High Performance Solid State Storage Under Linux

Eric Seppanen, Matthew T. O'Keefe, David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN 55455
{sepp0019, mokeefe, lilja}@umn.edu

***Abstract*—Solid state drives (SSDs) allow single-drive performance that is far greater than disks can produce. Their low latency and potential for parallel operations mean that they are able to read and write data at speeds that strain operating system I/O interfaces. Additionally, their performance characteristics expose gaps in existing benchmarking methodologies.**

We discuss the impact on Linux system design of a prototype PCI Express SSD that operates at least an order of magnitude faster than most drives available today. We develop benchmarking strategies and focus on several areas where current Linux systems need improvement, and suggest methods of taking full advantage of such high-performance solid state storage.

We demonstrate that an SSD can perform with high throughput, high operation rates, and low latency under the most difficult conditions. This suggests that high-performance SSDs can dramatically improve parallel I/O performance for future high performance computing (HPC) systems.

I. Introduction

Solid state storage devices have become widely available in recent years, and can replace disk drives in many applications. While their performance continues to rise quickly, prices of the NAND flash devices used to build them continue to fall. Flash-based SSDs have been proposed for use in computing environments from high-performance server systems to lightweight laptops.

High-performance SSDs can perform hundreds of thousands of I/O operations per second. To achieve this performance, drives make use of parallelism and complex flash management techniques to overcome flash device limitations. These characteristics cause SSD performance to depart significantly from that of disk drives under some workloads. This leads to opportunities and pitfalls both in performance and in benchmarking.

In this paper we discuss the ways in which high-performance SSDs are different from consumer SSDs and from disk drives, and we set out guidelines for measuring their performance based on worst-case workloads.

We use these measurements to evaluate improvements to Linux I/O driver architecture for a prototype high-performance SSD. We demonstrate potential performance improvements in I/O stack architecture and device interrupt handling, and discuss the impact on other areas of Linux system design. As a result of these improvements we are able to reach a significant milestone for single drive performance: over one million IOPS and read throughput of 1.4GBps.

II. Background

A. Solid State Drives

In many ways, SSD performance is far superior to that of disk drives. With random read performance nearly identical to sequential read performance, seek-heavy workloads can perform orders of magnitude faster with an SSD compared to a disk drive. SSDs have other benefits besides performance: they consume less power and are more resistant to physical abuse than disk drives.

But SSDs bear the burden of NAND flash management. Large flash block erase sizes cause write amplification and delays, and drives must have large page-mapping tables to support small writes efficiently. Limited block erase cycles require write-leveling algorithms that must trade off performance for reliability. [1] is a comprehensive review of SSD architecture.

NAND flash devices have large (at least 128KB) block sizes, and blocks must be erased as a unit before fresh data can be written. SSDs usually support standard 512-byte sectors in order to be compatible with disk drives. This gap must be filled by management functions built into the drive or its host driver, which need to manage the layout of data on the drive and ensure that live data is moved to new locations. The relocation of live data and use of a block erase to reclaim partially used flash blocks is known as garbage collection.

NAND flash bits also have a limited lifespan, sometimes as short as 10,000 write/erase cycles. Because a solid-state drive must present the apparent ability to write any part of the drive indefinitely, the flash manager must also implement a wear-leveling scheme that tracks the number of times each block has been erased, shifting new writes to less-used areas. This frequently displaces otherwise stable data, which forces additional writes. This phenomenon, where writes by the host cause the flash manager to issue additional writes, is known as write amplification [2].

Because SSDs can dynamically relocate logical blocks anywhere on the media, they must maintain large tables to remap logical blocks to physical locations. The size of these "remap units" need not match either the advertised block size (usually 512 bytes for compatibility) or the flash page or block size. The remap unit size is chosen to balance table size against the performance degradation involved in read-modify-write cycles needed to service writes smaller than the remap unit.

Solid state drives make use of overprovisioning, setting aside some of the flash storage space to provide working room for flash management functions and reduce write amplification and write delays. It also allows the drive to absorb bursts of higher-than-normal write activity. The amount of overprovisioning need not be visible to the host machine, but may be configurable at installation time.

A very thorough examination of three modern SATA SSDs can be found in [3].

B. Performance Barriers

The performance of disk drives and solid state drives can be roughly outlined using four criteria: latency, bandwidth, parallelism, and predictability.

Latency is the time it takes for a drive to respond to a request for data, and has always been the weak point of disks due to rotational delay and head seek time. While disk specifications report average latency in the three to six millisecond range, SSDs can deliver data in less than a hundred microseconds, roughly 50 times faster.

Interface bandwidth depends on the architecture of the drive; most SSDs use the SATA interface with a 3.0Gbps serial link having a maximum bandwidth of 300 megabytes per second. The PCI Express bus is built up from a number of individual serial links, such that a PCIe 1.0 x8 device has maximum bandwidth of 2 gigabytes per second.

Individual disk drives have no inherent parallelism; access latency is always serialized. SSDs, however, may support multiple banks of independent flash devices, allowing many parallel accesses to take place.

These first three boundaries are easy to understand and characterize, but the fourth, predictability, is harder to pin down. Disk drives have some short-term unpredictability due to millisecond-scale positioning of physical elements, though software may have some knowledge or built-in estimates of disk geometry and seek characteristics. SSDs are unpredictable in several new ways, because there are background processes performing flash management processes such as wear-leveling and garbage collection [1], and these can cause very large performance drops after many seconds, or even minutes or hours of deceptively steady performance. Figure 1 shows the drop-off in write performance of a SATA SSD after a long burst of random 4KB write traffic.

C. Command Queuing and Parallelism

Modern disk drives usually support some method of queuing multiple commands, such as Tagged Command Queuing (TCQ) or Native Command Queuing (NCQ); if re-ordering these commands is allowed, the drive may choose an ordering that allows higher performance by minimizing head movement or taking rotational positioning into account [4]. However, disk drives cannot perform multiple data retrieval operations in parallel; command queuing only permits the drive to choose an optimal ordering.

Solid state drives are less restricted. If multiple commands are in an SSD's queue, the device may be able to perform them simultaneously. This would imply the existence of some

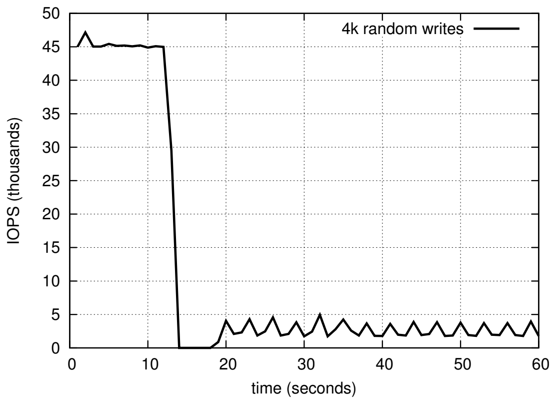

Fig. 1. Write performance drop-off

parallel hardware, such as independent flash dies or planes within a die, or parallel buses to different banks of devices.

