Towards SSD-Ready Enterprise Platforms

Annie Foong
Intel Corporation
2111 NE 25th Ave
Hillsboro, Oregon 97124
annie.foong@intel.com

Bryan Veal
Intel Corporation
2111 NE 25th Ave
Hillsboro, Oregon 97124
bryan.e.veal@intel.com

Frank Hady
Intel Corporation
2111 NE 25th Ave
Hillsboro, Oregon 97124
frank.hady@intel.com

ABSTRACT

High-performance solid state disks (SSDs) deliver a 2–3 orders of magnitude increase in I/O operations per second (IOPS) over hard disk drives (HDDs). Extreme-performance SSDs can produce up to 120,000 IOPS for random-access reads, taking as few as eight direct-attached SSDs to reach one million IOPS. However, today's multi-core platforms have long been optimized for HDDs, leading to the question: Are platforms ready to deliver such extreme IOPS to the application? To this end, we provide measurements aimed to expose the correct optimizations necessary to deliver such performance. We found that the majority of platform I/O latency still lies in the SSD and not in system software. We identified data copies, uncacheable MMIO reads, interrupt processing, and context switches to be the primary contributors of I/O processing cost. We found typical reference platforms to be surprisingly robust, delivering up to 177,000 IOPS from a single host-bus adapter (HBA). We validated almost 500,000 IOPS with as many HBAs we can put into the platform while maintaining 50% CPU utilization, showcasing the extreme IOPS capable on single volume platform.

1. INTRODUCTION

Solid State Drives (SSDs) achieve latencies 1/100 that of hard disk drives (HDDs) and deliver unprecedented random-access performance. Currently, a single Intel® X25-E SATA* II SSD can deliver 40,000 read IOPS [1], a Crucial* C300 SATA III SSD ca deliver 60,000 IOPS, and the PCIe-based Fusion-io* ioDrive* can deliver 120,000 IOPS [2]. It now takes as few as 8 direct-attached SSDs to deliver one million IOPS to a single platform—something that requires a SAN array of HDDs to deliver.

This new storage technology motivated many ideas about using SSDs to deliver platform performance. Researchers suggest that self-managing SSDs must be approached differently from HDDs, and proposed that a more expressive interface (e.g. object-based storage) be adopted [3]. Others have focused on designing new caching schemes and file systems specifically for SSDs [4][5]. However, most studies failed to provide quantitative evidence of performance improvement. To characterize file systems suitability for SSD, Shin, et.al., empirically compared common Linux file systems (NILFS, btrfs, ext2, ext3, ext4, ReiserFS, and XFS) by running Postmark (an email server) on SSDs [6]. He noted performance differences when key parameters (e.g. block size, allocation policies, barrier enforcement) are changed. Benchmarking led him to conclude that correct alignment is important for SSDs and specific file systems (e.g. xfs) perform significantly better when mounted without barriers.

Since SSDs fill the gap between memory and disk cost and speed, many have proposed to adopt SSDs as a cache, either as OS-managed extension of the buffer cache [5][7] or managing HDDs and SSDs in storage hybrid pools [8]. Roberts, et.al., argued for the extended buffer cache model as the best way to integrate NVM into servers [5]. They showed through trace-driven simulation that splitting NVM-based buffer cache into read and write regions improved power performance and reliability. Still others have proposed reconfigurable flash controllers to adapt to differing workloads [9]. These efforts have been based on the assumption that the platform is already designed to handle the extreme I/O improvement that SSDs deliver.

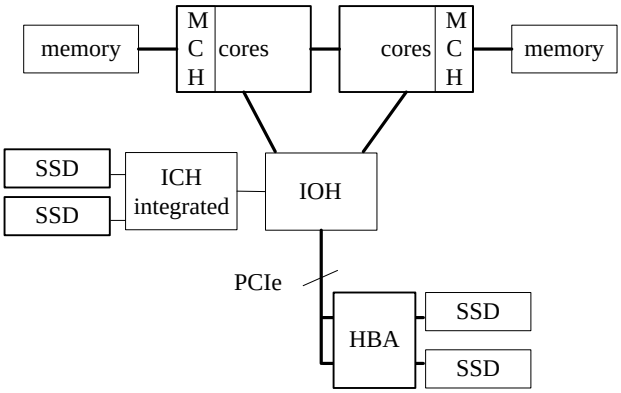

Figure 1. Architecture of a reference multi-core platform.

