1st Workshop on Integrating Solid-state Memory into the Storage Hierarchy,
March 7, 2009, Washington DC.

Enabling Enterprise Solid State Disks Performance

Milo Polte, Jiri Simsa, Garth Gibson
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA, 15213

***Abstract*—In this paper, we examine two modern enterprise Flash-based solid state devices and how varying usage patterns influence the performance one observes from the device. We observe that in order to achieve peak sequential and random performance of an SSD, a workload needs to meet certain criteria such as high degree of concurrency. We measure the performance effects of intermediate operating system software layers between the application and device, varying the filesystem, I/O Scheduler, and whether or not the device is accessed in direct mode. Finally, we measure and discuss how device performance may degrade under sustained random write access across an SSD's full address space.**

I. Introduction

Flash storage devices are a promising new technology which have engendered much excitement for their potential to fill the memory hierarchy gap between the access times of DRAM and mechanical disks [1]. However, how well they work in real system with complex workloads is not so clear. In addition, the performance of Flash devices and their pice is changing rapidly, sometimes monthly [2] [3], yet accurate assumptions about Flash behavior, including price and performance, is important to correctly design a hybrid system. For example, some authors assume Flash has poor random write performance [4].

A Flash device is made out of memory cells that can be read anytime, but before a non-empty cell can be written it must be first erased. The write and erase operations are considerably slower than reading. In addition, the erase operation is destructive and, based on the particular Flash technology, a cell can sustain between 10,000 to 1,000,000 erase cycles before wearing out. Flash cells are grouped into pages (typically 4 KB), which are the atomic units of access. Pages are grouped a block (typically 256 KB), which is the atomic unit of erasing [5].

Although the read and write operations take in the order of tens of microseconds, the erase operations take milliseconds. Therefore, using Flash naively would not provide for high random write performance.

As Flash technology matures, techniques such as log-based writing with background cleaning and wear-leveling, over-provisioning and multiple data and control channels were adopted to enable high random I/O performance.

In this paper, we examine two modern enterprise Flash solid state devices and how varying usage patterns influence the performance one observes from the device. We consider the effects of specific operating system storage software on top of the devices, the role of parallelism, and the consequences of sustained random write access on device performance. We chose to measure these specific devices as they represent current state-of-the-art Flash-based solid state devices with fast random write performance. Measurements for older devices and magnetic disks can be found in [2].

The following section describes our experimental setup. Section III presents performance measurements. Finally, this study is concluded in Section IV.

II. Experimental Setup

All experiments were run on a server with 4 GB of RAM and an Intel Pentium D Dual Core 3.0GHz processor with an 8x PCIe slot and 3 Gb/s SATA interconnect. The operating system used for these experiments was the Ubuntu Linux Distribution (version 8.04) [6] running the default 2.6.24-19-generic Linux kernel. In this paper, all filesystem and I/O scheduler versions correspond to those included in this kernel.

We examined two different Flash based solid state devices (SSDs) for these experiments. Their interconnect types and advertised performance characteristics are summarized in Table I. Both of these devices were released in 2008 and display superior performance to previous generation SSDs, especially in the area of random writes [2].

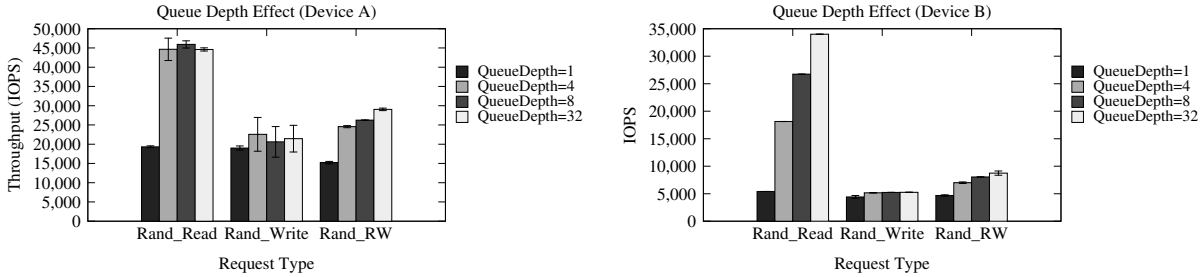
Results were obtained from two benchmarking tools: IOZone version 3.279 [7] and the Flexible IO Tester (`fio`) version 1.22 [8]. IOZone is a filesystem benchmark generally used to write to a single file on top of a filesystem using read() and write() calls, although it is configurable for different access patterns and concurrency. The Flexible IO Tester is another highly configurable benchmark program, recommended to us by an SSD vendor, that allows the user to control the number of I/O jobs to be run concurrently on top of files or raw devices. Although both tools are configurable for a variety of access patterns, we found it more straightforward to use IOZone for our benchmarks that worked through the filesystem and the Flexible IO Tester for benchmarks that accessed the raw device. The configurations of these benchmarks for each test appear in the relevent result subsections below. In figures where error bars are shown, the results represent the average of 5 runs unless stated otherwise.