This introduces not just a difference in performance between SSDs and disk drives, but a functional difference, that of a parallel storage system rather than a serialized one.

If we presume a device that can complete a small (4KB or less) read in 100 microseconds, we can easily calculate a maximum throughput of 10,000 IOPS (I/O operations per second). While this would be impressive by itself, a device with 2-way parallelism could achieve 20,000 IOPS; 4-way 40,000 IOPS, etc. Such latency is quite realistic for SSDs built using current technology; flash memory devices commonly specify read times lower that 100 microseconds.

Older software interfaces may support only one outstanding command at a time. Current standardized storage interfaces such as SATA/AHCI allow only 32 commands. Non-standard SSDs may support a 128 or larger command queue. There is no limit to how much parallelism may be supported by future devices, other than device size and cost.

D. Impact on HPC

High Performance Computing (HPC) systems can benefit greatly from SSDs capable of high-throughput, low-latency operations. Raising performance levels in storage subsystems will be critical to solving important problems in storage systems for HPC systems, including periodic burst write I/O for large, time-dependent calculations (to record the state of the fields at regular points in time) and for regular checkpoint restart dumps (to restart the calculation after a system failure). These I/O patterns are key problem areas for current teraflop and petaflop systems, and will become even more difficult to solve for exascale systems using traditional, disk-based architectures [5], [6].

HPC storage requirements favor SSDs over disk drives in many ways. High throughput allows a reduction in the number of storage devices and device interface hardware. Support for parallel operations also may provide a reduction in device count. SSDs have low, predictable latencies, which can have significant impact on parallel applications that would be harmed by a mismatch in latency between I/O operations sent to different devices. Additionally, SSDs consume far less power and produce less heat than disk drives, which can allow more compact storage arrays that use less energy and cooling.

Large HPC system designs based on SSD storage are starting to appear [7]. However, to fully exploit SSDs in high performance computing storage systems, especially petascale and exascale systems with tens of thousands of processors, improvements in I/O subsystem software will be needed. High demands will be placed on the kernel I/O stack, file systems, and data migration software. We focus on the Linux kernel I/O processing improvements and SSD hardware required to achieve the performance necessary for large-scale HPC storage systems.

E. Aggregate and Individual Performance

It is possible to take a number of drives with low throughput or parallelism and form an I/O system with large amounts of aggregate throughput and parallelism; this has been done for decades with disk drives. But aggregate performance is not always a substitute for individual performance.

When throughput per disk is limited, many thousands of disks will be required to meet the total throughput required by large-scale parallel computers: protecting and managing this many disks is a huge challenge.

Disk array parity protection schemes require large, aligned block write requests to achieve good performance [8], yet it may be difficult or impossible to modify application I/O to meet these large block size and alignment constraints. In addition, disk array recovery times are increasing due to the very large capacities of current disk drives.

Individual SSDs offer potential orders of magnitude improvement in throughput. Along with greater parallelism, this means that many fewer SSDs must be used (and managed) to achieve the same performance as disk drives.

F. The Linux I/O Stack

The Linux kernel I/O software stack is shown in figure 2. Applications generally access storage through standard system calls requesting access to the filesystem. The kernel forwards these requests through the virtual filesystem (VFS) layer, which interfaces filesystem-agnostic calls to filesystem-specific code. Filesystem code is aware of the exact (logical) layout of data and metadata on the storage medium; it performs reads and writes to the block layer on behalf of applications.

The block layer provides an abstract interface which conceals the differences between different mass storage devices. Block requests typically travel through a SCSI layer, which is emulated if an ATA device driver is present, and after passing through a request queue finally arrive at a device driver which understands how to move data to and from the hardware. More detail on this design can be found in [9].

The traditional Linux storage architecture is designed to interface to disk drives. One way in which this is manifested is in request queue scheduler algorithms, also known as elevators. There are four standard scheduler algorithms available in the Linux kernel, and three use different techniques to optimize request ordering so that seeks are minimized. SSDs, not having seek time penalties, do not benefit from this function. There does exist a non-reordering scheduler called the "noop" scheduler, but it must be specifically turned on by a system administrator; there is no method for a device driver to request use of the noop scheduler.

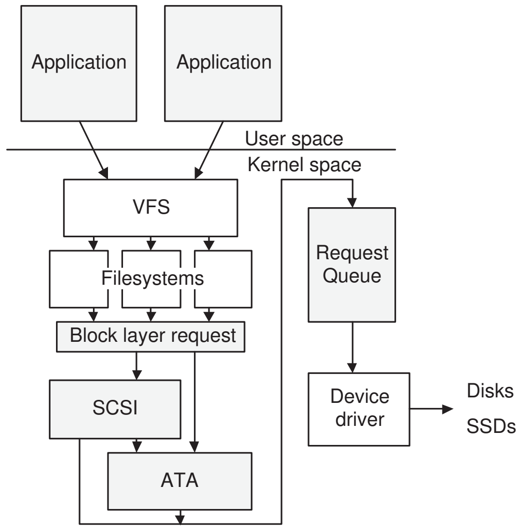

Fig. 2. Linux I/O stack

When new requests enter the request queue, the request queue scheduler attempts to merge them with other requests already in the queue. Merged requests can share the overhead of drive latency (which for a disk may be high in the case of a seek), at the cost of the CPU time needed to search the queue for mergeable requests. This optimization assumes that seek penalties and/or lack of parallelism in the drive make the extra CPU time worthwhile.

The request queue design also has a disk-friendly feature called queue plugging. When the queue becomes empty, it goes into a "plugged" state where new requests are allowed in but none are allowed to be serviced by the device until a timer has expired or a number of additional commands have arrived. This is a strategy to improve the performance of disk drives by delaying commands until they are able to be intelligently scheduled among the requests that are likely to follow.

Some of these policies are becoming more flexible with new kernel releases. For example, queue plugging may be disabled in newer kernels. However, these improvements have not yet filtered down to the kernels shipped by vendors for use in production "enterprise" systems.

G. Linux Parallel I/O

The Linux kernel as a standalone system has limited support for parallel I/O. Generally, to achieve I/O parallelism an application must already be parallelized to run on multiple local CPUs, or use Asynchronous I/O system functions to submit several requests at once.

An application designed for a single-disk system typically makes serialized requests for data. This results in several calls to the *read* system call. Unfortunately, this I/O style (frequently referred to as "blocking" reads) allows no parallelism. The operating system cannot know of a request before it happens, and applications using blocking reads perform a second *read* only after the preceding *read*'s (possibly significant) latency.