Does this assumption hold, and if not, how should platforms be redesigned to deliver extreme IOPS cost-effectively? While performance reports of SSDs abound in trade literature [1][10], market-targeted performance measurements and empirical studies alone are inadequate to enable systems architecture advancement. New technologies require exposing basic relationships. Accardi and Wilcox took a first step by implementing an ATA-based and SCSI-based RAM disk in Linux [11]. They found the ATA-based emulation to perform significantly better than the SCSI-based version. From this observation, they deduced that SCSI adds significant overheads. However, due to the adoption of a RAM disk, their analysis did not consider overheads due to HW/SW interaction, nor other OS-support processing (besides SCSI) occuring in the SCSI layer. Further root-causing is required. Agrawal, et.al., provide an excellent assessment of algorithms and basic design considerations of SSDs [12]. Chen, et.al., designed a set of measurement methods targeted specifically to expose differing SSD behavior from different vendors [13]. These researchers provided the basics necessary for assessing SSDs as individual devices. In this paper, we expose the basics for assessing them as part of the larger system. We include in our analysis chipsets, HBAs, cores, software and the interactions among them (Figure 1). Our goals are two-fold:

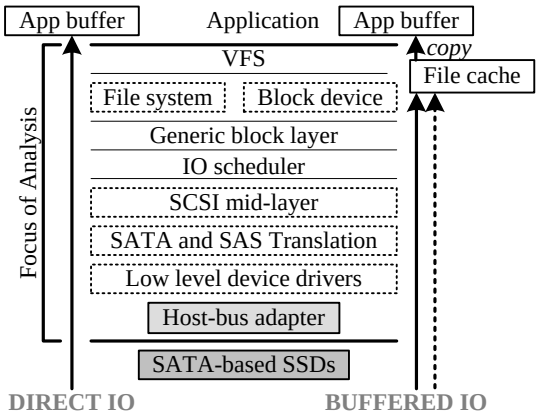

Figure 2. Architecture of the Linux I/O subsystem.

1. provide fundamental measurements and relationships that form the primitives necessary to frame platform storage architecture research, and
2. pinpoint platform bottlenecks and optimization opportunities as a first step toward solutions.

We kept *the existing HW platform and operating system (OS) infrastructure* in place, and determined how far we can push current architecture. We assumed applications to be well-parallelized, and we isolated system software and hardware bottlenecks exposed by SSDs. To this end, we focus on three sub-goals:

1. minimize platform latency,
2. ensure processing efficiency to maximize throughput per core, and
3. scale performance with the numbers of cores and SSDs.

Trade literature claimed that direct PCI Express connectivity can deliver 10x lower latency than SATA-based SSDs by eliminating the need for intermediate protocols [14]. However, we confirmed that platform latency is *not* an issue—the SSD remains the primary contributor. We determined that HBAs with optimized host interfaces enable a processing efficiency of 20K clocks (CPU cycles) per I/O. Ironically, except for the block layer, we found the majority of processing overheads lay in the cost of generic device/OS functions (e.g. the driver, copies, interrupts and context-switching) and not in storage specific functions such as SCSI or ATA processing. However, we did find a scaling limitation imposed by a single HBA as we push IOPS with more SSDs and cores.

In the next section, we present background necessary to understand the reference platform and system software. We then proceed to present the experimental designs and measurements needed to achieve the goals we have outlined.

2. PLATFORM AND ARCHITECTURE

We used Linux* as a reference OS for experiments. The components (Figure 2) can be generalized to any general-purpose OS.

The file system organizes data into files and directories and provides methods to create, update or delete data. The virtual file system (VFS) in Linux provides a common application interface for multiple types of file systems. It also provides raw disk access by treating whole disks and partitions as files (e.g. `/dev/sda`).

Table 1. Make up of different I/O paths.

