different blocks, the scheduler will look even further ahead in the request queue. If the two requests in front of the recently served request are in the same block, the scheduler will service the ‘front’ requests, since they are in the same block and we can avoid an extra boundary crossing.

In the case where both groups of ‘front’ and ‘back’ requests are in contiguous respective blocks, the scheduler simply chooses the largest combination of the two requests to service.

While it would be possible to search ahead even further in the request queue and come up with many different scenarios, we found that increasing the search space did not significantly affect the performance of the scheduler.

The four possible scenarios of request positioning are illustrated in Figure 21.

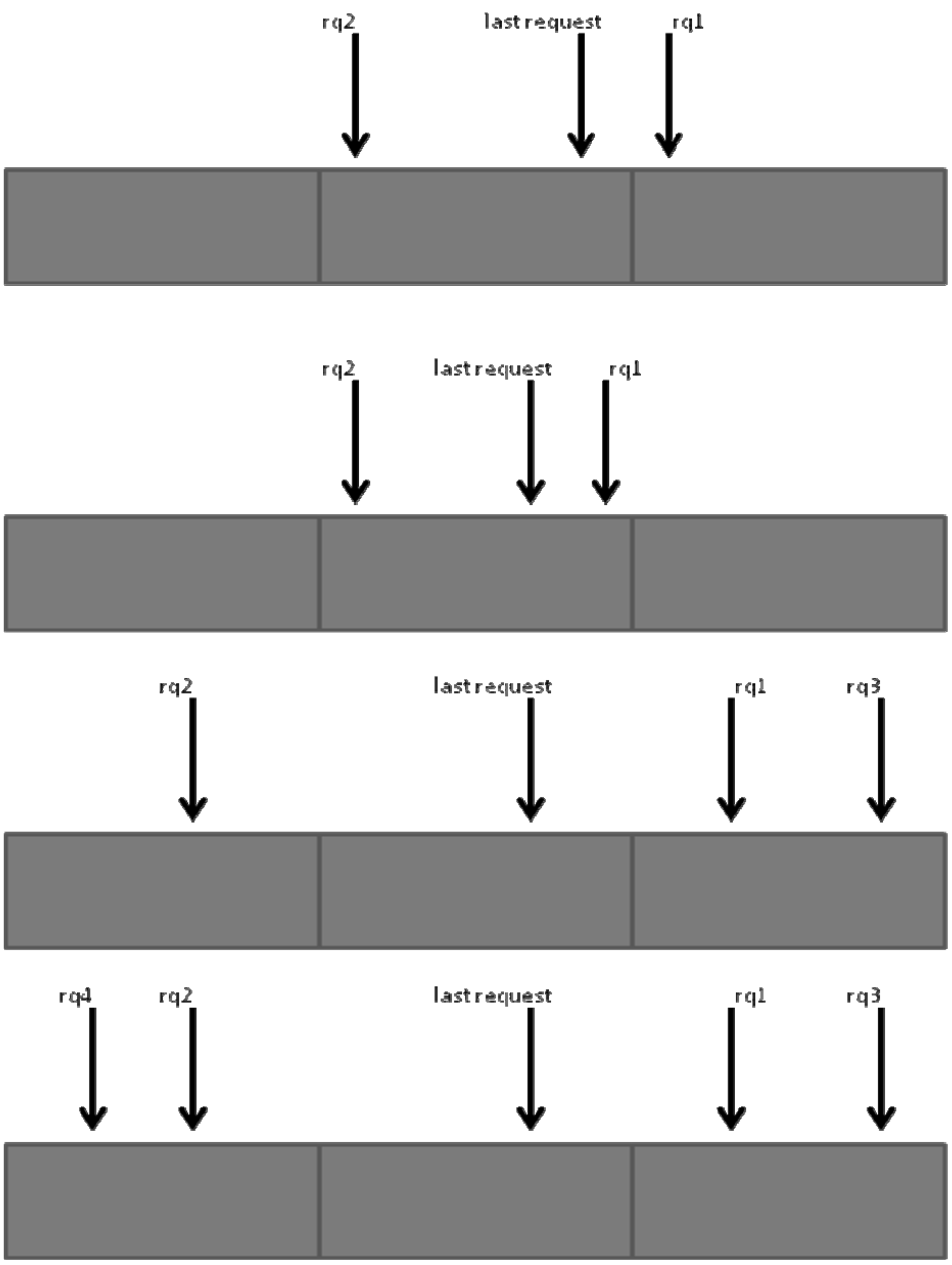

Figure 21 - Four Request Positioning Cases (1-4 from Top)