An operating system might be able to work around this problem with pre-fetching, but this misses the point: sometimes applications might know in advance that they want to perform several reads from known locations, but serialize the operations anyway.

Applications that are able to be designed with I/O parallelism for single threads or processes may use Asynchronous I/O (AIO). An application designed using AIO may batch together several read or write requests and submit them at once. AIO cannot benefit applications that were not designed for it and cannot be modified.

Linux systems support AIO in two ways. Posix AIO is emulated in userspace using threads to parallelize operations. The task-scheduling overhead of the additional threads makes this a less attractive option. Linux native AIO, known by the interfacing library "libaio," has much lower overhead in theory because it truly allows multiple outstanding I/O requests for a single thread or process [10].

Parallelism is easier to achieve for I/O writes, because it is already accepted that the operating system will buffer writes, reporting success immediately to the application. Any further writes can be submitted almost instantly afterwards, because the latency of a system call and a buffer copy is very low. The impending writes, now buffered, can be handled in parallel. Therefore, the serialization of writes by an application has very little effect, and at the device driver and device level the writes can be handled in parallel.

Real-world single-computer applications exhibiting parallelism are easy to come by. Web servers, Relational Database Management Servers (RDBMS), and file servers all frequently support many clients in parallel; all of these applications can take advantage of a storage system that offers parallel command processing.

HPC systems support more sophisticated methods of performing parallel I/O [11], but those interfaces are not supported at the Linux system level. Our contribution to these environments is enabling high-performance Linux systems that can function as powerful storage nodes, allowing parallel I/O software to aggregate these nodes into a parallel I/O system capable of meeting stringent HPC requirements.

H. Kernel Buffering/Caching

The Linux kernel has a standard buffer cache layer that is shared by all filesystems, and is optionally available when interfacing to raw devices.

There is only limited application-level control over the Linux buffer cache. Most commonly, Linux systems access data only through standard filesystem modules that make full use of the buffer cache.

Applications can request non-buffered I/O using the `O_DIRECT` option when opening files, but this option is not always supported by filesystems. Some filesystems fail to support `O_DIRECT` and some support it in a way that is prone to unpredictable performance.

There is another way for applications to communicate to the kernel about the desired caching/buffering behavior of their file I/O. The *posix_fadvise* system call allows applications to deliver hints that they will be, for example, performing sequential I/O (which might suggest to the kernel or filesystem code that read-ahead would be beneficial) or random I/O. There is even a hint suggesting that the application will never re-use any data. Though this would provide an alternative to `O_DIRECT`, the hint is ignored by the kernel.

The closest one can come to non-cached file I/O under Linux is to request, via *posix_fadvise*, that the kernel immediately discard buffers associated with a particular file. Though it's possible to use this function to emulate non-cached I/O (by requesting buffer discard several times per second), this is an inelegant solution.

I. Raw I/O

Linux allows applications to directly access mass storage devices without using a filesystem to manage data; we refer to this as "raw" I/O. This is useful for measurement of the performance of mass storage devices and drivers, as well as offering a very low-latency API for applications that have their own data chunk management scheme; database management software or networked cluster-computing I/O nodes are examples of applications that can make use of this feature. Typically, raw device access is used with the `O_DIRECT` option, which bypasses the kernel's buffer cache. This allows applications that provide their own cache to achieve optimal performance. Raw device mode is the only time that uncached I/O is reliably available under Linux.

J. Terminology

When discussing storage devices that can handle commands in parallel, it is easy to cause confusion when using the term "sequential," which may mean either commands which read or write to adjacent storage regions or commands which are issued one after another with no overlap.

We use "sequential" to mean the former, and "serialized" to refer to operations that cannot or do not take place in parallel. This does not imply any conflicts or dependencies between the commands, such as a read following a write of overlapping regions. Such conflicting commands are not considered here at all, as they are a rare special case.

Also, we use the term "disk" to refer to a conventional disk drive with rotating media. "Drive" is used to describe a single storage device, either a disk or an SSD.

III. Proposed Solutions

A. SSD-Aware Benchmarking

1) Pessimistic Benchmarking: SSDs can be benchmarked in ways that show very different numbers. As seen in figure

1, a 90% loss in write performance may await any application that performs enough writes that the drive's internal garbage-collection limits are reached.

The magnitude of this performance drop could cause severe problems. If a production database server or file server were deployed based on SSD benchmarks that failed to anticipate these limitations, it could fail at its task if I/O capacity suddenly fell by 90% at the moment of peak I/O load.

Because of this problem, we choose our benchmark tests pessimistically. We anticipate that real-world workloads may be very random in nature, rather than sequential. We choose small I/O block sizes rather than large ones. And we benchmark drives at full capacity. Our goal is to measure the performance of drives such that there is no hidden danger of a workload that will make the drive lose much of its expected performance.

The one way in which we are not pessimistic is with parallelism. We are trying to demonstrate the performance of the I/O subsystem, assuming an application that can take advantage of it. If an application cannot, the I/O subsystem has not behaved in a surprising or unpredictable manner, which is the source of the pessimistic approach.

2) Full Capacity: SSD drives have new performance characteristics that allow old benchmarking tools to be used in ways that deliver unrealistic results. For example, vendor benchmarks often include results from empty drives, that have had no data written to them. This is an unrealistic scenario that produces artificially high throughput and IOPS numbers for both read (because the drive need not even access the flash devices) and write (because no flash page erases are needed, and also because random data can be laid out in a convenient linear format).

It might be reasonable to benchmark in a partially-empty state; for example, with 90% of the drive filled. But a benchmark scenario which leaves one giant block unused may have different performance on different drives compared to a scenario where the empty space is in fragments scattered throughout the drive. This inability to set up equally fair or unfair conditions for all drives, combined with the pessimistic outlook, leads us back to the 100% full case.

Another problem with less empty space on the drive makes it easier to achieve steady state with respect to write benchmarks (see next subsection). Because we do not wish to be misled by our own benchmarks, we choose to benchmark in a configuration that makes it easy to reach the steady state.

A system deployed in new condition may have a drive that is only 20% filled with data. This condition may allow better performance than our configuration would produce. Our pessimistic view anticipates the situation months or years later, when the system behavior changes because the drive has reached an unanticipated tipping point where performance has degraded. In our experience, the *worst-case* performance is not improved by reserving moderate amounts of free space (less than 50% of the drive capacity).

For these reasons we operate with the drive 100% full; before running performance tests we write large blocks of zeros to the drive sequentially. In our testing this has been sufficient to reproduce the performance of a drive filled with a filesystem and normal files.