	Direct I/O	**Buffered**	**Direct I/O**	**Buffered**
	Block	**Block**	**Files**	**Files**
VFS	x	x	x	x
filesystem			x	x
metadata			x	x
cache management		x		x
copies		x		x
I/O stack	x	x	x	x
DMA	x	x	x	x
hardware interface	x	x	x	x

The generic block layer is responsible for locating the actual storage device and logical block address (LBA) on the drive for each I/O request. This typically involves translating a partition and offset into an absolute LBA. The generic block layer may also invoke the device-mapper if the partition belongs to a logical device. For example, it may invoke the Logical Volume Manager (LVM) to translate a request for a logical device into a physical device. Once a real device and LBA are found, the generic block layer invokes the I/O scheduler which attempts to reduce media seek time by reordering requests and coalescing adjacent requests.

Devices, that use common storage protocols (e.g. SAS* and SATA), are abstracted and exposed to the generic block layer as SCSI disks via the SCSI mid-layer. Requests to SATA and SAS adapters are translated from SCSI to their respective protocols, either through an OS-supplied library (e.g. `libata` and `libsas`), or through solutions supplied by HBA vendors.

Four major I/O paths may be composed between the application and device (Table 1) by choosing one of each of the following:

1. I/O through a file system or directly to a block device, and
2. I/O through the buffer cache (buffered) or directly to the device (direct I/O).

Using the VFS, Linux presents the same interface to applications independent of the chosen path. If data is accessed through a file system, extra processing and disk reads are needed to access meta-data to locate the actual file on first access. Such meta-data are stored in the buffer cache to accelerate future accesses to the same data. The meta-data is *always* brought into the buffer cache even if data accesses were done in the direct I/O mode. As such, the method of acquiring meta-data is dependent on the workload and the state of cache. There are usage models that benefit from using or bypassing the buffer cache or controlling its use explicitly.

Whether an application incurs buffered or direct I/O and whether it hits or misses buffer cache when doing meta-data accesses are workload and policy dependent. Workload and file system characterization of real applications is required to solicit the most frequently occurring (and hence important) complex paths. In this paper, we offer an initial characterization of the complex paths, but focus our in-depth analysis on direct I/O. Direct I/O without a file system is the most primitive path upon which the other paths are built, and its performance is relevant to any application.

Table 2. Test system setup.

Platform	Dual 2.93GHz Intel Xeon® X5570 Processors
Adapters	Integrated AHCI SATA adapter 8-port LSI* Fusion MPT* SAS adapter
Application	FIO synchronous I/O with up to 128 concurrent threads
Operating system	Linux 2.6.28
File systems	direct block access, ext3
I/O scheduler	No-op
Drives	Intel X25E SSDs Hitachi* Deskstar* HDDs
Other configuration	SMT, CPU low-power states, frequency scaling disabled; Linux tickless timer disabled; 10ms scheduler time slice; polling when idle

3. METHODOLOGY AND SETUP

Having identified independently separable components of an I/O path (Table 1), we build upon microbenchmarks (FIO [15]) that exercise these components exactly. We have also identified a minimum set of platform parameters that is important for duplicating our measurements in (Table 2).

We measured path timing by reading the processor's timestamp counter (`rdtsc()` [16]), and we gathered statistics reported by the OS (e.g. `mpstat`, `iostat` [17]). We adopted OProfile [18], a statistical event sampling tool, as an alternate means of validating clocks and instruction count. To maintain a small number of representative experiments, we focused on fixed-size 4KB random reads. Random I/O ensures that improvements are observed independent of I/O merging policies. We used only reads since writes take a nearly identical path through the platform (with data transfer in the opposite direction). Furthermore, SSDs deliver much higher performance for read versus writes. A read-intensive workload pushes the platform further—if the platform can deliver full performance for reads, it certainly will deliver full performance for writes.

4. PLATFORM BOTTLENECK ANALYSIS

We classified platform I/O bottlenecks into three types and we examined them separately.[1]

1. **Platform latency bottlenecks**—We isolated the component which dominates I/O latency, thus determining which component to fix to reduce overall latency.
2. **I/O processing bottlenecks**—We determined which software contributed the most CPU overhead for I/O processing. I/O processing overhead reduces the maximum bandwidth per CPU and limits the CPU clocks available to the application.