To implement the necessary changes, we focused on modifying the anticipatory and CFQ schedulers in Linux. Since the schedulers are similar in design, the changes needed for each were very similar. In the original schedulers, to choose the next request to service, there are two functions involved in each scheduler: as_choose_rq, as_find_next_rq, cfq_choose_req, and cfq_find_next_req. In the stock schedulers, these functions implement searching the red-black tree as well as the one-way elevator which is present in the anticipatory and CFQ schedulers. In our analysis though, since the one-way elevator is no longer applicable to solid state devices, this portion of the scheduler was removed. In its place, the scheduler now considers block boundaries rather than one-way elevator considerations or seek time. We allow the scheduler to insert requests into the queue as normal, but we change the portion of the scheduler that chooses which order to service the requests in. From our tests, we determined that no additional overhead was added from the stock schedulers as the only operations which were implemented were simple modular arithmetic and comparisons.

To summarize, the scheduler evaluates the two closest requests ahead of the previous request as well as the two closest requests behind the previous request and services the one which will result in the fewest boundary crossings overall. Saving even a few of the grossly expensive boundary crossing costs is well worth the use of the new scheduler in many applications.

CHAPTER V

RESULTS

We implemented the new scheduler design in the Linux 2.6.15.25 kernel. It involves a recompile of the kernel since significant modifications are made to the block scheduling files. The anticipatory and CFQ scheduler were modified while deadline and noop were left unmodified as a control group. The modified anticipatory and CFQ schedulers are expected to take SSD characteristics into account for I/O scheduling similar to how earlier schedulers considered magnetic disk drive characteristics in scheduling the I/O requests. We evaluate the modified schedulers and compare them with the unmodified I/O schedulers in the Linux distributions.

Iozone Sequential Tests

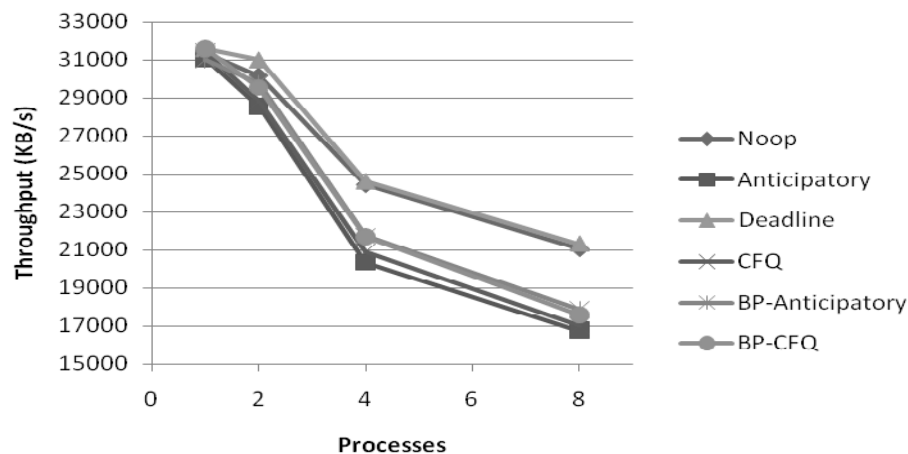

Figure 22 - Iozone Sequential Tests (Higher is Better)

The first test we ran was the standard Iozone [23] sequential write performance test. The modifications to the schedulers exhibit a small 2-5% improvement, shown in Figure 22, over their unmodified counterparts, but large gains are not to be expected from this write pattern. Importantly, however, we can show that there are no adverse effects on sequential write performance with our new scheduler.

Iozone Mixed Workload Tests

To evaluate the scheduler under a mixed workload environment, we ran an Iozone mixed workload test. This test involved a combination of reads and sequential writes to be performed. For this specific test, the reads occurred twice as often as the writes. The results from this test are shown in Figure 23.