New operating systems and drives support TRIM [12], a drive command which notifies the drive that a block of data is no longer needed by the operating system or application. This can make write operations on an SSD faster because it may free up sections of flash media, allowing them to be re-used with lessened data relocation costs.

Use of TRIM can sometimes increase drive performance, as it effectively (though temporarily) increases the amount of overprovisioning, allowing more efficient flash management. It can also reduce performance if TRIM commands are slow operations that block other activity [13]. TRIM can cause some unpredictability in benchmarking, because it is not possible to predict when or how efficiently the drive can reclaim space after data is released via TRIM.

Because we want to minimize unpredictability and benchmark the drive in its full state, we benchmark without any use of the TRIM command or any equivalent functionality.

3) Steady State: Disk drives offer predictable average performance: write throughput does not change after several seconds or minutes of steady running. SSDs are more complex; due to overprovisioning the performance-limiting effects of flash management on writes do not always manifest immediately. We measure random write performance by performing "warm-up" writes to the drive but not performing measurements until the performance has stabilized.

While the period of time needed to reach steady state is drive-dependent, we have found that on a full drive a test run of 20 seconds of warm-up writes following by 20 seconds of benchmark writes allows true steady-state performance to be measured.

Because the difference in throughput between peak performance and steady-state performance is so great (figure 1), an average that includes the peak performance would not meet our goals for a stable pessimistic benchmark.

4) Small-Block Random Access Benchmarks: There are several reasons to benchmark random reads and writes. The most important is that random workloads represent real-world worst-case system behavior. Systems designed with required minimum performance must be insulated from the risk of a sudden drop in performance. Such a drop can easily occur if I/O patterns become less sequential. As filesystems become fragmented, even fixed workload performance may degrade as once-sequential access patterns become more random.

Sequential workloads are not challenging for either disks or SSDs. Inexpensive drives can hit interface bandwidth limits for sequential data transfers; for workloads of this sort storage media can be chosen based on other factors like cost, power or reliability.

We also want to be able to test drive performance even when the kernel's buffer cache layer is buffering and caching I/O data. Anything other than random access patterns will start to measure the performance of the cache rather than the performance of the drive and the I/O subsystem.

We focus on small block sizes because they are more typical and challenging for disks and SSDs. Linux systems most frequently move data in chunks that match the kernel page size, typically 4KB, so data transfers of small numbers of pages (4-16KB) are most likely to be critical for overall I/O performance.

Disk-based I/O (both serial and parallel) for HPC and other large-scale applications generally requires aligned, large block transfers to achieve good performance [5]. Generally, only specialized applications are written so that all I/O strictly observes alignment constraints. Forcing programmers to do only aligned, large block requests would create an onerous programming burden; this constraint is very difficult to achieve for random access, small file workloads, even if these workloads are highly parallel.

In addition, I/O access patterns depend on the way the program is parallelized — but the application data is not always organized to optimally exploit parallel disk-based storage systems. Parallel file systems become fragmented over time, especially as they fill up, so that even if the application I/O pattern is a sequential stream, the resulting block I/O stream itself may be quite random. Although using middleware to rearrange I/O accesses to improve alignment and size is possible, in practice, this approach has achieved limited success.

We believe that our benchmarks should reflect the practical reality that parallel I/O frequently appears to the block-level storage system as small, random operations. It is currently very difficult for most storage systems to achieve consistently high performance under such workloads; systems that can perform well on these benchmarks can handle high-performance applications with difficult parallel I/O patterns.

B. Kernel and Driver-Level Enhancements

1) I/O Stack: SCSI and ATA disks have made up the majority of Linux mass storage systems for the entire existence of the Linux kernel. The properties of disks have been designed in to parts of the I/O subsystem.

Systems designed for use with disk drives can safely make two assumptions: one is that CPU time is cheap and drives are slow, and the second is that seek time is a significant factor in drive performance. However, these assumptions break down when a high-performance solid-state drive is used in place of a disk.

There are two specific areas where disk-centric design decisions cause problems. First, there are in some areas unnecessary layers of abstraction; for example, SCSI emulation for ATA drives. This allows sharing of kernel code and a uniform presentation of drive functions to higher operating system functions but adds CPU load and delay to command processing.

Second, request queue management functions have their own overhead in added CPU load and added delay. Queue schedulers, also known as elevators, are standard for all mass storage devices, and the only way to avoid their use is to eliminate the request queue entirely by intercepting requests before they enter the queue using the kernel's `make_request` function. While there are useful functions in the scheduler, such as providing fair access to I/O to multiple users or applications, it is not possible for device driver software to selectively use only these functions.

Our driver code for the PCIe SSD uses no request queue at all. Instead, it uses an existing kernel function called `make_request`, which intercepts storage requests about to enter the queue. These storage requests are simply handled immediately, rather than filtering down through the queue and incurring additional overhead. `make_request` is most often used by kernel modules that combine devices to form RAID arrays [14]. Its use by a device driver may be a misapplication, but if this mode of operation did not exist we would desire something roughly equivalent.

Additionally, our driver ignores the standard kernel SCSI/ATA layer. With the potential for one new command to process every microsecond, time spent converting from one format to another is time we cannot afford. In this case our driver, rather than interfacing to the standard SCSI or ATA layers of the kernel, it interfaces directly to the higher-level block storage layer, which stores only the most basic information about a command (i.e. where in the logical storage array data is to be read/written, where in memory that data is to be delivered/read from, and the amount of data to move).

2) Interrupt Management: Interrupt management of a storage device capable of moving gigabytes of data every second is, unsurprisingly, tricky. Each outstanding command might have an application thread waiting on its completion, so prompt completion of commands is important. It becomes doubly important because there are a fixed number of command queue slots available in the device, and every microsecond a completed command goes unretired is time that another command could have been submitted for processing. But this must be balanced against the host overhead involved in handling device interrupts and retiring commands, which for instruction and data cache reasons is more efficient if done in batches.

A typical Linux mass storage device and driver has a single system interrupt per device or host-bus adapter; the Linux kernel takes care of routing that device to a single CPU. By default, the CPU that receives a particular device interrupt is not fixed; it may be moved to another CPU by an IRQ-balancing daemon. System administrators can also set a per-IRQ CPU affinity mask that restricts which CPU is selected.

Achieving the lowest latency and best overall performance for small I/O workloads requires that the device interrupt be delivered to the same CPU as the application thread that sent the I/O request. However, as application load increases and needs to spread to other CPUs, this becomes impossible. The next best thing is for application threads to stay on CPUs that are close neighbors to the CPU receiving the interrupt. (Neighbors, in this context, means CPUs that share some cache hierarchy with one another.)