[1] Performance results are based on certain tests measured on specific computer systems. Any difference in system hardware, software or configuration will affect actual performance. **For more information, go to http://www.intel.com/performance.**

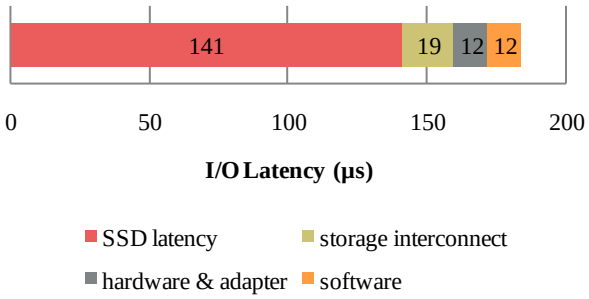

Figure 3. Latency of a 4KB I/O to a SATA II SSD through the platform as experienced by an application.

3. **Performance scaling bottlenecks**—We measured performance as the number of CPU cores available for processing increased, and we determined which platform components limited scaling of performance.

4.1 Platform Latency

We define *Total I/O latency* seen by an application thread as the time from when the application issues an I/O to the time it receives its completion (last byte):

$$\text{total I/O latency} = \text{time due to media} + \text{time due to platform}.$$

Time due to platform includes latency through software (I/O issue and completion processing), time through queues, platform hardware (e.g. chipsets), adapters, and the time it takes to transfer a given size of the payload data through a given link speed.

Because there is no "seek time" to an SSD, the I/O scheduling policy has minimal impact on performance for SSDs. We set the I/O scheduling policy to *no-op* and ensured (and validated) nearly zero wait time in queues. From our processing cost measurements (Section 4.2), we derived total latency through software. Measurements taken at the SATA bus (using a SATA bus analyzer) exposed media latency and time taken to transfer payload bytes across the 3Gbps SATA II link. Combining both sets of measurements, we derived latency contributions of the entire I/O path through the platform.

Despite huge improvements in media access times, the SSD is still the major contributor of latency (Figure 3). *The platform only contributes 26% of the total latency.* Optimizing the media is necessary to make meaningful latency improvements.

4.2 I/O Processing Cost

I/O processing on the platform must be highly efficient in terms of CPU clocks to reach the high IOPS made possible by SSDs. We consider *clocks (CPU cycles) per I/O* as the metric of processing efficiency. We used instead of nanoseconds to reflect the efficiency of a platform with a given memory architecture. In this section, we restrict analysis to a single core, differentiating it from scaling analysis that is addressed later (Section 4.3).

We connected the SSDs to the platform-integrated AHCI-based SATA controller. Figure 4 shows the processing time of an I/O from issue to completion. An I/O issue request begins with a system call by the user application and transitions into kernel space. User-kernel-user transitions (system call and return) took 924 clocks. The original file-based I/O is framed into blocks, then

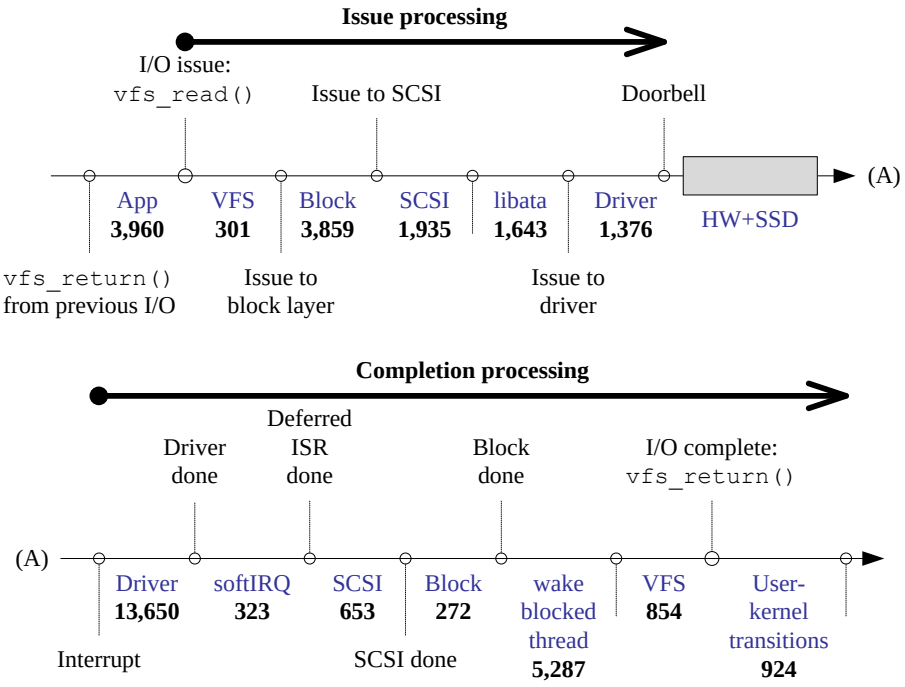

Figure 4. Processing requirements from I/O issue to completion in clocks (not to scale).

