High Performance SSDs Under Linux

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Eric Seppanen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

David Lilja

February, 2011

Acknowledgements

There are many people that made my time at the University of Minnesota exciting and fulfilling. I would like to thank my classroom instructors, especially Wei Hsu, Kia Bazargan, Eric Van Wyk, and Yongdae Kim.

I would like to extend additional thanks to Matt O'Keefe and David Lilja for helping me polish and publish my research.

Dedication

For Tesla, who persuades me that I can do anything.

Abstract

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 random read IOPS with throughput of 1.4GBps.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this paper we discuss the properties of NAND flash solid state drives (SSDs). SSDs are storage devices used in place of hard disks in environments for which disks are too slow, bulky, or fragile. As the price of NAND flash falls relative to disk, more and more environments become suitable for SSD usage, from lightweight laptops to high-performance server systems.

We examine the performance properties of high-performance SSDs suitable for use in enterprise server environments. High-performance SSDs can make use of highly parallel, low-latency read and write operations to provide I/O performance that can surpass disk drives by two to three orders of magnitude. However, SSDs can have unusual characteristics that can cause this performance to vary significantly by workload.

These properties reveal a need for improved benchmarking guidelines for high-performance solid state drives. We develop benchmarking guidelines and methodologies and show results of high-performance and consumer SSDs under adverse conditions.

Armed with high-performance SSD benchmarks, we examine SSD performance under the Linux operating system, and introduce several ways in which Linux kernel I/O performance can be improved when using high-performance solid-state storage.

- Chapter 2 offers an introduction to solid state storage, Linux I/O, and benchmarking of mass storage devices.
- In Chapter 3 we describe our improvements to solid state benchmarking and Linux kernel I/O techniques.

- Chapter 4 describes our test configuration and performance measurement techniques.
- Chapter 5 presents performance results and a discussion of the significance of those results.
- Chapter 6 presents a final summary of the analyses presented in this thesis.

Chapter 2

Background

2.1 Solid State Drives

A Solid State Drive (SSD) is a mass storage device typically used as a replacement for hard disk drives. Most SSDs use the same hardware and software interface as hard disk drives, allowing them to be simple plug-in replacements for hard disks.

SSDs have dramatically different performance characteristics from hard disks, however. They typically have higher performance than disk drives, though at significantly higher cost per gigabyte.

SSDs are constructed from many NAND flash devices, along with a controller device that allows host operating system software to access a flat storage space. SSD controllers may include an embedded microprocessor and a DRAM buffer, and possibly some sort of battery backup to allow safe operation across unexpected power loss. Figure 2.1 illustrates typical SSD architecture.

2.1.1 NAND Flash

NAND flash devices are the storage medium in all mainstream SSDs. NAND flash can perform three basic operations: page read, page write, and block erase. A typical device is composed of many large (i.e. 512KB) blocks, each of which is subdivided into smaller (typically 4KB) pages. Pages can be read many times, but written only once before the containing block must be erased. Table 2.1 shows typical page read, write, and erase times for a large NAND flash device [1].

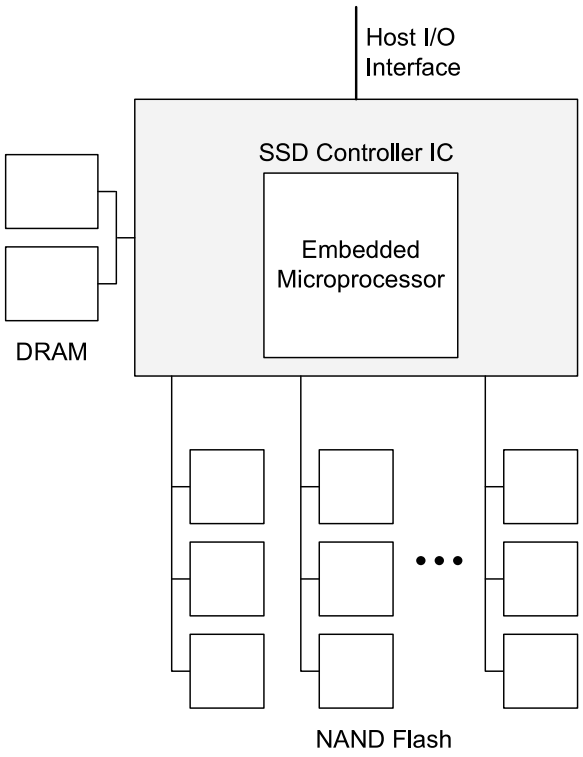

Figure 2.1: SSD Architecture

NAND flash blocks are contained within a flash plane, and multiple planes make up a die. Multiple dies may be combined within a package.