For the mixed workload, we can see that all of the schedulers which actively address the deceptive idleness problem perform much better, as we would expect them to. The modifications to the CFQ scheduler provide a small gain in this scenario, however, the modifications to the anticipatory scheduler provide up to 15% improvement in the mixed workload scenario.

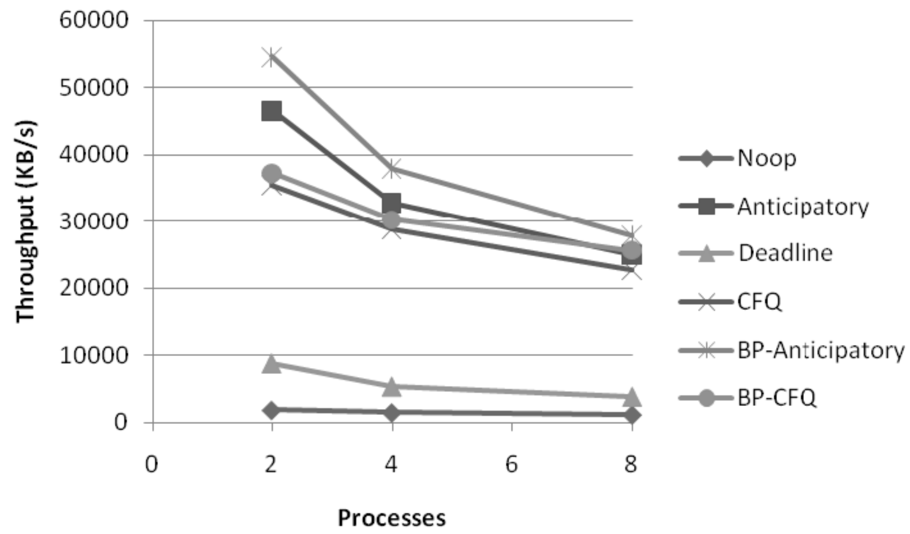

Figure 23 - Iozone Mixed Workload Performance (Higher is Better)

Postmark Benchmark

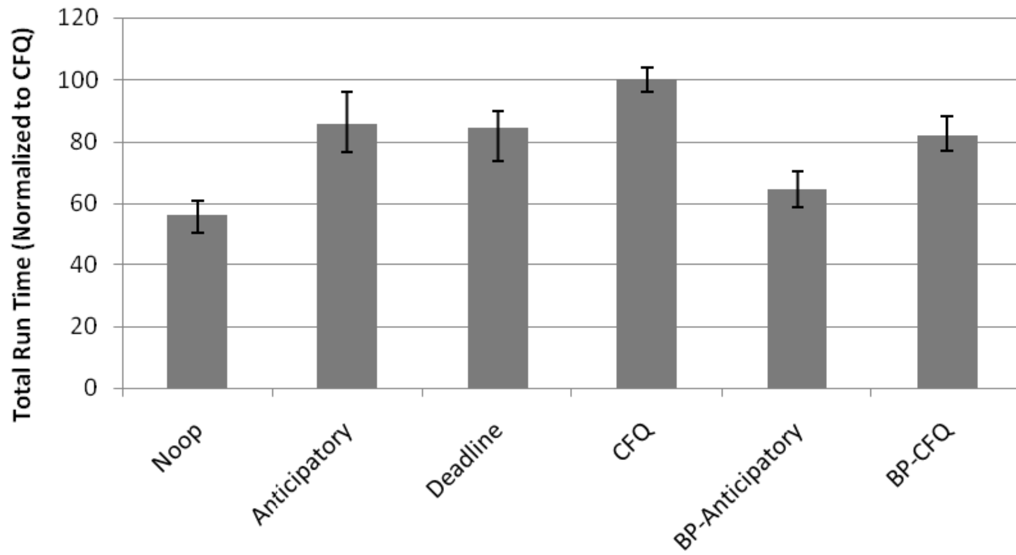

Figure 24 - Postmark Performance (Lower is Better)

We show the normalized performance of different schedulers for the Postmark benchmark workload in Figure 24. The Postmark benchmark performs approximately 25% better with a modified anticipatory scheduler and approximately 18% better with a modified CFQ scheduler, when compared to their respective base schedulers. Neither of the new schedulers achieve the same performance as the noop scheduler, however, they have fairness and starvation prevention measures in place making them more useful scheduler for mixed workloads as seen earlier.

