A New I/O Scheduler for Solid State Devices

Marcus Dunn and A. L. Narasimha Reddy
Department of Electrical and Computer Engineering
Texas A&M University
College Station, TX 77843-3259, USA
{marcusd, reddy}@ece.tamu.edu

Abstract— **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 paper, 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.**

Keywords-I/O Schedulers, Flash Drives.

1. INTRODUCTION

In recent years, enormous advances in the area of solid state storage devices have begun to attract wide attention[15,18,19,20,22]. 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. Flash drives also employ a Flash Translation Layer (FTL), which can have unforeseen effects on drive performance. Many different FTL schemes have been proposed, but most vendors keep their implementations proprietary, which further complicates designing for solid state drives.

Many file system approaches have been designed to try and take advantage of the specific characteristics of solid state drives[7,9,10]. 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 this paper, we address the differences of SSD characteristics at the I/O scheduler layer. Current I/O schedulers take seek time, rotational latency overheads of magnetic disk drives into account in scheduling requests[1,2,3,4,6]. 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.

SSDs are organized into multiple banks that can be independently accessed [11,16,17,21]. 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. 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[11,12,13,14].

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[11,16,17], 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 flash transition layers add additional functionality. The advanced devices will have translation layers which implement some form of error correction, wear leveling, and block/page management algorithm. The presence of this layer and the varying implementations make it difficult for a simple device characterization, a problem which we will address in a later section.

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 this paper, we 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.

Secondly, 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 Section 2 we outline the current approach to I/O schedulers and some of the assumptions which are made about the physical storage devices. In Section 3, we present our framework and tests which we used to model the performance of flash disks. In Section 4, we present our approach to scheduler design based on some of the results from Section 3. Section 5 presents our implementation and benchmark results. Finally, our conclusions and ideas for possible future work are presented in Section 7.

2. I/O SCHEDULERS

2.1 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 handles 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.

2.2 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. Noop scheduler supports request merging where an incoming request may be merged with an already queued contiguous 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.

2.3 Deadline scheduler

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 secondly, 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 magnetic disk drives. In addition, by guaranteeing a start service time, the deadline scheduler can avoid starvation that may occur in the noop scheduler.

2.4 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. After serving a process's read 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.

Anticipatory scheduling attempts to compensate for this by pausing for a short time after a read operation, in essence anticipating that the next read operation may occur close by. Anticipatory scheduling also employs 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.

2.4 Completely Fair Queueing

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 queues 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.

Another feature of note in the CFQ scheduler is that it also adopts the "one-way" elevator and makes some of its scheduling decisions based on the notion of a rotating disk at the physical layer.

2.5 Scheduler Comparison

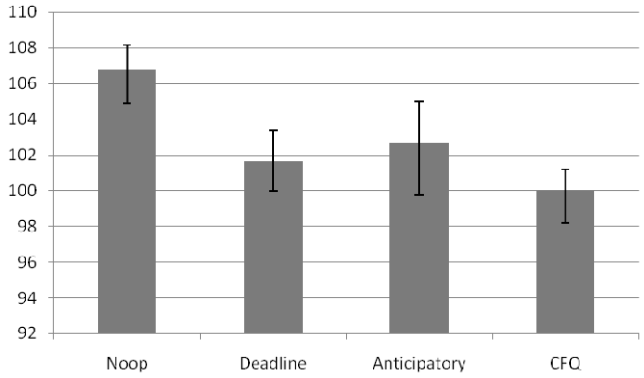

Figure 1 Scaled Comparison of Schedulers on HDD (Lower is better)

Figure 1 shows a comparison of the four default Linux schedulers on a sample Postmark workload [8] 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 2 shows the performance of the schedulers on the same workload on a SSD. Figure 2 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 1.