into SCSI commands and prepared by the driver for issue to the HBA. Once the I/O request is handed off to the adapter (through the issue of a doorbell), CPU clocks are no longer expended. When the disk completes the request, an interrupt to the CPU enables entry into driver's interrupt service routine (ISR) and completes the rest of the I/O; the I/O is finally handed back to the application.

Processing an I/O required about 35,000 clocks with disks connected via AHCI. The largest hotspot was in the return path of the driver. The most expensive functions were in `ahci_interrupt()` and `ahci_scr_read()`. These functions executed uncacheable (UC) reads from memory-mapped device registers on the adapter. The UC reads incurred significant processing cost, averaging 2,100 clocks per UC read. Device

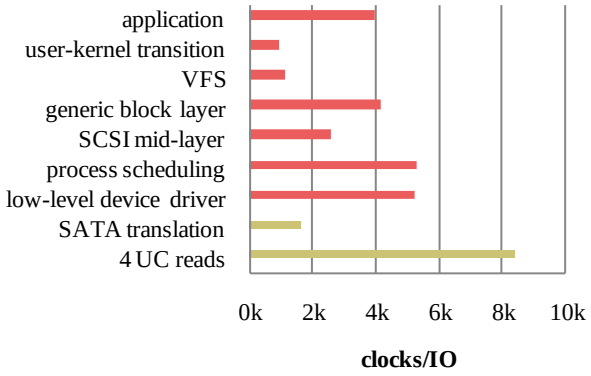

Figure 5. I/O processing distribution with LSI (4KB reads, single-core). SATA and UC read costs are shown for comparison.

interfaces that adopt message signaled interrupts (MSI), and the added intelligence to push status to drivers, can eliminate such UC reads. We predict that this optimization alone will reduce overhead by about 8,400 clocks/IO.

We were able to validate this insight using an enterprise-targeted adapter (the LSI Fusion MPT SAS adapter). Figure 5 is based on the same instrumentation done in Figure 4 but is presented in a different way for ease of comparison. The profile combines both issue and completion processing on a per-layer basis. We found no UC reads in LSI's driver (`mpt2sas`). Additionally, no SATA-related routines were used. The LSA adapter offloaded the SAS/SATA conversion saving 1,600 clocks on the host. While the cost-targeted AHCI SATA controller had served the industry well for HDDs, SSDs expose the need for higher performance adapters. Given LSI adapter's more optimal interface, further measurements are done with LSI adapters. I/O processing, when done through an MSI-based interface like LSI's, incurred 25,000 clocks/IO, yielding a 30% improvement in processing efficiency.

The LSI's driver return path (5250 clocks/IO) is still substantial. We expected this to reduce due to batch processing of I/O interrupts by employing interrupt coalescing. Our validation of this optimization potential is shown in Figure 6. At large numbers of coalesced interrupts, all but 650 clocks remain in the driver return path, resulting in about 20,000 clocks/IO for throughput-intensive workloads that can leverage interrupt coalescing.

Beyond these optimizations, the largest system overheads for the direct I/O path occurred in OS context switching and the block layer. We found that the clocks/instruction (CPI) of the block layer to be small (1.3), and it would be difficult to improve CPI further. Improving and/or finding alternatives to reduce context switches may yield better results.