The results of Postmark benchmark are shown for all the three drives in Figure 25. Performance of the schedulers for each drive are normalized to the performance of the CFQ scheduler for that drive. The block-preferential schedulers perform significantly better than their standard counterparts on all the drives. Although noop consistently gives the best performance for this workload across all drives, the block preferential schedulers come close to noop in performance. The performance improvements of our modified schedulers range from 18% for the Transcend drive, to 13% for Memoright drive to 11% for the Samsung drive over their respective base schedulers.

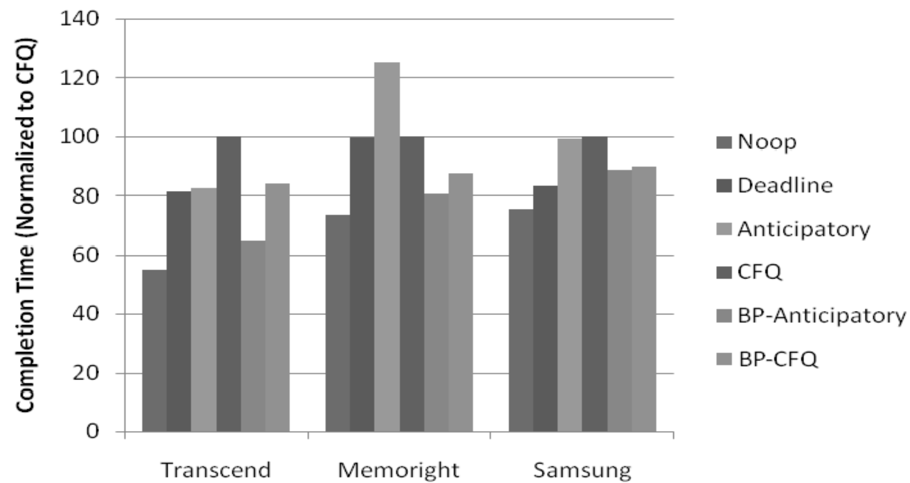

Figure 25 - Postmark Summary (Lower is Better)

Dbench Performance

The dbench benchmark simulates the load of the industry-standard netbench program used to rate Windows file servers [24]. Dbench is used as an example file system benchmark in our tests. Due to the vastly different performance of the drives on this benchmark, we have normalized their performance to that of the unmodified CFQ scheduler, the default for Linux.

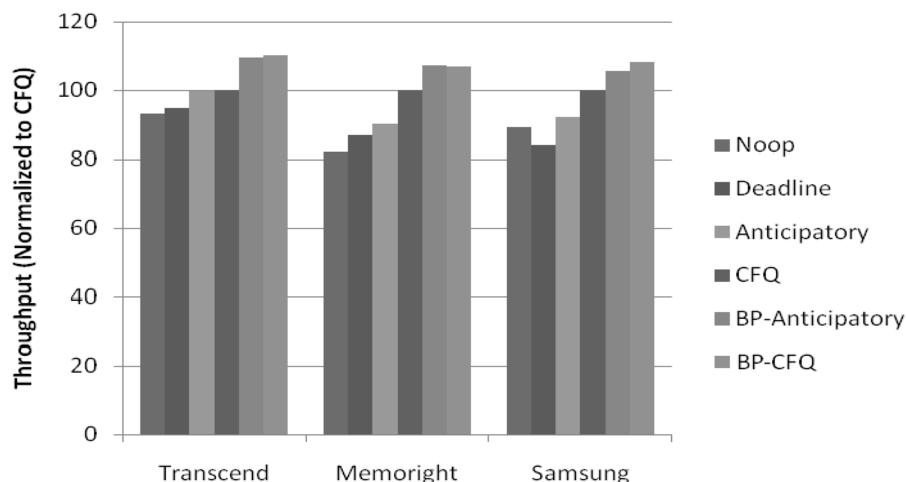

Figure 26 - Dbench Performance (Higher is Better)

Figure 26 shows the normalized performance of different schedulers across all the three SSDs. For dbench's workload, the modified flash schedulers perform the best of all the schedulers, giving a 7-9% increase in performance over the unmodified CFQ scheduler and up to a 25% performance increase against the noop scheduler. The Anticipatory-BF scheduler improves performance by 17% over the native anticipatory scheduler for the Memoright drive. It is also noted that the CFQ and anctipatory schedulers outperform the noop scheduler for this workload.