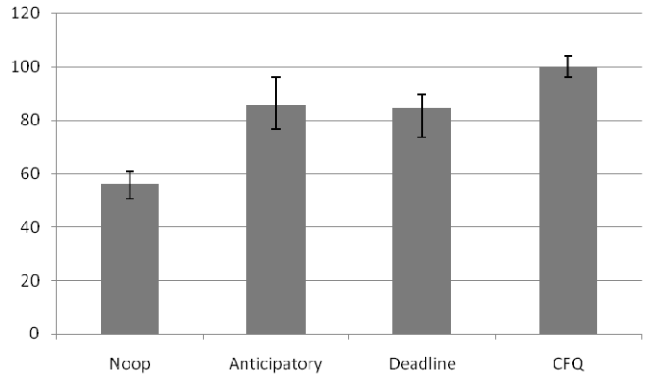

Figure 2 – Scaled Comparison of Schedulers on SSD (Lower is Better)

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.

3. FLASH DRIVE CHARACTERISTICS

3.1 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. 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 (space doesn't allow us to provide all the details from all the drives).

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

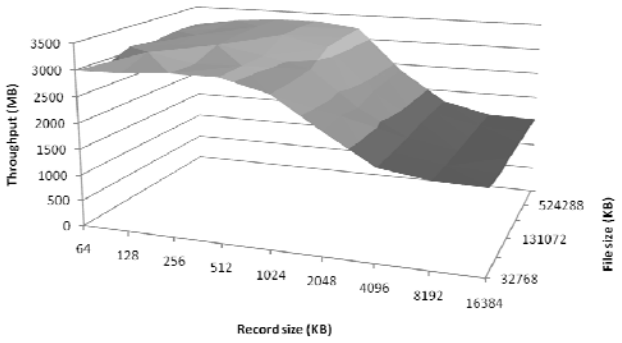

Figure 3 Sequential Read Performance

Figure 3 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.

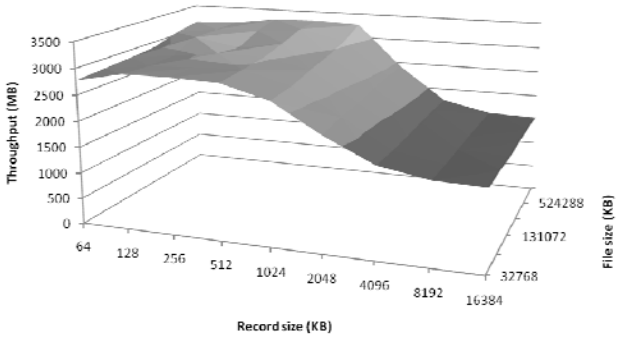

Figure 4 Random Read Performance

Figure 4 shows a surface curve of the random read performance of the drive. A comparison of the two graphs in figures 1 and 2 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.

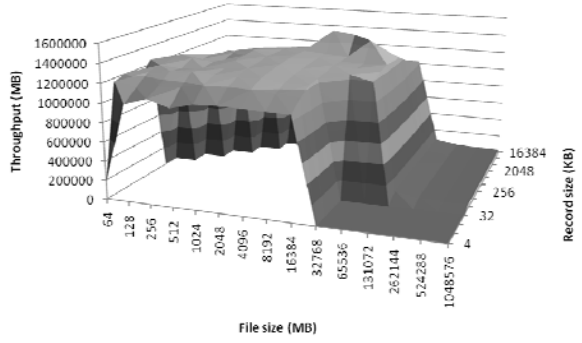

Figure 5 Sequential Write Performance

Figure 5 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 the 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.

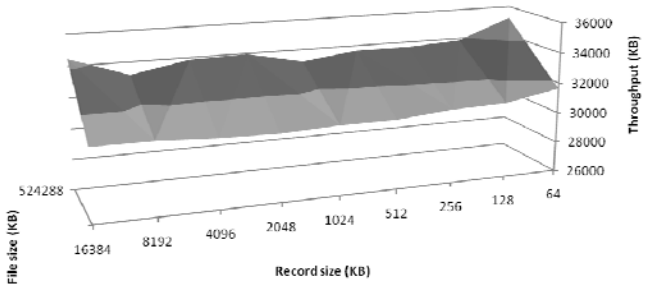

Figure 6 Sequential Write Performance (Detail)

In Figure 6, 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.