Sending the interrupt to the CPU that started the I/O request (or a close neighbor) improves latency because there are

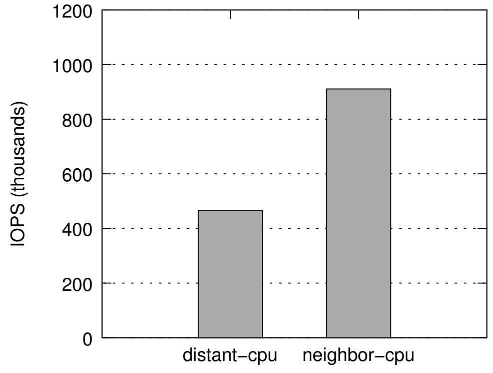

Fig. 3. Interrupt routing effects on read performance

memory data structures related to that I/O that are handled by code that starts the request as well as by code that retires the request. If both blocks of code execute on the same CPU or a cache-sharing neighbor the cache hits allow faster response.

There is also a worst-case scenario, where I/O requests are originating at a CPU or set of CPUs that share no caches with the CPU that is receiving device interrupts. Unfortunately, the existing IRQ balancing daemon seems to seek out this configuration, possibly because it is attempting to balance the overall processing load across CPUs. Sending a heavy IRQ load to an already busy CPU might seem counter-intuitive to a systems software designer, but in this case doing so improves I/O performance.

Figure 3 shows the result of running a 128-thread 512-byte read workload on one physical CPU (a neighboring set of four CPU cores) while the resulting device interrupts are sent to one of those cores and a non-neighbor CPU core.

As I/O load increases, eventually all system CPUs will be needed to perform I/O requests. This raises two problems. First, how can cache-friendly request retirement be performed by the interrupt handler if half (or more) of the requests originated on distant CPUs? Second, how can the system avoid a single CPU becoming overwhelmed by interrupt-related processing?

The best solution would be to dynamically route the interrupt such that retirement always occurs on the right CPU. This is not easy to implement, however. There is no way to re-route the interrupt signal on a fine-grained basis, and the Linux kernel has no standard mechanism for a driver to forward interrupt handling work to another CPU. Additionally, an interrupt may signal the completion of a number of I/O requests, so there is not a one-to-one relationship between an interrupt and a "correct" CPU.

We implemented two separate designs in an attempt to solve this problem. Both make use of a Linux kernel function that allows an inter-processor interrupt (IPI) to force another CPU to call a function we specify.

Our first design splits interrupt workload in two pieces, by sending an IPI to a distant CPU to force half of the retirements to occur there. We divide the command queue tag space in two, such that tag numbers 0-63 are retired on an even-numbered CPU, and tag numbers 64-127 are retired by an odd-numbered CPU. We then modified the code in the tag allocator such that tag numbers are allocated to CPUs in the same way, so that in most cases the originating and retirement CPU will be neighbors. (On a Linux machine, logical CPU numbers are assigned such that adjacent CPU numbers are not neighbors.)

```
hd=localhost,jvms=4
sd=sd1,lun=/dev/sdb,threads=128,openflags=O_DIRECT
wd=wd1,sd=sd1,xfersize=4K,rdpct=0,seekpct=random
rd=run1,wd=wd1,iorate=max,elapsed=20,forxfersize=(4k)
```

Fig. 4. VDBench configuration script

This design has several shortcomings: it assumes a system where there are two physical CPU packages, and all the cores contained within share some cache. Additionally, it assumes an even balance of I/O requests from both cores. If a disproportionate amount of requests originate from one of the two physical CPU packages, tags from the "unfriendly" side will be allocated (rather than delay the request), and the retirement will experience the cache miss delay.

Our second design is more ambitious. It tracks the originating CPU of every request, and the interrupt handler sends an IPI to every CPU core; each core is responsible for performing retirement of requests that originated there.

IV. Methodology

A. Benchmarking Tools

Most I/O benchmarking tools under Linux measure serialized file-based I/O. The set of tools that can perform both raw device and filesystem I/O and support parallel I/O are limited.

VDBench [15] supports both raw and filesystem I/O using blocking reads on many threads. It has a flexible configuration scheme and reports many useful statistics, including IOPS, throughput, and average latency.

Fio [16] supports both raw and filesystem I/O using blocking reads, Posix AIO, or Linux Native AIO (libaio), using one or more processes. It reports similar statistics to VDBench.

We use VDBench for our reported results, as it delivers more consistent results. A typical configuration file for VDBench is shown in figure 4, using 128 threads to perform 4KB random reads on a raw device.

VDBench is used both as traffic generator and measurement device. It records instantaneous and average IOPS and throughput, average and maximum command latency, and CPU load.

B. Trace-Driven Benchmarks.

Trace-driven benchmarking is making a recording of I/O activity on one system configuration, then replaying that activity on another system configuration. The promise of trace-driven benchmarks is that it allows for the evaluation of application I/O performance on a system without having access to the original application (or its data). An example of trace-driven performance evaluation with SSDs is given in [17].

However, I/O traces fail to preserve the serialization or dependencies between operations; it is not possible to reconstruct whether two operations could have proceeded in parallel on the target system if they did not do so on the traced system. When evaluating the performance of a drive that has different performance characteristics from the drives that precede it, especially with respect to support for parallel operations, trace-driven benchmarks are of questionable value.

C. PCIe SSD

We perform testing using a 96GiB (103GB) prototype Micron PCI Express SSD. This test drive uses a standard PCIe 1.0 x8 interface to connect to the host system. The drive supports a queue containing 128 commands; at any time the drive may be performing work on all 128 commands simultaneously, assuming that the commands are independent.

All flash management functions take place inside the drive; device drivers communicate with the drive using a standard AHCI command interface. Some extensions to AHCI are used by the driver to support larger command queues.

All tests are performed with a single PCIe SSD installed. (Multiple drives may be used where system bus bandwidth is available, but that configuration is not benchmarked here.)

D. Test System

All testing was performed on an Intel S5520HC system board with two Intel E5530 Xeon CPUs running at 2.40GHz. Each CPU contains four processor cores, and with hyper-threading enabled, this results in a total of 16 logical CPUs in the system.

All tests were performed using CentOS 5.4; both the standard CentOS (updated) kernel version 2.6.18-164.6.1-el5 and unmodified mainline kernel 2.6.32 were tested. Both kernels provided similar results, so only the standard kernel results are included here.

When the standard Linux AHCI driver was used, the system BIOS was configured to enable AHCI. When a dedicated block driver was used, AHCI was disabled.

E. SATA SSD

For comparison, we also test a 256GB Micron C300 SATA drive, though only 103GB were used during performance testing. The rest of the drive was filled with zeroed data to avoid the "empty drive" phenomenon where SSD performance can be inflated. The C300, as a SATA drive, supports a 32-command queue.