OLTP Benchmark

The Sysbench OLTP benchmark was created to test system performance on a real database workload [18]. A Mysql database is created on the disk to be tested and a series of complex transactional requests is performed on it. The results of this benchmark on the different drives are shown in Figure 27.

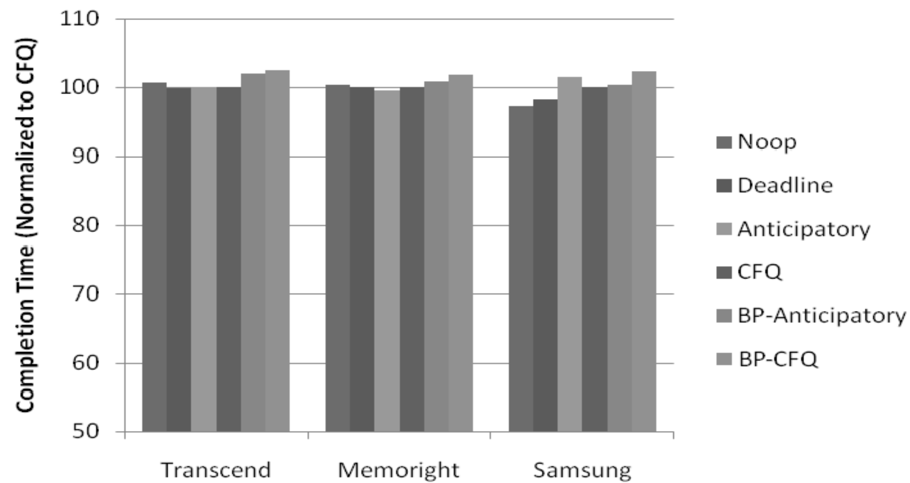

Figure 27 - OLTP Database Performance (Lower is Better)

The choice of scheduler in this workload does not affect transactional performance in a large way to the positive or negative. However, even though the performance loss is minimal, the block preferential schedulers perform between 2-3% lower than the standard schedulers.

NILFS Performance

A very common solution to the random write problem on solid state devices is the implementation of a log-structured file system. To show that our scheduler is not negatively impacted by the presence of a log-structured file system, we present Dbench benchmark results with the NILFS2 file system in Figure 28.

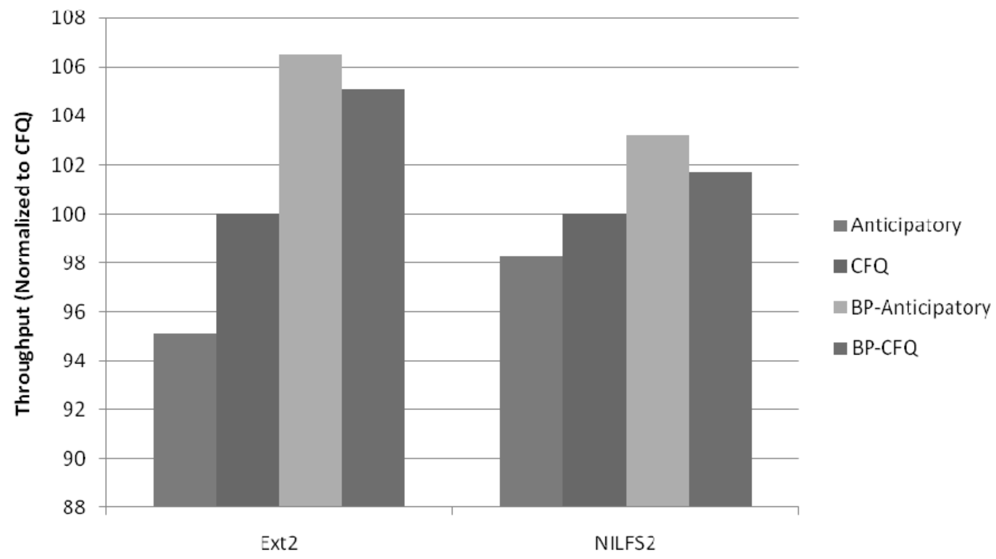

Figure 28 - Dbench Performance on NILFS2 File System. (Higher is Better)

Our scheduler still results in a 2-3% gain over the stock schedulers even in the presence of a log-structured file system.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

As demonstrated by our benchmark results, some of the assumptions made by the current I/O schedulers are outdated when addressing solid state storage and some can even negatively impact performance.