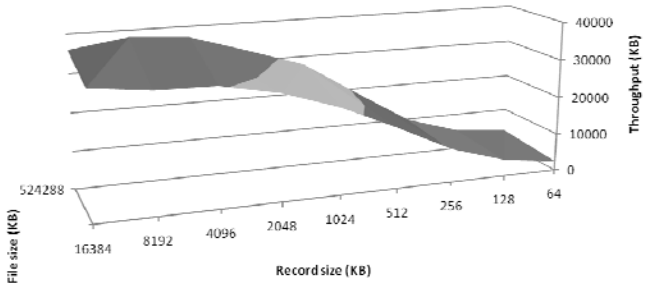

Figure 7 Random Write Performance (Detail)

Figure 7 shows the same region for random write operations. In figure 7, 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.

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.

3.2 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 seeking 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.

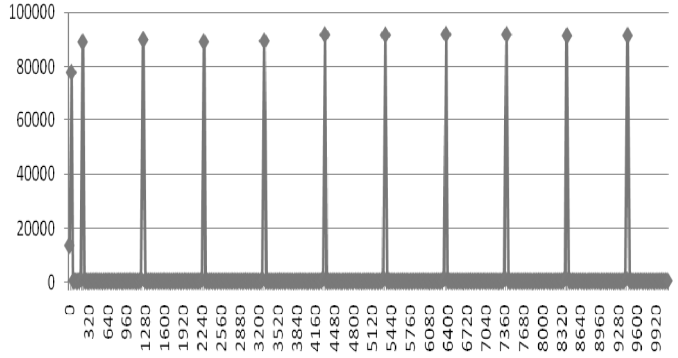

Figure 8 32KB stride test

Figure 8 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 structuring 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.

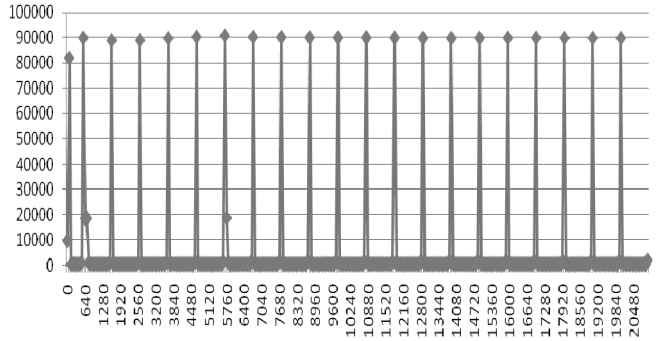

Figure 9 64KB Stride Test

Figure 9 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. We confirmed this to be the case with larger stride tests (data is not presented here due to space reasons).

The write performance of the drive is shown in Figure 10. 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.