III. Measurements

A. Impact Of Queue Depth

Consider a simple model of a Flash SSD as a disk with zero legnth "seeks". Such a device would achieve its maximum I/O accesses per second (IOPS) for a random workload regardless of whether those requests are issued serially or many at once.

FIGURE 1. Queue Depth Effect



Label	Device A	Device B
Interface Type	PCIe SSD	SATA SSD
User Capacity	80 GB	32 GB
List Price	$2400	$810
$/GB	$30	$25.31
Access Time	0.05 msec	0.085 msec
Advertised Bandwidth	700 MB/s read 550 MB/s write	250 MB/s read 170 MB/s write
Advertised Peak IOPS	102k IOPS read 91k IOPS write	35k IOPS read 3.3k IOPS write

TABLE I
SUMMARY OF SOLID STATE DISKS

In practice, however, modern Flash devices can consist of multiple concurrently addressable banks, all of which would have to be busy in parallel to achieve peak performance.

To describe the impact of parallel access, we configured the `fio` benchmark to maintain a variety of queue depths (number of outstanding I/O requests that are active at the device at a given time) for three different access patterns (direct random reading from the entire device address space, direct random writing to the entire device address space, and a 50/50 mix of the two). In all cases, the raw device is accessed "directly" rather than through a mounted filesystem on the device. All accesses were for 4 KB blocks. The experiment was run for 10 minutes for each configuration and IOPS measured. Results are shown in Figure 1.

On random reads, Device A achieves high performance once the I/O queue depth reaches 4 concurrent requests and on Device B, we see IOPS throughput increasing with deeper queue depths from 1 to 32. We hypothesize that the devices have different numbers of banks for performing parallel operations. Our understanding is that Device A has 4 banks whereas Device B has 10 banks, consistent with our hypothesis.

On random writes, performance does not vary widely with queue depth. We speculate that written data is experiencing less queue depth dependency because the device is delaying the actual write and doing it in the background, and remapping a location of its choice, allowing it to fully utilize all banks. The performance of the writes is rather low compared to the reads, which is a characteristic of Flash as mentioned in Section I. Later in Section III-F we will also discuss the negative performance impact of writing randomly to the entire address space of a Flash device for long periods of time. In the mix workload of reads and writes, we see some improvements in performance with more queue depth, because of the dependence of queue depth for reads.

In order to achieve the best performance for random reads of small request sizes, a system must maintain a queue of outstanding requests that fully utilizes the device, the depth of which may vary from device to device. This may not always be possible for a workload that issues small read requests from a small number of threads.

It is noteworthy for designers of hybrid systems that we did not initially see the performance of Device B increasing with queue depth. We had to update operating system software and explore configurations until the correct kernel module was loaded, the device's Native Command Queueing was enabled, and the server's BIOS configured to treat SATA devices in "native" rather than "compatible" mode. Significant end user tuning may be required in order to get high performance from rapidly evolving Flash devices.