To put further potential optimization into perspective, we offer a quick assessment of the overheads for other paths that exist in the

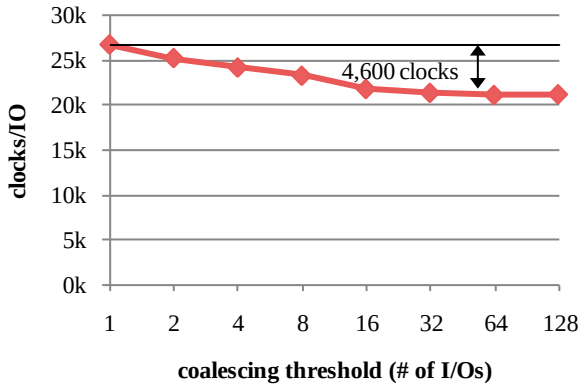

Figure 6. I/O processing cost with interrupt coalescing.

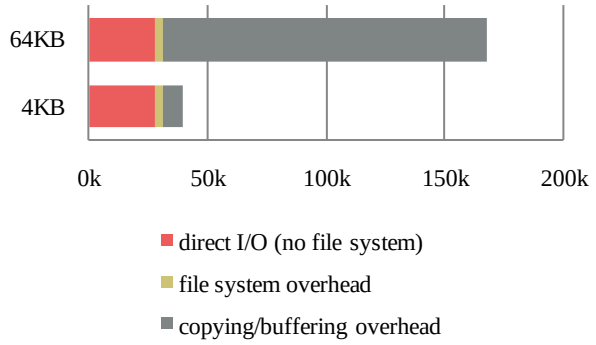

Figure 7. Processing overheads of file system and buffered I/O.

I/O stack (Table 1)—including a file system (ext3) and buffered I/O. As noted earlier, analysis of these complex paths are a topic of future work and will require more careful treatment than given here. Nevertheless, it is still worthwhile to get an indicator of the magnitude of these overheads.

When the application uses a file system, meta-data is required to determine the location of the intended block. The number of meta-data accesses is dependent on the file system in question. Moreover, the meta-data may or may not be in cache depending on the application workload. We made a simplifying assumption that platforms have a well-resourced buffer cache, and as such, a majority of file accesses do not require meta-data access from the device (except for the very first access to a location in the file). Our measurements are taken by first ensuring that the file system's meta-data is already in cache. The clocks/IO attributed to the file system is the processing required to read data (not including meta-data) from the disk. We noted that the file system (ext3) adds only 3,400 clocks (<15%) to the path (Figure 7).

In the case of buffered I/O, the buffer cache is now the DMA destination, requiring an extra copy of the payload to its final destination in the application (Figure 2). We designed our experiments to fully expose the entire buffered I/O path (and not just a copy from cache). The difference in clocks is expected to increase with payload sizes. The extra cost of copies and associated cache management was validated to be a near-linear function of request size (2.1 clocks/byte) (Figure 7). We note that

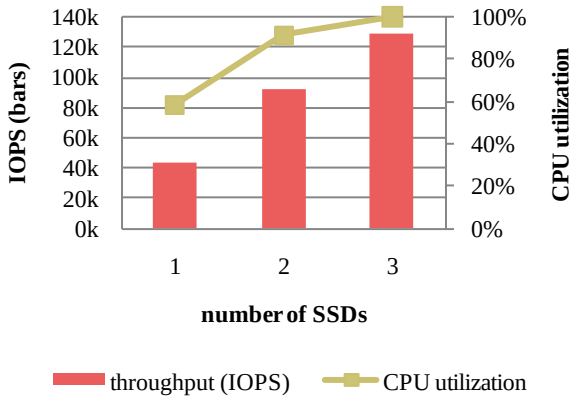

Figure 8. IOPS and CPU utilization for 3 SSDs taken on a single core.

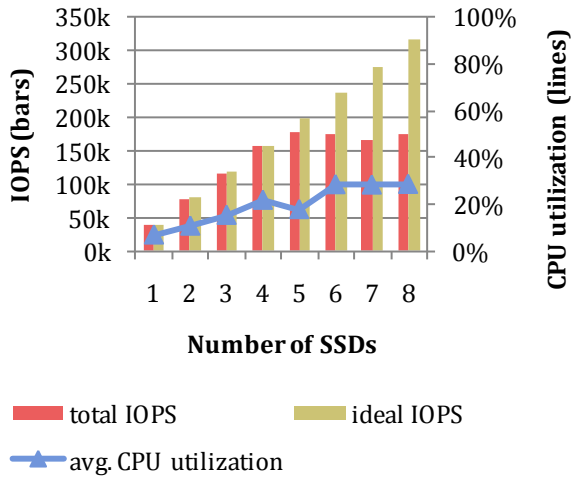

Figure 9. IOPS scaling and CPU utilization for 8 SSDs on 8 cores.