F. Benchmarking Configuration

Benchmarking was performed with VDBench, and unless otherwise stated VDBench was configured for 128 threads for the PCIe SSD, and 32 threads for the SATA SSD (and whenever AHCI drivers are in use).

When the standard Linux AHCI driver was used, the device queue was configured to use the "noop" scheduler. When the custom block driver was used, it bypassed the standard request queue so scheduler options have no effect.

Where filesystems were used, the drive was partitioned on 128KB block boundaries. All filesystems were created and mounted with default options.

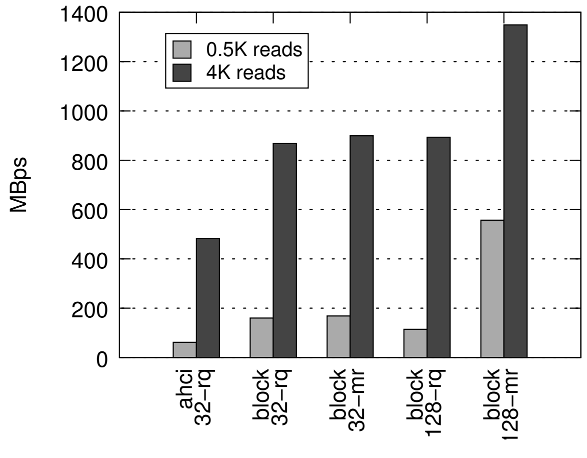

Fig. 5. Driver performance gains

V. Results and Discussion

A. I/O Stack

In order to test the performance of the I/O stack modifications, we performed small (512 byte) random reads to the PCIe SSD. Raw mode was used with the Linux buffer cache disabled (using `O_DIRECT`). With this configuration we avoid hitting bus bandwidth limits, allowing the full effect of the changes to be visible.

Figure 5 demonstrates the performance gains: the stock Linux AHCI driver, which supports a 32-command queue but uses the kernel SCSI/ATA layers, is the slowest access method. A dramatic improvement is seen by moving to a dedicated block driver that does not use the SCSI/ATA layers.

The risk of drivers short-circuiting the SCSI/ATA layers of the Linux I/O subsystem is that drivers may provide different functionality or pose a maintenance problem, as duplicate code already exists elsewhere in the kernel. However, it may be possible to retain consistent abstract interfaces while providing a fast code path for performance-sensitive read and write operations.

There is insignificant benefit in moving to a `make_request` design, which eliminates request queue overhead, while retaining a 32-command queue. Similarly, there is no benefit to quadrupling the queue size to 128 commands, while leaving the request queue overhead in place. However, these two changes combined (using `make_request` with a 128-command queue) allow another large increase in performance to 557 MBps for 512-byte reads and 1349 MBps for 4KB reads.

Drivers that short-circuit the request queue in this way are not encouraged by Linux kernel engineers. Even so, this work demonstrates that the current request queue design is insufficient for high-performance drives. It may be possible to improve the request queue to the point where it can handle such high throughput. Until then, high performance can still be achieved with the inelegant `make_request` design.

There is minimal downside to using `make_request` with the PCIe SSD as long as transfer sizes are 4KB or larger.

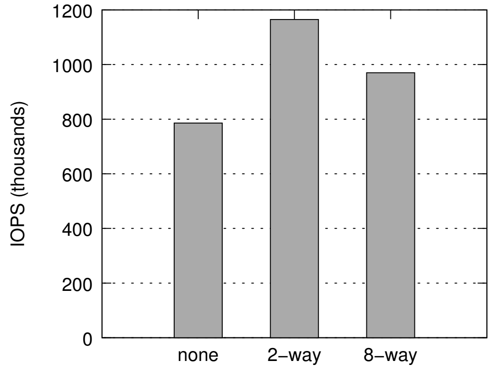

Fig. 6. IOPS improvement with interrupt handler division

Smaller writes would require read-modify-write transactions, adding overhead and possibly reducing drive life. These small transfers might be combined if the standard request queue is used. This problem will appear whenever writes are smaller than the device's native block size (the "remap unit" in section II-A).

B. Interrupt Management

We measured the performance of the interrupt management enhancements by running a 128-thread 512-byte random read test. No threads were bound to specific CPUs, which results in an even distribution of threads over the 8 CPU cores (and 16 logical CPUs). This test demonstrates the overhead added to command handling by the Linux kernel under the test configurations.

Figure 6 shows the results of the division of work by the interrupt handler. A handler with no work division performs relatively poorly, while the 2-way split handler achieves 48% higher IOPS. The 8-way split handler, which retires commands on the same CPU core as they were started, is 17% slower than the 2-way split handler.

The 8-way split fails to perform better for two reasons: first, a Linux IPI (Inter-processor interrupt) is very expensive, so the 8-way split has inherently higher overhead. Second, an L3 cache hit (which replaces a miss in the 2-way split driver) is not dramatically slower than an L2 hit, which is likely to be the best result in the 8-way split design. The overhead of such a scheme is just too high, and this design is unable to top the performance of the 2-way driver.

The improvements shown in this experiment might suggest that further improvement might be achieved with hardware that supports multiple independent interrupt requests, such that interrupt load could be spread out over several system CPUs when the device is busy. We might also consider a hardware design that is able to track originating-CPU information for each request, and send an interrupt to the correct CPU (or a neighbor) when the command is completed.

Interrupt load can also be lowered by using hardware interrupt mitigation techniques [18]. We performed experiments with different interrupt mitigation settings on the PCIe SSD. Normally, interrupt mitigation is used to reduce the interrupt load so that I/O can be completed in efficient batches. However, we found that the cost of delaying command completion (both the latency cost to the completed command as well as the opportunity cost for a command that could have occupied its slot) was greater than any benefit, and that minimal interrupt mitigation (4 commands per interrupt) worked best.