B. Storage Software Layers

Next, we evaluate the effect of using different storage software to issue I/O requests to the device. To this end we consider three attributes to vary—the file system, the I/O block request scheduler, and the I/O mode (between non-direct—that is, allowing prefetching and write behind in the filesystem's buffer cache—and direct). A file system is a software layer between an application and the storage device that accesses and manages persistent data. The file systems considered in this study are:

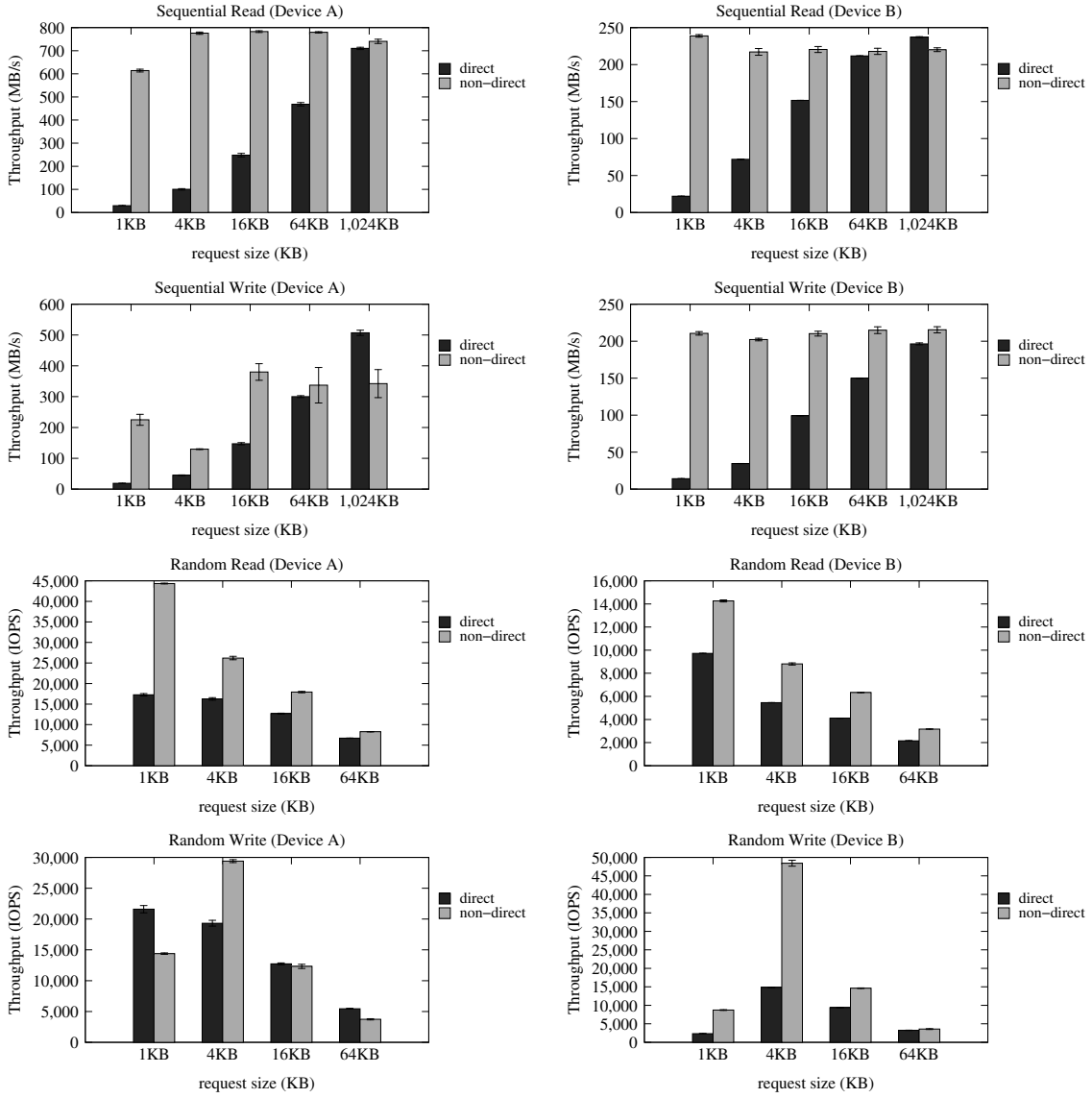
1) XFS – a high performance journaling file system
2) ReiserFS – the first journaling file system in Linux
3) ext3 – the standard journaling Linux file system
4) ext2 – the standard pre-journaling Linux file system

All file system versions are those contained in kernel version 2.6.24-19-generic from Ubuntu Linux.

An I/O scheduler is an algorithm used inside the block layer of an operating system to issue requests to a block device. The I/O schedulers considered are:

1) noop – a single FIFO queue for all requests with minimal coalescing. As long as it has pending I/O requests, the noop scheduler issues requests as fast as the operating system and device allows.