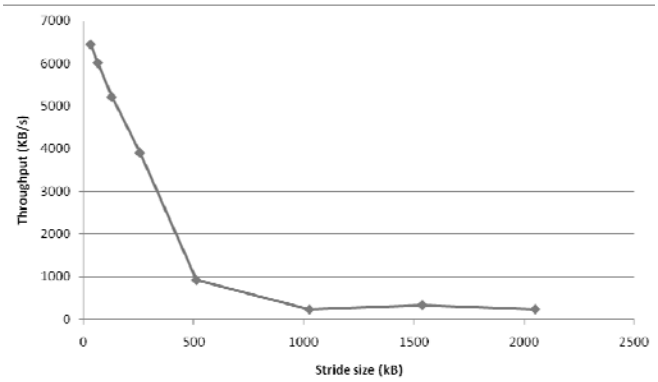

Figure 10 Striding Test Summary for Transcend

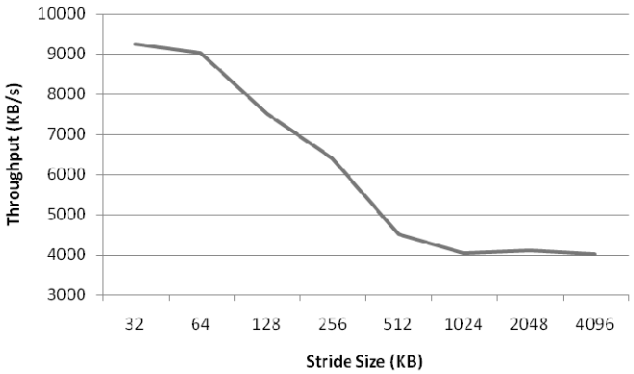

Figure 11 Striding Test for Memoright Drive

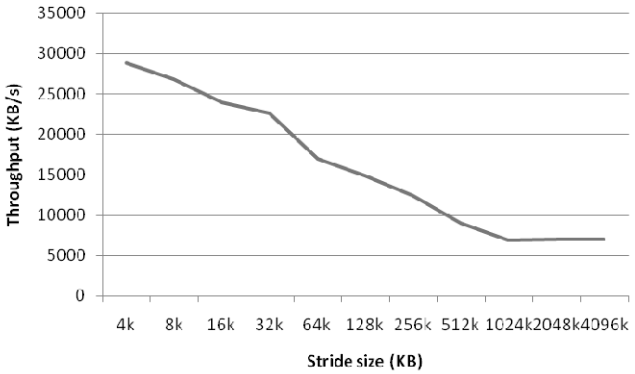

Figure 12 Striding Test for Samsung Drive

We have carried out similar tests on the other two drives from Memoright and Samsung. The results from these tests are presented in Figures 11 and 12. 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).

4. NEW SCHEDULER DESIGN

4.1 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 new block writing 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 backwards seeks and forward seeks as the schedulers do for rotational

disks. Without a rotational component, forward and backwards 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 page. 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.

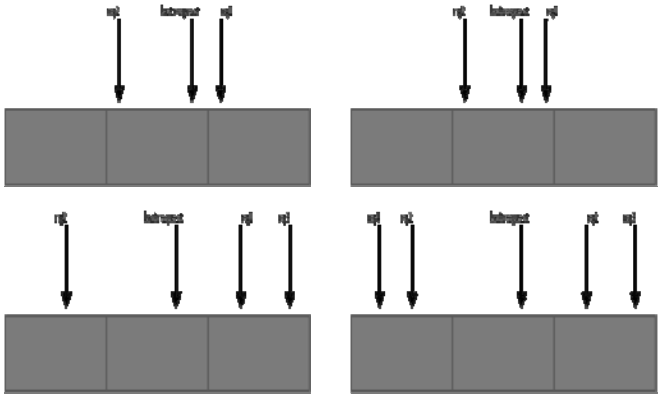

Figure 13 Four request positioning cases (1-4 clockwise from top-left)

The four possible scenarios of request positioning are illustrated in Figure 13.

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. Even saving a few of the grossly expensive boundary crossing costs is well worth the use of the new scheduler in many applications.

5. 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.

We performed several different tests during evaluation. We present data from most relevant tests here.

5.1 Iozone sequential tests

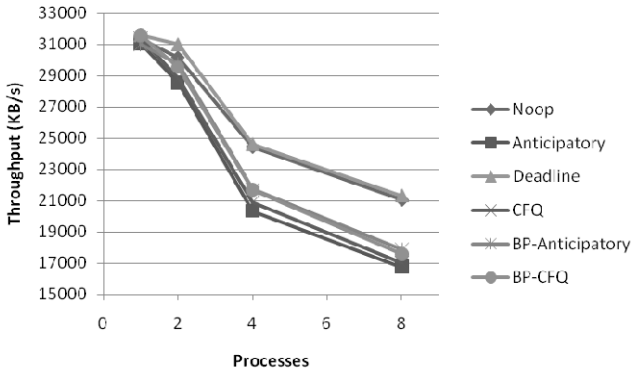

Figure 14 Iozone Sequential Write Performance

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 14, 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.