There are two common types of NAND flash device available: SLC and MLC. SLC (Single Level Cell) devices store one bit of user data per gate, and have a lifespan of roughly 100,000 program/erase cycles. MLC (Multi-Level Cell) devices store multiple (usually 2) user bits as an analog value within a single gate, allowing more user bits stored per die, but have a shorter lifespan, roughly one-tenth that of an equivalent SLC device.

2.1.2 FTL

Because all of the pages composing a block must be erased at once, NAND flash contents cannot be overwritten in place like some other read-write media. Instead, a logical-to physical mapping must be established, allowing user data with a known logical address

Operation	MLC time	SLC time
Page Read	50μs	20μs
Page Program	650μs	220μs
Block Erase	2000μs	1500μs

Table 2.1: NAND Flash Operation Times

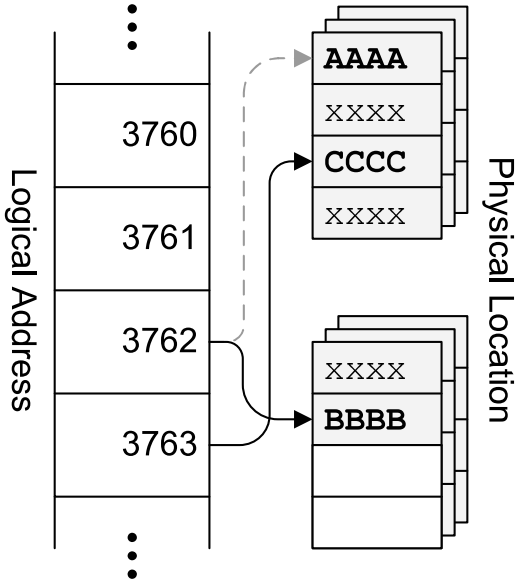

Figure 2.2: FTL Logical/Physical Mapping

to be stored at any physical location.

The management of data storage and retrieval is performed by a Flash Translation Layer (FTL). The FTL algorithms take an active part in placing and retrieving data from the flash devices. The FTL manages the physical location of user data (which flash device and location within that device stores a particular unit of data), the erasing of NAND blocks for re-use, and any data migration needed to move live data to another location. Figure 2.2 illustrates this mapping, showing static data stored in one logical location (3763) while new data is written into another logical location (3762). This logical overwrite is accomplished by simply changing the table to supersede the former contents (which remain in their original location).

FTL algorithms may be implemented in hardware, in embedded software, or as part of the host operating system or drivers, or any combination of the three.

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

SSD read operations are straightforward: the FTL must find the physical location of a user data block, retrieve the data, and return it to the host.

SSD write operations can involve escalating complexity. In the simplest case, the FTL must find an available physical location, then write the new data to that location. If physical locations are becoming scarce, more work may be necessary. Flash blocks may need to be erased, which may involve migrating live data from one location to another to clear out an entire block. The relocation of live data and use of a block erase to reclaim partially used flash blocks is known as garbage collection.

An SSD may have background tasks running independently of host requests. For example, blocks containing "dead" pages are desirable targets for future erase to store fresh data because less copying is needed to migrate the block's live data. Proactive migration of any live data remaining in these blocks can shorten latency of future writes.

SSDs typically advertise less logical capacity than there is physical flash media. This "overprovisioning" allows a logical page overwrite to finish quickly. The old physical location is simply marked "dead" in the FTL tables, and the new data is written to a new physical location. With sufficient overprovisioning, the garbage collection of dead pages may be postponed to a later time, allowing write operations to finish in less time (though sustained write load will eventually have to compete with background garbage collection activity).

Overprovisioning also allows page or block failures without reducing the available capacity. As flash page and block failures are common, failures must be tolerated for an SSD to maintain the same advertised storage capacity over its entire lifetime.

2.1.3 SSD Lifespan Management

For compatibility with hard disks, a solid-state drive must present the apparent ability to write any part of the drive an arbitrary number of times. Because NAND flash

blocks wear out relatively quickly, the FTL must 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].

Wear-leveling can be conceptually divided into "dynamic" wear leveling, which monitors each block's erase count when selecting partially-dead blocks for migration and erase, and "static" wear-leveling, which migrates data from blocks that contain no "dead" blocks in order to push low-erase-count blocks back into the available pool.