FIGURE 2. Impact on Performance of Varying I/O Modes



2) anticipatory – reorders requests based on patterns observed in the past. The anticipatory scheduler may pause before issuing a pending request if it 'anticipates' that a new request that is more sequential to the previously completed request will arrive within a few milliseconds.
3) cfq – keeps per-process FIFO queue and allocates time-slices to each process based on their priority.

Further discussion of these schedulers and their performance characteristics on disks is found in [9].

An I/O mode is simply a path through the operating system that an I/O request takes. The I/O modes considered are:

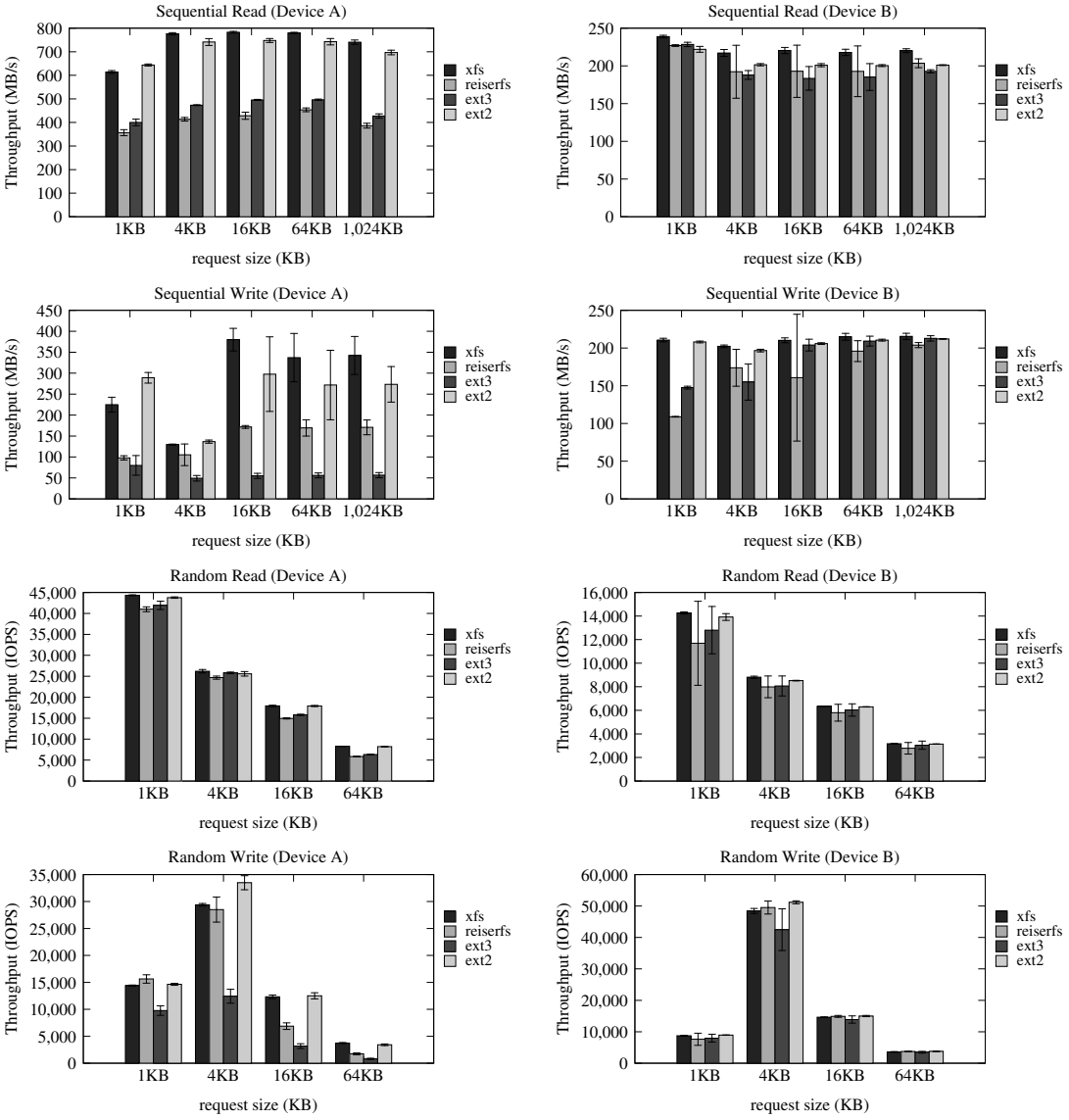
1) non-direct – requests go through the buffer cache, which enables read-prefetching or write-behind, time shifting when device accesses occur
2) direct – requests bypass buffer cache and thus use neither read-prefetching nor write-behind on the way to the device