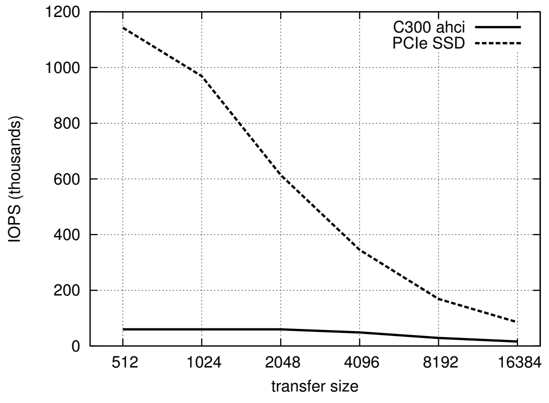

Fig. 7. Read IOPS

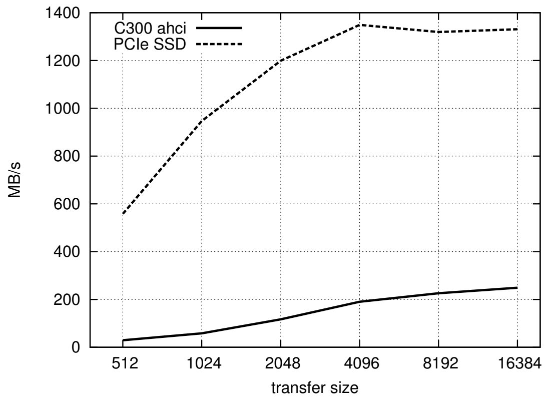

Fig. 8. Read throughput

C. Final Performance

After all of our improvements, we were able to demonstrate performance of over 1.1 million IOPS on a single drive performing random 512 byte reads, and we were able to hit the device's PCIe bandwidth limit of 1.4 GBps with 4KB reads.[1] Figure 7 shows IOPS when benchmarking random reads using 128 threads. Figure 8 shows the total read throughput. Figure 9 shows the average latency.

Write performance is also quite impressive: Figure 10 shows the steady-state IOPS when performing random writes. Figure 11 shows the total write throughput. Figure 12 shows the write latency. These measurements show little variability; different test runs show changes of less than 1%.

Figure 13 shows the throughput of several filesystems with the PCIe SSD using different numbers of client threads

[1]The theoretical limit of the bus is 2.0 GBps, but packet overhead, small (cacheline-sized) payloads, and prototyping compromises consume 30% of this.

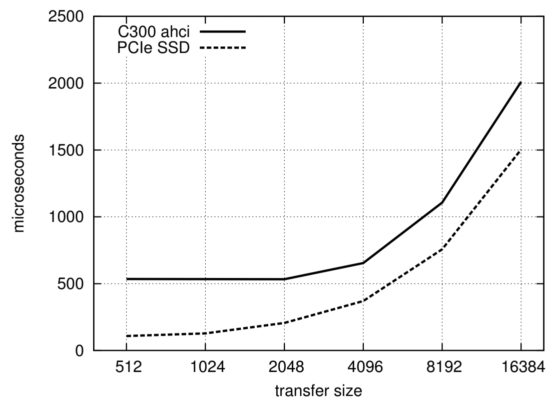

Fig. 9. Read latency

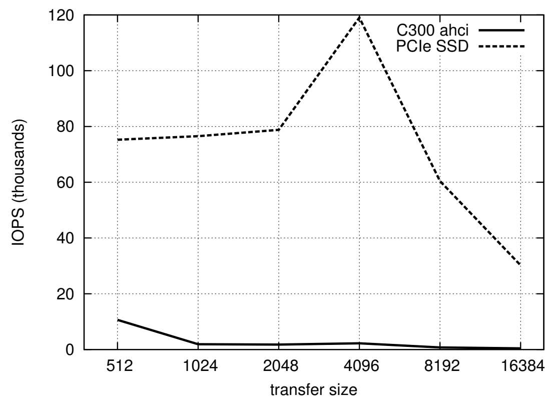

Fig. 10. Write IOPS

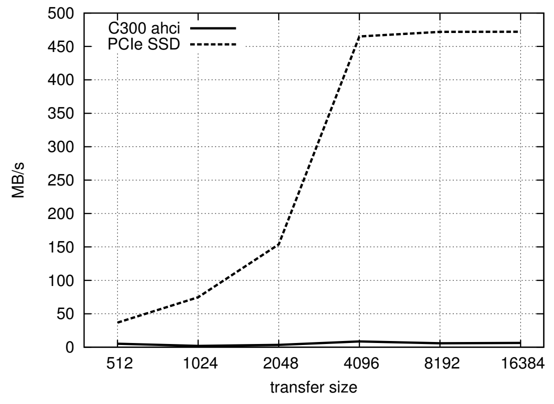

Fig. 11. Write throughput

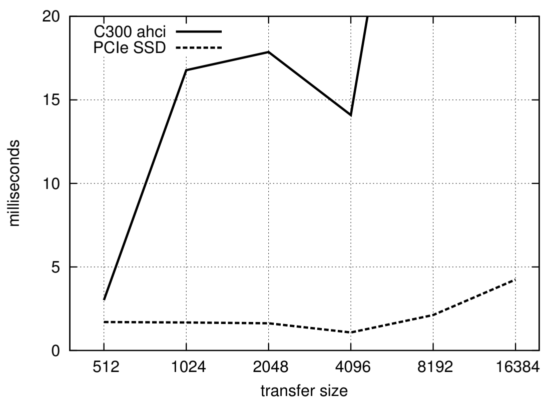

Fig. 12. Write latency

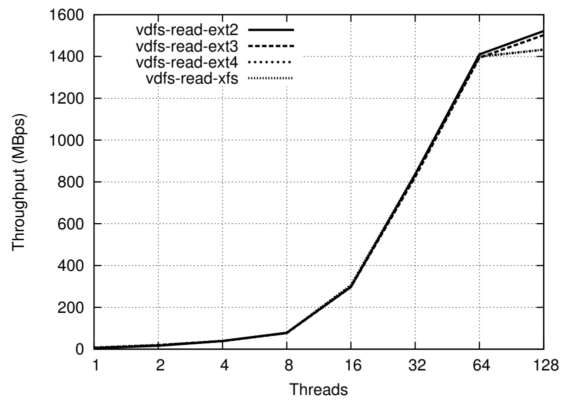

Fig. 13. Filesystem read throughput

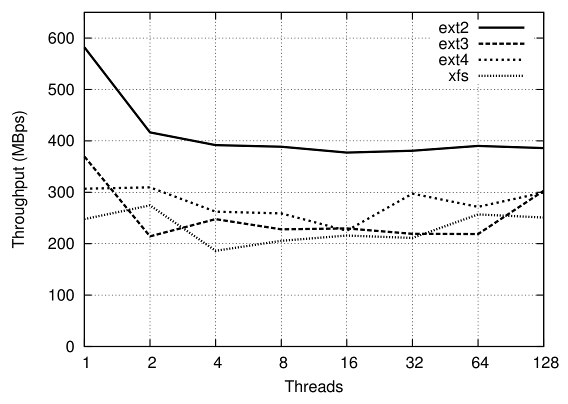

Fig. 14. Filesystem write throughput

performing 4KB random reads. The different filesystems show little variation in performance when reading files; all filesystems hit the drive bus bandwidth limit with 64 or more threads.

Figure 14 shows the filesystem random write throughput. Write performance shows the benefit of the write buffering function of the Linux buffer cache: single-thread write workloads are able to perform as well or better as multi-threaded workloads. When writing, the journaling filesystems ext3, ext4 and xfs have lower effective throughput than ext2, which performs no journaling.