5.2 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 15.

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.

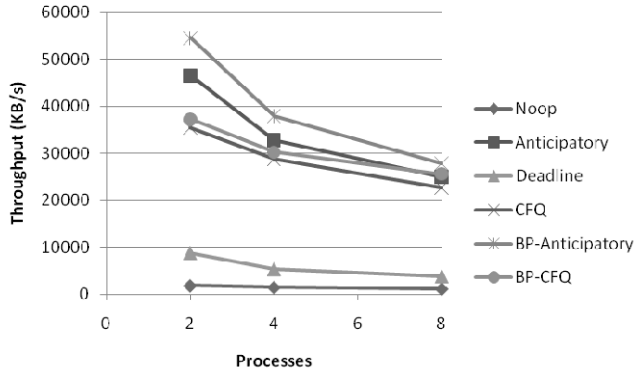

Figure 15 Iozone Mixed Workload Performance

5.3 Postmark benchmark

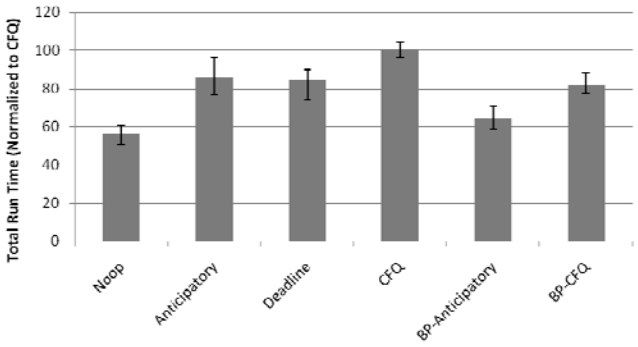

Figure 16 Postmark Performance (Lower is better)

As seen in Figure 16, 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, the BP-CFQ scheduler also has fairness and starvation prevention measures in place which make it a much more useful scheduler for everyday performance.

The results of Postmark benchmark are shown for all the three drives in Figure 17. 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 drives. Although noop consistently gives the best performance for this workload across all drives, the block preferential schedulers come in just under noop in performance. The performance improvements range from 18% for the Transcend drive, to 13% for Memoright drive to 11% for the Samsung drive.

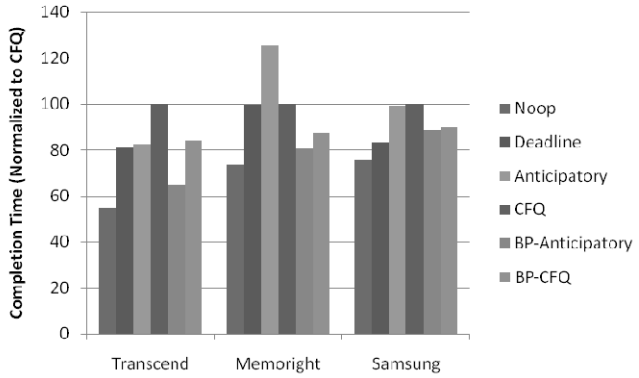

Figure 17 Postmark Summary (Lower is better)

5.4 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.