To evaluate the effects of varying each software layer option, we vary it while keeping the other layer options fixed and use the IOZone benchmark to measure performance for sequential and random I/O. For each experiment the `iozone` benchmark is run with the following parameters:

- `-R` generate a report for all specified request sizes
- `-r 1k -r 4k -r 16k -r 64k -r 1m` use 1KB, 4KB, 16KB, 64KB and 1MB request sizes
- `-i 0` run sequential read and write test
- `-i 1` run random read test
- `-i 2` run random write test
- `-s 4g` use 4GB for the size of the test file
- `-U` remount the filesystem before each test

In particular, every test is performed repeatedly for request sizes specified above, and the file system was remounted with no special options before every test. Note that run in this

FIGURE 3. Impact on Performance of Varying the File System



fashion, IOZone issues a single request at a time, similar to the behavior of a queue depth of one in Section III-A above.

In the following subsections we discuss the impact on performance of varying each software layer.

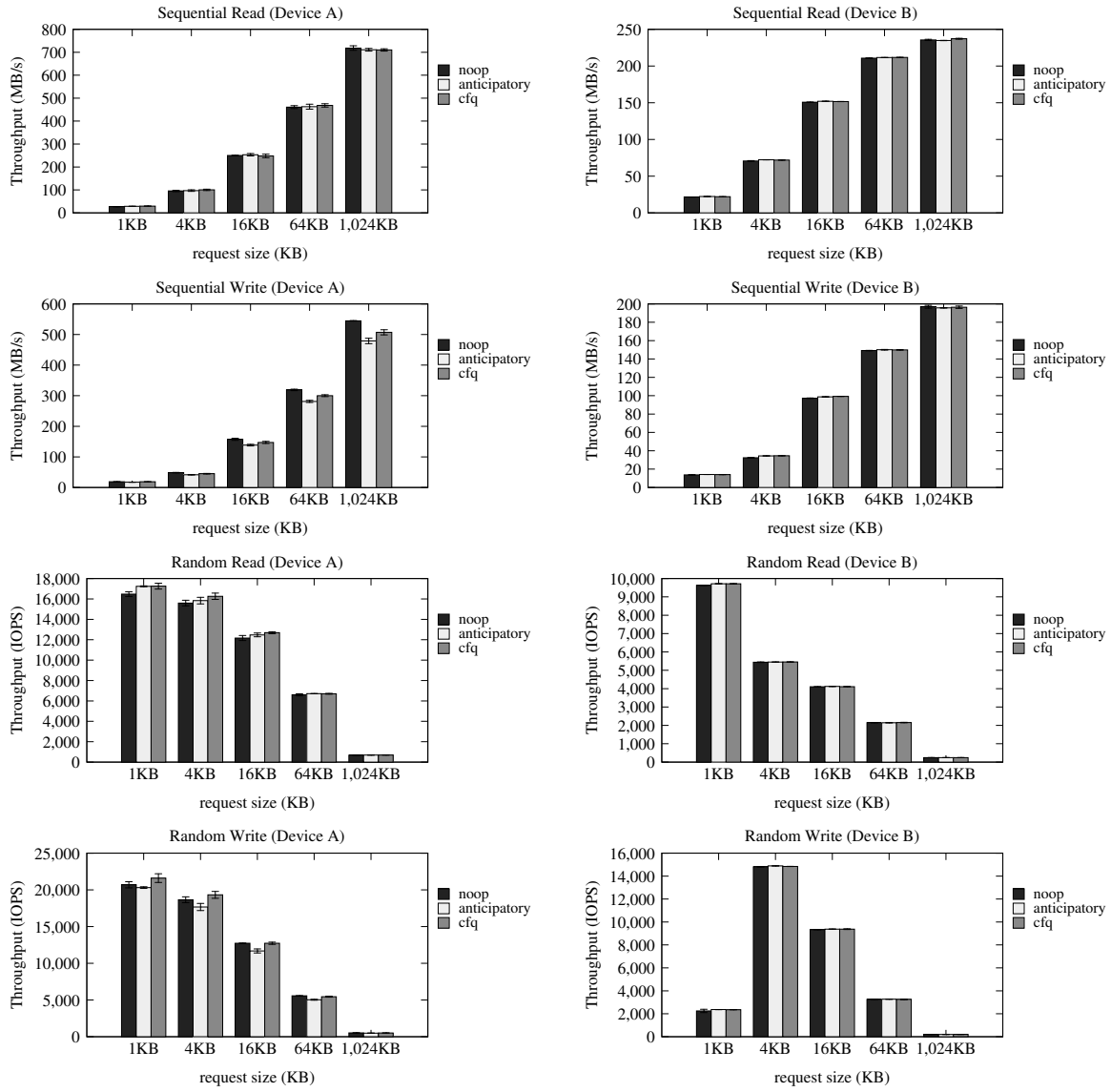
C. I/O Modes

For comparing performance across different I/O modes, the XFS file system and the cfq I/O scheduler are used, because these are the best in general in other tests. The outcomes of the measurements are summarized in the Figure 2.

For sequential I/O, the non-direct mode achieves flat performance across most of the request sizes for both reads and writes. This phenomenon can be attributed to read-prefetching and write-behind for reads and writes respectively, filesystem mechanisms that convert many small accesses into a few larger accesses. On the other hand, in the direct mode the observed bandwidth increases with increasing request size until saturated at the peak performance, meeting or even surpassing—for Device A writes—the bandwidth achieved by the non-direct mode. The advantage and limitation of direct mode is that aplication controls the parallelism provided by wide accesses; it does too little, performance suffers compared to non-direct mode, but it can improve on the non-direct mode if its access sizes exceed non-direct sizes (in terms of useful data for the application) and the device can service the extra size.

For random I/O, the non-direct mode again achieves superior performance to the direct mode. Given that the benchmark issues requests to a 4 GB file and the server contains 4 GB of memory, this phenomenon can be mostly explained by caching; that is, after a startup time, most of the file will be in cache and reads will not go to the device. For writes, the non-direct mode can still benefit from write-behind. Notably, for random writes on Device A the direct mode outperforms