Filesystems such as YAFFS or JFFS2 are designed to interface directly to NAND flash hardware, and implement their own flash translation layer, including journaling, garbage collection, and wear-leveling. SSDs that use disk-like interfaces implement their own FTL, and are unsuitable for use with these filesystems.

Though these performance numbers are impressive, It would be a straightforward exercise to imagine a device with greater bus bandwidth and even larger command queue, which could result in million-IOPS performance on 4KB reads for a single drive.

D. Buffer Cache Overhead

For most uses, the Linux buffer cache is not optional. We consider whether the overhead of the buffer cache is still worthwhile with a low-latency SSD.

We measured the latency of 4KB random reads while using a low-overhead filesystem (ext2) and with buffered raw mode and unbuffered raw mode (using `O_DIRECT`). There was no measurable difference in latency; in each case the average latency was 153 microseconds, while buffer cache hit latency is only 7 microseconds.

Assuming that the system RAM used for buffer cache could not be better used for other purposes, it is clear that the buffer cache is still useful when using fast SSDs.

E. Additional Discussion

1) Idle I/O Time: For disk-based systems, speculative data access can be detrimental to performance, because a seek to the wrong end of a disk might penalize a non-speculative access that arrives immediately afterwards.

Because SSDs have no seek penalty it seems worthwhile to revisit this idea. There are several potential uses for idle time on an SSD that might help system performance.

The operating system might be able to use the excess parallel read capabilities during times of less than full utilization, by doing more aggressive prefetching than would be done on a fully-loaded system. A balance would need to be found such that such prefetching wouldn't use up otherwise-needed CPU time or bus bandwidth, but such a feature could significantly improve the performance of applications that perform serialized sequential reads, which could be valuable if changing the application is impractical.

Additionally, the system could take advantage of times when the drive is not performing many writes, and use that time to write dirty buffers (operating system caches of modified file data). This would be much better than only writing out dirty buffers due to memory pressure, because those events may coincide with times when the drive's write capacity is already fully utilized, resulting in a system slowdown.

2) Kernel Enhancements: There are several areas that could be improved to make it easier to get good performance out of SSDs similar to the PCIe SSD examined in this paper.

Most significantly, additional IRQ handling features would be useful. For example, expensive cache misses that slow down requests could be avoided if there were a mechanism to quickly start an interrupt handler near the CPU core that started an I/O request. The IRQ balancing daemon could also be improved: we have demonstrated circumstances under which interrupts sometimes need to be routed to CPU cores doing the most I/O.

The current request queue scheme is available in an all-or-nothing form, as the functions that implement its various pieces (request merging, queue plugging) are not available for driver use without modifying the core Linux kernel source code. It would be helpful if an *a la carte* access model were adopted instead, where both device drivers and system administrators were able to pick and choose which functions should be utilized. For example, it should be possible to use a request queue with per-user balancing, but no request re-ordering. Additionally, it would be useful to be able to turn on request merging only when the device is already busy (and requests cannot be serviced immediately).

Additionally, there is code in newer kernels that makes the request queue more SSD-aware. If Linux vendors that use older kernel branches adopted some of these newer features, good SSD performance would be easier to achieve.

If possible, the current SCSI and ATA layers should be streamlined. Systems with the current design will not be able to extract full performance from fast SSDs unless they utilize device drivers that intentionally bypass these layers.

Finally, device drivers should be able to do more with SMP and NUMA machines. As I/O devices appear that require multiple CPUs to service them, it must be possible to write multi-processing drivers. Such drivers need the ability to schedule tasks on other CPUs and knowledge of the cache hierarchy between CPUs.

VI. Conclusions

In this paper, we have discussed the nature of high-performance SSDs both from a benchmarking and Linux I/O performance perspective. Our analysis has found that there is a need for a more pessimistic benchmark approach, and we have adopted simple guidelines that allow us to benchmark drives under realistic but difficult workloads.

We have used this approach to guide us in achieving performance improvements in the Linux kernel I/O block driver interface, and we have demonstrated performance exceeding one million IOPS, a significantly higher result than can be achieved today with any single drive. We used these results to demonstrate areas of future development for the Linux kernel I/O design and its driver interfaces.

These results indicate that SSDs are a good fit for high-performance systems, and demonstrate the suitability of SSD technology for demanding single machine I/O workloads as well as parallel I/O needs on future HPC systems.

Acknowledgment

This work was supported in part by National Science Foundation grants CCF-0621462 and IIP-0934396.

References

[1] 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.

[2] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. New York, NY, USA: ACM, 2009, pp. 1–9.

[3] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2009, pp. 181–192.

[4] B. Dees, "Native command queuing - advanced performance in desktop storage," *Potentials, IEEE*, vol. 24, no. 4, pp. 4–7, Oct.-Nov. 2005.

[5] G. Greider. (2006, May) Hpc i/o and file systems issues and perspectives. http://www.dtc.umn.edu/disc/isw/presentations/isw4_6.pdf.

[6] H. Newman. (2009, May) What is hpcs and how does it impact i/o. http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf.

[7] (2009, November) Nsf awards $20 million to sdsc to develop gordon. http://www.sdsc.edu/News Items/PR110409_gordon.html.

[8] A. Sahai, "Performance aspects of raid architectures," in *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, Feb 1997, pp. 321–327.

[9] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[10] H. N. Ken Chen, Rohit Seth, "Improving enterprise database performance on intel itanium," in *Proceedings of the Linux Symposium*, 2003.

[11] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *In Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX Association, 2000, pp. 317–327.

[12] F. Shu. (2007) Notification of deleted data proposal for ata8-acs2. http://t13.org/Documents/UploadedDocuments/docs2007/e07154r0-Notification_for_Deleted_Data_Proposal_for_ATA-ACS2.doc.

[13] J. Corbet. (2009) The trouble with discard. http://lwn.net/Articles/347511/.

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[15] H. Vandenbergh. Vdbench: a disk and tape i/o workload generator and performance reporter. http://sourceforge.net/projects/vdbench/.

[16] J. Axboe. Fio - flexible io tester. http://freshmeat.net/projects/fio/.

[17] S. Park and K. Shen, "A performance evaluation of scientific i/o workloads on flash-based ssds," in *Workshop on Interfaces and Architectures for Scientific Data Storage (LASDS '09)*, 2009.

[18] I. Kim, J. Moon, and H. Y. Yeom, "Timer-based interrupt mitigation for high performance packet processing," in *Proceedings of 5th International Conference on High-Performance Computing in the Asia-Pacific Region, September, Gold Coast, Australia*, 2001.