In this paper, we have shown that the I/O schedulers designed for magnetic disk drives perform very differently when employed with SSDs. The assumptions made by the current I/O schedulers are shown to be not appropriate for SSDs. We presented a framework for determining some very important parameters of a given solid state device. We showed that the block size of SSDs has an impact on the write performance of the SSDs.

By analyzing this data and what it tells us about how the drive structures its write requests, we presented a modified I/O scheduler which gives performance gains of up to 25% in some workloads. As flash drives continue to evolve, we hope to apply the tools we have developed here to design improved approaches to I/O scheduling which take all of a device's characteristics into account for the maximum possible performance.

In designing our new I/O scheduler, we have built on top of existing I/O schedulers which were designed for traditional hard drives and which were included in the Linux kernel. In order to fully utilize all of the performance a solid state drive has to offer, it may be necessary to design a new scheduler completely from scratch taking into account all we know about solid state devices and their unique properties. Uninhibited

by previous design decisions, it may be possible for even greater performance gains to be achieved.

REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. of USENIX Annual Technical Conference*, Boston, 2008, pp. 57-70.

[2] S. Baek, S. Ahn, J. Choi, D. Lee, S. Noh, "Uniformity improving page allocation for flash memory file systems," in *Proc. of ACM EMSOFT*, Salzburg, Austria, 2007, pp. 154-163.

[3] P. Barrett, S. Quinn and R. Line, "System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers," U.S. Patent 5,392,427. 21 February 1995.

[4] A. Birrell, M. Isard, C. Thacker and T. Webber, "A design for high-performance flash disks," in *ACM SIGOPS SOSP*, Stevenson, WA, 2007, pp. 88-93.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Proc. of IEEE Int. Conf. on Multimedia Computing and Systems*, Florence, Italy, 1999, pp. 400-405.

[6] D. Capps, "IOZONE Filesystem Benchmark," available at http://www.iozone.com/. Accessed in April 2009.

[7] A. Caulfield, L. Grupp and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," in *ACM ASPLOS*, Washington, DC, 2009, pp. 217-228.

[8] L. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proc. of ACM SAC Conference*, Seoul, Korea, 2007, pp. 1126-1130.

[9] E. Gal and S. Toledo, "A transactional flash file system for microcontrollers," in *Proc. of USENIX Annual Technical Conference*, Anaheim, CA, 2005, pp. 89-104.

[10] A. Gupta, Y. Kim and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *ACM ASPLOS*, Washington, DC, 2009, pp. 229-240.

[11] D. Hitz, J. Lau and M. Malcolm, "File system design for an NFS file server appliance," in *Proc. of USENIX Annual Technical Conference*, San Francisco, 1994, pp. 235-246.

[12] S. Iyer and P. Druschel, “Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O,” in *Proc. of ACM SOSP*, Banff, AB, 2001, pp. 117-130.

[13] S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in *Proc. of ACM SIGMETRICS*, Marina Del Rey, CA, 2003, pp. 31-42.

[14] J. Katcher, “Postmark: A new filesystem benchmark,” *Technical Report TR3022*, Network Appliance, 1997.

[15] T. Kgil, D. Roberts and T. Mudge, “Improving NAND flash based disk caches,” in *Proc. of ACM Int. Symp. On Computer Architecture*, Beijing, China, 2008, pp. 327-338.

[16] H. Kim and S. Ahn, “BPLRU: A buffer management scheme for improving random writes in flash storage,” in *Proc. of USENIX FAST Conference*, San Jose, CA, 2008, pp. 239-252.

[17] I. Koltsidas and S. Viglas, “Flashing up the storage layer,” in *Proc. of VLDB*, Auckland, New Zealand, 2008, pp. 514-525.