FIGURE 4. Impact on Performance of Varying I/O Schedulers



the indirect mode. This suggests that internally Device A is optimized for modifications smaller than 4 KB, perhaps using smaller page sizes or buffering internally, while Device B may be forced to do a 4 KB read-modify-write when issuing a 1 KB write.

In Section III-A above, we saw the importance of parallelism for random I/O performance. With larger request sizes, however, parallelism may be less important if the device stripes data across multiple banks. In Figure 2, looking at the "direct" mode of accessing file data through a filesystem but not allowing the filesystem to prefetch or write behind data in its cache, random read access of Device A on 16 KB requests achieves about 80% of the throughput in terms of IOPS as 4 KB requests (meaning we're able to randomly read almost 4 times faster with random reads of size 16 KB) despite IOZone running with concurrency of one. In a simplified model of Flash this doesn't make sense. However, it can be explained by the more complicated model of a Flash device that maps its logical address space to physical Flash striped on 4 KB blocks. Each four 4 KB direct request may be issued to only a single bank (achieveing the same IOPS as the random reader of queue depth one on Device A in Figure 1), whereas a 16 KB request can be striped across four banks (achieving the same bandwidth–in terms of bytes per second–as the random reader of queue depth four on Device A in Figure 1).

D. Filesystems

For comparing performance across different file systems, the I/O mode is set to "non-direct" and I/O scheduler to "cfq". We use "non-direct" access because we want to measure the affect of the filesystem's read ahead and write behind decisions. We use the "cfq" scheduler as it is the default I/O scheduler in our Linux kernel. The outcomes of the measurements are summarized in the Figure 3.

For sequential I/O, different file systems exhibit quite a range of behavior. This variability is present across all request sizes for the Device A and small writes for the Device B. In general both ReiserFS and ext3 underperform relative to ext2 and XFS. Also XFS and ext2 tend to achieve the advertised bandwidth, except for sequential writes on the Device A, which seems to have significantly more trouble with sequential writes, especially in 4 KB request sizes.