With good wear-leveling, all flash blocks will wear out at the same time. Assuming minimal premature block failures, the lifespan of an SSD can be extended, such that the drive lifespan is equal to the total aggregate block program/erase cycles of all flash devices within. For example, a 64GB SSD built from NAND flash devices with a lifespan of 100,000 program/erase cycles will have a total lifespan of 6.4 EB written (100,000 * 64GB). Reads have minimal impact on SSD lifespan.

Background tasks affecting drive lifespan and future write latency must be carefully balanced against foreground read/write operations, as both tasks require exclusive access to a particular bus and device for a certain time.

2.2 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.

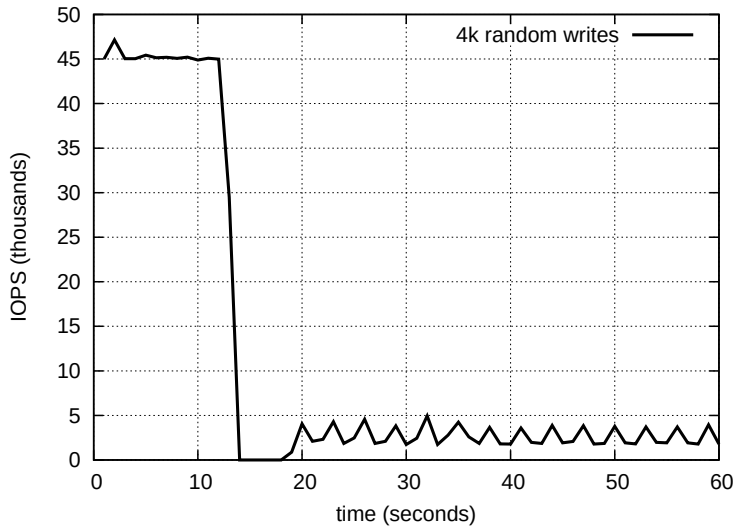

Figure 2.3: Write performance drop-off

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

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 [3], and these can cause very large performance drops after many seconds, or even minutes or hours of deceptively steady performance. Figure 2.3 shows the drop-off in write performance of a SATA SSD after a long burst of of random 4KB write traffic.

2.3 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].[1]

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 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 4.1 million bytes per second (MBps). While this would be impressive by itself, a device with 2-way parallelism could achieve 8.2 MBps; 4-way 16.4 MBps, 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.

2.4 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

[1] The fact that the drive can process commands out of order implies that drive queues are poorly named, as they are not queues in the usual sense.

(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 may also 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 require less 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.

2.5 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 for some workloads. Along with greater parallelism, this means that many fewer SSDs must be used (and managed) to achieve the same performance as disk drives.

2.6 Linux I/O

2.6.1 The Linux Kernel I/O Stack

The Linux kernel I/O stack is shown in figure 2.4. 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.[2] The filesystem 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. This generic layer includes a request queue, where scheduling decisions are made. Requests exiting the queue typically travel through a SCSI layer, which is emulated if an ATA device driver is present, and 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.

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

[2] Most filesystem designs still assume that the logical block location is the physical location, and act to minimize disk seek activity by placing associated data in nearby logical locations.

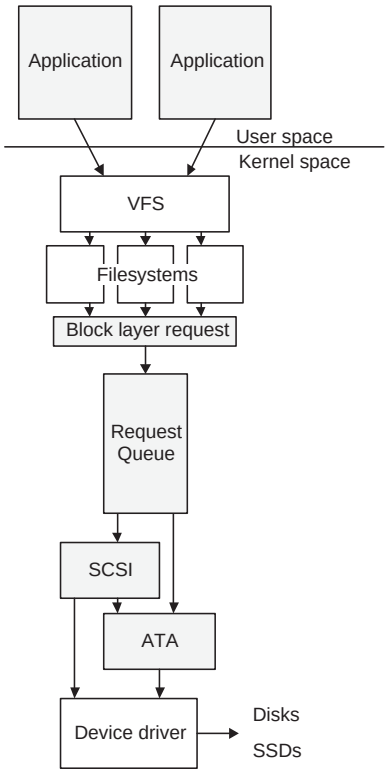

Figure 2.4: Linux I/O stack

cost of the CPU time needed to lock and 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 enterprise production systems.

2.6.2 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 pending writes, now buffered, can be handled in parallel. Therefore, the serialization of writes by an application has very little effect; at the device driver and device level the writes will appear parallelized.[3]

Real-world single-computer applications exhibiting parallelism are easy to come by.