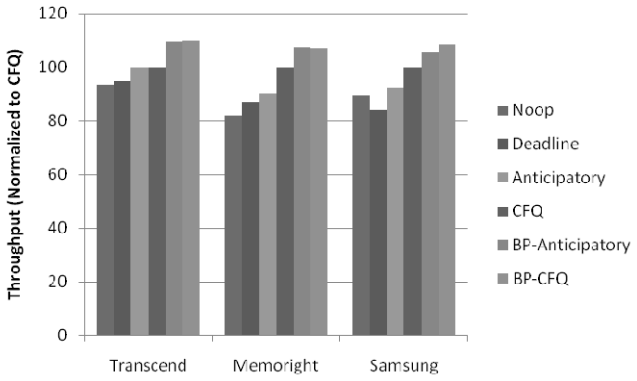

Figure 18 Dbench performance (Higher is better)

Figure 18 shows the normalized peroformance 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.

5.5 OLTP Benchmark

The Sysbench OLTP benchmark was created to test system performance on a real database workload [25]. 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 19.

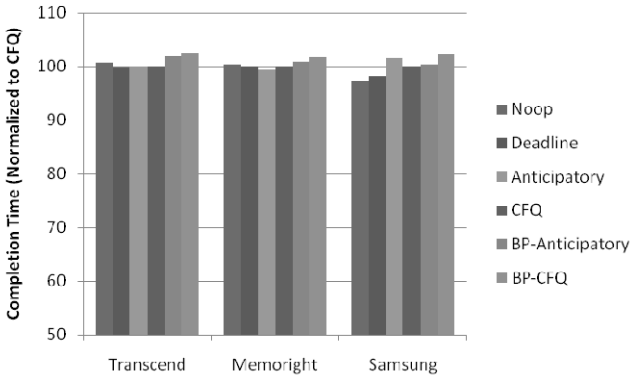

Figure 19 OLTP Database Performance (Lower is better)

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.

5.6 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 bencmark results with the NILFS2 file system in Figure 20.

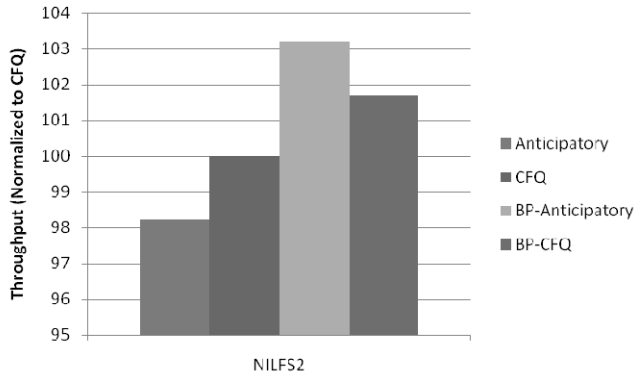

Figure 20 Dbench performance on NILFS2 file system. (Higher is better)

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

RELATED WORK

Much work has been done in file systems, I/O scheduling to take characteristics of magnetic disk drives into account. This work ranges from allocation decisions on disk, read/write caching and scheduling. Log-structured file systems were proposed as an alternative to update-in-place file systems [26,27,5]. Alternate file systems have been proposed for non-volatile memories and solid state devices [7,9,10].

I/O scheduling has received wide attention for magnetic disk drives. Many classical approaches of disk arm optimization have been studied in the literature. Anticipatory scheduling has been proposed to reduce the deceptive idleness in synchronous I/O operations [1]. Multimedia workloads have spurred considerable work on I/O scheduling to provide guarantees [2,3,4]. Our work here revisits this I/O scheduling problem in light of the increased use of SSDs.

SSDs have recently been studied in different workloads [15, 18-22,28]. The wear leveling, block remapping and garbage collection operations at the FTL level have been investigated [11-14,16]. The improvements and algorithms at the FTL level have resulted in considerable performance and life improvements of SSDs. Log-structured file systems like mechanisms have been incorporated into FTLs of SSDs.

Algorithms for extracting disk parameters have been proposed earlier [29]. Our SSD performance characterization and parameter extraction is motivated by this earlier work.

6. 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.

ACKNOWLEDGMENT

This work is supported by funding from the Department of Defense and NSF.

REFERENCES

[1] S. Iyer and P. Druschel, "Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O", Proc. of ACM SOSP, 2001.