For random I/O, one can observe less variability in the behavior across different file system. The exception being that ext3 tends to perform poorly on writes on Device A. It is worth noting that the IOPS achieved for 1 KB writes is smaller than that for 4 KB, breaking a trend of smaller accesses achieving higher IOPS. We conjecture that this is caused by the file system managing its buffer cache in 4 KB pages. Thus every 1 KB writes requires a read-modify-write sequence, which hurts performance.

For random reads in Device B, the IOPS generally halves with our increasing sizes. However, the size of our requests, and hence the amount transferred per IOP, quadruples. Despite their the intuition of SSDs as seekless devices, they still manifest higher read bandwidth with larger request sizes. On the other hand, for writes, going from 4 KB to 16 KB and 64 KB does not change the amount of data transfered significantly. For Device B it is actually very close to the peak bandwidth. We conjecture that for random writes, the throughput is dominated by the rate at which 256 KB erase blocks may be cleared rather than the time to write the data.

E. I/O Schedulers

For comparing performance across different I/O schedulers, the direct I/O mode is enabled and the XFS file system is used. XFS was chosen as the filesystem due to its high performance in Section III-D. We ran the experiments in direct mode, however, as we wanted to the scheduler, rather than the filesystem to be the only layer influencing the ordering and timing of requests. The outcomes of the measurements are summarized in the Figure 4. Note that we see performance numbers consistent with the direct mode results of Figure 2. For example, Device A performs relatively well on 1 KB random writes in terms of IOPS compared to Device B where we suspect that 1 KB random writes are slower than 4 KB random writes due to the necessity to perform read-modify-write operations in the device.

The performance observed for different access patterns and request sizes in our measurements varies at most by 5% between different I/O schedulers. This is very different from the performance on magnetic disks as described in [9]. Although one might expect that reordering or delaying the requests before issuing them to the device might matter, our findings speak otherwise. In short, unlike for mechanical disks, the choice of I/O schedulers has little impact on the performance of SSDs.

F. Sustained Random Write Performance

The principle special property of SSD design is the need to erase a large block before writing it. In addition to erase being a relatively slow operation, erase block sizes are usually relatively large (256 kilobytes on a typical NAND device [5]). With sufficient numbers of free pre-erased blocks, the cost of an erase cycle can be hidden in the response time of the write requests. So achieving consistently high write performance relies upon maintaining a pool of free erased blocks, an analogous problem to background cleaning in log structured filesystems [10].

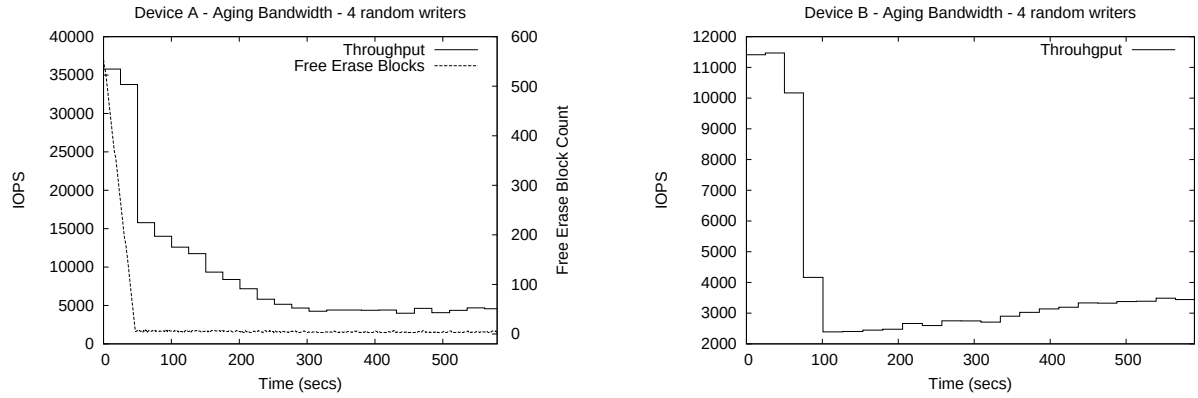
In this experiment, we examine how a sustained random write workload across the devices affects the bandwidth over time by depleting this pool of free erased blocks. We configured the `fio` benchmark to run four threads each writing randomly in 4 KB requests across the entire address space of the device. Figure 5 shows the aggregate IOPS seen by the four threads averaged over 25 second windows across a 10 minute run.