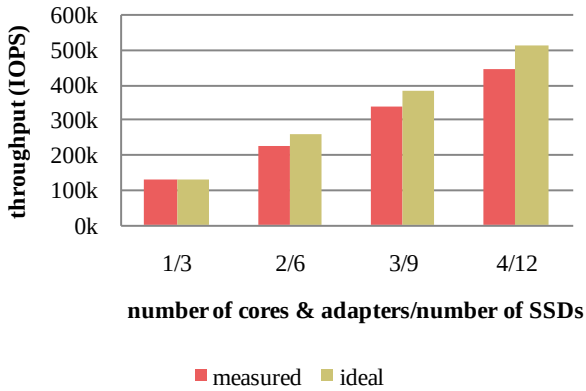

Figure 10: Scaling of IOPS as the number of adapters increases with the number of cores.

this overhead can overwhelm I/O stack requirements at large transfer sizes, and may be a worthy candidate of attention.

4.3 Performance Scaling

With the ubiquity of multi-core platforms, we turn our attention to ensuring that I/O processing scales with cores and SSDs. Having ensured processing efficiency, we now focus on achieving maximum throughput (IOPS and MB/s) platform-wide.

We show in Figure 8 that the maximum throughput on a single core for 4KB random reads reached 129,000 IOPS. At this point, a single core was fully saturated—more cores were required.

Figure 9 shows that with all eight cores available for I/O processing, throughput stalled at 177,000 IOPS with one adapter. Thus, we added more adapters. As shown in Figure 10, adapters were increased one at a time from one to four while also increasing the number of available cores from one to four. (Due to the limited number PCI Express* slots on our reference platform, we were only able to showcase up to four adapters.) We tested three SSDs on each adapter for a maximum of twelve. The figure shows that throughput scaled within 15% of linear—up to 445K IOPS. Since more adapters enabled higher total throughput, *the single adapter was the bottleneck.* At this performance, I/O processing saturated four cores—half the available cores on the platform.

We also measured data throughput across the four adapters and 12 SSDs with 64KB I/O sizes. The result was 3GB/s, within 2% of linear (Figure 11).

5. CONCLUSIONS

We have exposed all platform bottlenecks that may interfere with delivering full SSD performance to applications, and have pinpointed further areas of investigation. The following measurements make up key primitives that drive our future design decisions:

1. I/O latency (for 4KB reads): 180µs. The platform hardware and software made up 20µs of the latency while the the SSD accounted for the rest.
2. I/O processing cost (using performance-targeted adapters): 25K clocks/IO (20K with interrupt coalescing).
3. Performance scaling: Limited to 177,000 IOPS with a single adapter. We reached 445K IOPS (15% from linear) and 3GB/s (2% from linear) with four adapters at 50% average CPU utilization.

We set out to determine if HDD-targeted platforms could deliver the full performance from SSDs. From our investigation, we concluded that the SSD, not the platform, is still the primary contributor to I/O latency. We also concluded and validated that removing UC reads and employing interrupt coalescing will reduce I/O processing cost significantly for SSDs behind an AHCI host interface, and we found an enterprise-targeted adapter to be CPU-efficient. We found a performance scaling bottleneck incurred by using a single adapter which delivered 177K IOPS (matching the IOPS of 4 SSDs). We had to increase the number of adapters to achieve more—a total of four adapters were required to deliver 445K IOPS (matching the IOPS possible from 12 SSDs) For enterprise workloads that would not require extreme IOPS from a single adapter, we found that existing platforms to be ready for SSDs.

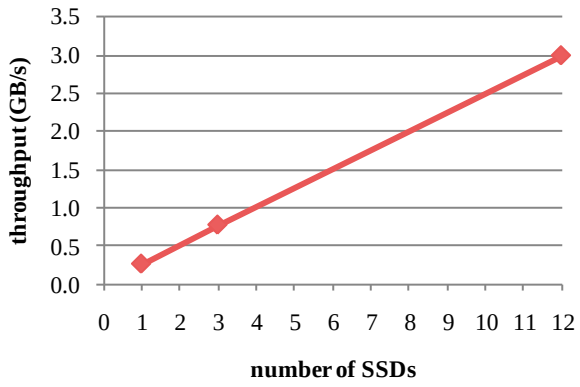

Figure 11: Scaling of MB/s as the number of adapters increases with the number of cores.