[2] J. Bruno et al, "Disk scheduling with quality of service guarantees", Proc. of IEEE Int. Conf. on Multimedia Computing and Systems, June 1999.

[3] P. Shenoy and H. Vin, "Cello: a disk scheduling framework for next generation operating systems", Proc. of 1998 ACM SIGMETRICS, June 1998.

[4] R. Wijayaratne and A. L. N. Reddy, "System support for providing integrated services from networked multimedia storage servers", Proc. of ACM Multimedia Conf., Sept. 2001.

[5] M. Seltzer, K. Bostic, K. McKusick and C. Staelin, "An implementation of a Log-structured file system for UNIX", Proc. of USENIX, Jan. 1993.

[6] B. Worthington, G. Ganger and Y. Patt, "Scheduling algorithms for modern disk drives", Proc. of 1994 ACM SIGMETRICS, May 1994.

[7] P. Reiher, A. Wang, G. Kuenning and G. Popek, "The Conquest file system: Better performance through disk/persistent-ram hybrid design", ACM Trans. On Storage, 2006.

[8] J. Katcher, "Postmark: A new filesystem benchmark", Technical Report, TR3022, Network Appliance, 1997.

[9] M. Wu and W. Zwaenpoel, "eNVY: A non-volatile, main memory storage system", ACM ASPLOS, 1994.

[10] E. Miller, S. Brandt and D. Long, "High performance reliable MRAM enabled file system", Proc. of USENIX HOTOS, May 2001.

[11] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage", Proc. of USENIX FAST Conf., Feb. 2008.

[12] S. Baek et al, "Uniformity improving page allocation for flash memory file systems", Proc. of ACM EMSOFT, Oct. 2007.

[13] J. Lee et al, "Block recycling schemes and their cost-based optimization in NAND flash memory based storage systems", Proc. of ACM EMSOFT, Oct. 2007.

[14] Li-Pin Chang, "On efficient wear leveling for large-scale flash-memory storage systems", Proc. of ACM SAC Conf., Mar. 2007.

[15] A. Caulfield, L. Grupp and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications", ACM ASPLOS, 2009

[16] A. Gupta, Y. Kim and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings", ACM ASPLOS, 2009.

[17] T. Kgil, D. Roberts and T. Mudge, "Improving NAND flash based disk caches", Proc. of ACM Int. Symp. On Computer Architecture, June 2008.

[18] A. Birrell, M. Isard, C. Thacker and T. Webber, "A design for high-performance flash disks", ACM SIGOPS Oper. Syst. Review, 2007.

[19] I. Koltsidas and S. Viglas, "Flashing up the storage layer", Proc. of VLDB, Aug. 2008

[20] V. Prabhakaran, T. Rodeheffer, and L. Zhou, "Trasactional Flash", Proc. of USENIX OSDI Conf., 2008.

[21] N. Agrawal et al, "Design tradeoffs for SSD performance", Proc. of USENIX Annual Technical Conf., June 2008.

[22] E. Gal and S. Toledo, "A transactional flash file system for microcontrollers", Proc. of USENIX Annual Tech. Conf. April 2005.

[23] IOZONE benchmark, www.iozone.org, accessed 2008.

[24] Dbench file system benchmark, ftp://samba.org/pub/tridge/dbench/

[25] A. Kpytov, "SysBench OLTP benchmark ", http://sysbench.sourceforge.net/, 2006.

[26] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems", ACM SOSP 1989.

[27] D. Hitz, J. Lau and M. Malcolm, "File system design for an NFS file server appliance", Proc. of USENIX Annual Tech. conf., 1994.

[28] L. Useche et al, "EXCES: External caching in energy saving storage systems", Proc. of HPCA, Feb. 2008.

[29] B. Worthington, G. Ganger, Y. Patt and J. Wilkes, "On-line extraction of SCSI disk drive parameters", ACM SIGMETRICS 1995.