Figure 5 shows the performance is initially high at the start of the run, but quickly degrades at the point at which writes can no longer be issued without synchronously performing an erase. Device A actually lets us access the number of free erase blocks on the device, and we have shown this over the course of the experiment in the left figure. Once the free erase block count hits a certain low watermark, performance begins to level off. Although we have no way to access the count of free erase blocks on Device B, we expect that it is internally experiencing a similar situation.

This behavior could have consequences for any long running, large workload. For example, the queue depth experiment in Section III-A wrote to the entire devices for ten minutes for its random write bars, meaning the IOPS shown in Figure 1 is the aggregate performance including both an initial period of fast access and a longer, slower access period.

In order to maintain high sustained write speeds one must find ways to avoid running out of free erase blocks while servicing writes. For example, writing to only a subsection of the device address space (over provisioning the Flash storage) would allow the cleaner to more easily maintain a pool of free erased blocks. Arguably, this is a common usage scenario and an application which writes randomly to the entire address space of the device is an unsual or even pathological workload. However, we found that even if we repeated the experiments shown in Figure 5 over only 10% of the address space, after having written the entire address space once, there was still an identical falloff. The problem arises because although only a small portion of the address space is being used by the current workload, the contents of the other 90% of the address space is still being maintained even if there is no useful data in these blocks. In this case, the device was still responsible for the integrity all data written to it from previous experiments, including the experiments that issued random writes across its entire address space. If the device were not responsible for maintaining the contents of all blocks, but only

FIGURE 5. Sustained Random Write Performance



"in-use" blocks, it could more aggressively erase blocks in the background or overwrite a block without first copying the contents out of the way.

In order to benefit from writing a smaller portion of the address space, the filesystem could use a SATA command to inform the device that it no longer needs to maintain the data stored in a block range. Such a command, called 'trim', has been proposed [11] . We believe that such operations will be important to achieving high performance in SSDs in the future.

IV. Conclusion

In this study we have examined a number of different influences on the performance of two enterprise Flash-based solid state devices. We have seen how the choice of filesystem can have a surprisingly large effect on I/O performance and that the choice of accessing a device through the filesystem versus directly has consequences for throughput—especially with regards to sequential access. We have seen how high parallelism is important to achieving the best random access on the devices and how certain write patterns can hurt performance by exhausting the supply of free erase blocks.

The design of a storage system including Flash-based solid state devices must take into account questions of how it will write to the SSD (how randomly and to how much of the address space), what level of concurrency the workload can expect to achieve, and through what storage software layers will the devices be accessed.

In order to achieve peak sequential performance of an SSD, a workload needs to read and writes in large chunks—either directly or leveraging read-ahead and write-behind through a file system layer. This is the same advice as for magnetic disks.

To achieve peak random performance proves to be much harder. A workload has to exhibit a high degree of concurrent I/O to keep all independent banks of an SSD busy. At the same time, one must refrain from an access pattern that would result in depleting the pool of pre-erased blocks that would in turn detoriate the performance.

References

[1] E. Pugh, "Storage Hierarchies: Gaps, Cliffs and Trends," *IEEE Trans. Magnetics, v. MAG-7*, pp. 810–814, 1971.

[2] Milo Polte, Jiri Simsa, Garth Gibson, "Comparing Performance of Solid State Devices and Mechanical Disks," *3rd Petascale Data Storage Workshop, Supercomputer 08*.

[3] "Flash Memory vs. Hard Disk Drives - Which Will Win?" [Online]. Available: http://www.storagesearch.com/semico-art1.html

[4] I. Koltsidas and S. D. Viglas, "Flashing up the storage layer," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 514–525, 2008.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD performance," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[6] "Ubuntu Linux Distribution." [Online]. Available: http://www.ubuntu.com/

[7] "IOZone Filesystem Benchmark." [Online]. Available: http://www.iozone.org/

[8] Jens Axboe, "Flexible IO Tester." [Online]. Available: http://git.kernel.dk/?p=fio.git;a=summary

[9] S. L. Pratt and D. A. Heger, "Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers," *Linux Symposium, Ottawa, 2004.*

[10] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, 1992.

[11] "Windows 7 Enhancements for Solid-State Drives."