6. FUTURE WORK

Beyond these issues, we recommend the following directions as worthy of pursuit:

1. Determine the scalability of file systems and RAID implementations. We have exposed scaling issues when SSDs are used as JBOD (just-a-bunch-of-disks), but real enterprise applications run with file systems and RAID arrays. Additionally, file systems may gain from organizing meta-data in a way that take into account the special characteristics of SSDs.
2. Determine whether the overheads of context-switching can be mitigated with asynchronous I/O (AIO). We did not observe processing cost improvements with AIO, and believe (but have not confirmed) that Linux's use of a kernel thread to emulate AIO to be insufficient. Such an implementation moved (but did not eliminate) the context switch. We call for developers to optimize AIO in Linux.
3. Expose I/O behavior of real applications. We have focused on data transfer as the performance-critical path, but cannot assume such behavior to be universally true of all applications. Characterization of more applications may expose bottlenecks in other important paths (e.g. retrieval of meta-data).

Removing platform bottlenecks is a necessary first step—further optimization based around SSDs unique characteristics can now follow. We have established an accurate set of fundamental measurements and performance expectations, which we hope provide a reference that other researchers can build upon, ultimately leading to innovations that drive new platform architectures around this exciting technology.

7. REFERENCES

[1] Coles, O. *Marvell SATA-6G SSD performance vs. Intel ICH10*. Benchmark Reviews.com. 2009. http://benchmarkreviews.com/index.php?option=com_content&task=view&id=413&Itemid=38.

[2] FUSION-IO. *MySpace uses fusion powered I/O to drive greener and better data centers*. 2009.

[3] Rajimwale, A., Prabhakaran, V., and Davis, J.D. Block management in solid-state devices. In *USENIX Annual*

Technical Conference Proceedings (San Diego 2009), USENIX.

[4] Kim, H. and Ahn, S. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST Proceedings* (San Jose 2008), USENIX.

[5] Roberts, D., Kgil, T., and Mudge, T. Integrating NAND flash devices on to servers. *Communications of the ACM*, 52, 4 (April 2009), 98–106.

[6] Shin, D. About SSDs. In *Linux Storage and File System Workshop Proceedings* (San Jose 2008), USENIX.

[7] Graefe, G. The five-minute rule 20 years later. *Communications of the ACM*, 52, 7 (2009), 48–59.

[8] Leventhal, A. Flash storage memory. *Communications of the ACM*, 51, 7 (July 2008).

[9] Shin, J.Y., Xia, Z.L., Xu, N.Y., Gao, R. Cai, X.F., Maeng, S., and Hsu, F.H. FTL design exploration in reconfigurable high-performance SSD for server applications. In *ICS Proceedings* (Grenoble, France 2009), ACM.

[10] Connolly, C. *Windows 7 HD and SSD performance analyzed*. 2009. http://hothardware.com/Articles/Windows-7-Disk-Performance-Analyzed/.

[11] Accardi, K.C. and Wilcox, M. Linux storage stack performance. In *Linux Storage and Filesystem Workshop* (San Jose 2008), USENIX.

[12] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M., and Panigrahy, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference Proceedings* (Boston 2008), USENIX.

[13] Chen, F., Koufaty, D.A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS Proceedings* (Seattle 2009), ACM.

[14] DOLPHIN. *Dolphin expands PCI Express SSD storage product line*. SEO Press Releases, Marlborough, 2009. http://www.seopressreleases.com/dolphin-expands-pci-express-solid-state-disk-ssd-storage-product-line/2563.

[15] Axboe, J. *Flexible IO Tester (fio)*. 2010. http://git.kernel.dk/?p=fio.git;a=summary.

[16] INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2009.

[17] Godart, S. *SYSSTAT*. 2010. http://sebastien.godard.pagesperso-orange.fr/.

[18] Levon, J. *OProfile*. 2009. http://oprofile.sourceforge.net/news/.

[*] Other names and brands may be claimed as the property of others.