[18] A. Kpytov, “SysBench OLTP benchmark,” available at http://sysbench.sourceforge.net/. Accessed in April 2009.

[19] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, S. Noh, “Block recycling schemes and their cost-based optimization in NAND flash memory based storage systems,” in *Proc. of ACM EMSOFT*, Salzburg, Austria, 2007, pp. 174-182.

[20] C. Manning, “YAFFS: A NAND-flash filesystem,” in *FOSDEM 2004,* Brussels, Belgium, 2004.

[21] N. Meggido and D.Modha, “ARC: A self-tuning, low overhead replacement cache,” in *Proc. of USENIX FAST*, San Francisco, CA, 2003, pp. 115-130.

[22] E. Miller, S. Brandt and D. Long, “High performance reliable MRAM enabled file system,” in *Proc. of USENIX HOTOS*, Elmau, Germany, 2001, pp. 83-87.

[23] J. Ousterhout and F. Douglis, “Beating the I/O bottleneck: A case for log-structured file systems,” in *ACM SOSP,* Litchfield Park, AZ, 1989, pp. 11-28.

[24] V. Prabhakaran, T. Rodeheffer and L. Zhou, “Trasactional flash,” in *Proc. of USENIX OSDI Conference*, San Diego, CA, 2008, pp. 147-160.

[25] P. Reiher, A. Wang, G. Kuenning and G. Popek, “The Conquest file system: Better performance through disk/persistent-ram hybrid design,” in *ACM Transactions On Storage 2006*, Vol. 2, pp. 309-348, 2006.

[26] M. Seltzer, K. Bostic, K. McKusick and C. Staelin, “An implementation of a Log-structured file system for UNIX,” in *Proc. of USENIX*, San Diego, CA, 1993, pp. 3-3.

[27] P. Shenoy and H. Vin, “Cello: a disk scheduling framework for next generation operating systems,” in *Proc. of 1998 ACM SIGMETRICS*, Madison, WI, 1998, pp. 44-55.

[28] Texas Instruments, “Internal architecture of SSD,” available at http://focus.ti.com/graphics/blockdiagram/blockdiagram_images/6260.gif. Accessed in May 2009.

[29] A. Tridgell, “Dbench file system benchmark,” available at ftp://samba.org/pub/tridge/dbench/. Accessed in April 2009.

[30] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, R. Rangaswami, “EXCES: External caching in energy saving storage systems,” in *Proc. of HPCA*, Salt Lake City, UT, 2008, pp.89-100.

[31] R. Wijayaratne and A. L. N. Reddy, “System support for providing integrated services from networked multimedia storage servers,” in *Proc. of ACM Multimedia Conference*, Ottawa, Canada, 2001, pp. 270-279.

[32] D. Woodhouse, “JFFS: The Journaling Flash File System,” in *Ottawa Linux Symposium,* Ottawa, Canada, 2001.

[33] B. Worthington, G. Ganger and Y. Patt, “Scheduling algorithms for modern disk drives,” in *Proc. of 1994 ACM SIGMETRICS*, Nashville, TN, May 1994, pp. 241-251.

[34] B. Worthington, G. Ganger, Y. Patt and J. Wilkes, “On-line extraction of SCSI disk drive parameters,” in *Proc. of 1995 ACM SIGMETRICS*, Ottawa, Canada, 1995, pp. 146-156.

[35] M. Wu and W. Zwaenpoel, “eNVY: A non-volatile, main memory storage system,” in *Proc. of ACM ASPLOS*, San Jose, CA, 1994, pp. 86-97.

VITA

Marcus Paul Dunn received his Bachelor of Science degree in Computer Engineering from Texas A&M University in College Station in 2006. While at Texas A&M he was a National Merit Scholar and recipient of the President's Endowed Scholarship and the Director's Achievement Scholarship. He entered the Computer Engineering graduate program at Texas A&M University in January 2007 and graduated with his Master of Science degree in August 2009. His research interests while at Texas A&M involved storage devices, network security, and solid state disks. Starting in June, Mr. Dunn will be working for Cisco Systems, Inc. Mr. Dunn can be reached at 214 Zachry Engineering Center, TAMU 3128, College Station, TX 77843-3128.