[3] The automatic parallelization of writes has a useful side-effect: as long as there is parallel capacity to spare, the actual hardware write latency is relatively unimportant. Application performance will be unaffected unless latency becomes so severe that all command queue slots become occupied.

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 request processing.

HPC systems support more sophisticated methods of performing parallel I/O [11], but those interfaces are not supported at the Linux kernel 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.

2.6.3 Linux 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.

2.6.4 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.

2.7 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 in time.

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 (and likely to be serviced from the buffer cache).

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.

2.8 Recent Work

Solid state drive performance has been addressed in several recent papers.

In [12], the authors construct a model of SSD performance, and develop a kernel-based SSD simulator. This simulator is capable of testing performance by running real applications, a significant advantage over trace-based simulators.

In [13], the authors examine a hybrid SSD and disk storage system, exploiting parallelism to achieve best performance. Similarly, [14] demonstrates a hybrid system that allows a large inexpensive disk storage array to retain some of the high-performance I/O characteristics of SSDs.

Another hybrid system is described in [15], this time constructed from NAND flash memory paired with phase-change memory. In this design the NAND flash is the inexpensive, low-performance component, with the phase-change memory used to improve performance.

In [16], the authors examine the problem of SSD wear life, and propose a disk-based write log/cache to reduce write amplification during random write loads.

Chapter 3

Proposed Solutions

3.1 SSD-Aware Benchmarking

3.1.1 Pessimistic Benchmarking

SSDs can be benchmarked in ways that show very different numbers. As seen in figure 2.3, 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.

3.1.2 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, leaves the 100% full case as the best option.

Another problem with less than full SSDs is that the empty space makes it harder to achieve steady state performance on 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.

For these reasons we operate with the drive 100% full; before running performance tests we write large blocks of zeros to the drive sequentially.[1] While a sequential fill is not perfectly representative of a drive filled using random writes representative of the test workload, when paired with a benchmark "warm-up" period 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 [17], a drive command which notifies the drive that a block of data is no longer needed by the operating system

[1] Strictly speaking, random data should be used. We allow this shortcut only because it is known that the drives tested do not detect and optimize storage of zeroed data blocks.

or application. This can make write operations on an SSD faster because it may free up sections of flash media, reducing write amplification and allowing faster garbage collection.

Use of TRIM might increase drive performance, as it effectively (though temporarily) increases the amount of overprovisioning, allowing more efficient flash management. It might 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.

3.1.3 Steady State

Disk drives offer predictable average performance: write throughput does not change after many seconds or minutes of steady-state operation. 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.

The period of time needed to reach steady state is drive-dependent. The random write performance should be completely stabilized once write volume roughly equivalent to the drive size have been performed. This guarantees that the FTL logical/physical map is thoroughly scrambled, placing the greatest load on the garbage collector. Practical tests have shown that on a drive already filled by sequential writes this stabilization time can be reduced to about one-third of the full-drive write time (about three minutes in this case).

Read-only performance tests might require a warm-up period if the drive can cache read data; as the drive under test does not, the warm-up period is skipped during random read tests.

Because the difference in throughput between peak performance and steady-state performance is so great (figure 2.3), an average that includes the peak performance would not meet our goals for a stable pessimistic benchmark.

3.1.4 Small-Block Random Access Benchmarks

There are several reasons to benchmark random reads and writes. The most important is that random workloads are more representative than sequential workloads of real-world system behavior. Furthermore, 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.

Random reads and writes are difficult for different reasons. Though easily parallelized (see section 2.6.2), random writes take extra work on flash due to wear leveling and garbage collection. Reads are less complex for the device, but still require the FTL to perform logical to physical lookups, error correction, and resource contention. Random read workloads are often more difficult for applications to parallelize.

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. The buffer cache will complete requests that present temporal or spatial locality, passing through a more randomized set of requests to the drive.

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.

3.2 Kernel and Driver-Level Enhancements

3.2.1 I/O Stack

SCSI and ATA disks have made up the majority of Linux mass storage systems for the entire lifetime 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 disks 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 normally used for all mass storage devices, and the only way to avoid their use is to bypass the request queue entirely. 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.

To avoid locking and elevator overhead, 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 [18]. Its use in a device driver may be unusual, but if this mode of operation did not exist we would desire something roughly equivalent.

Additionally, our driver bypasses 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).

3.2.2 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

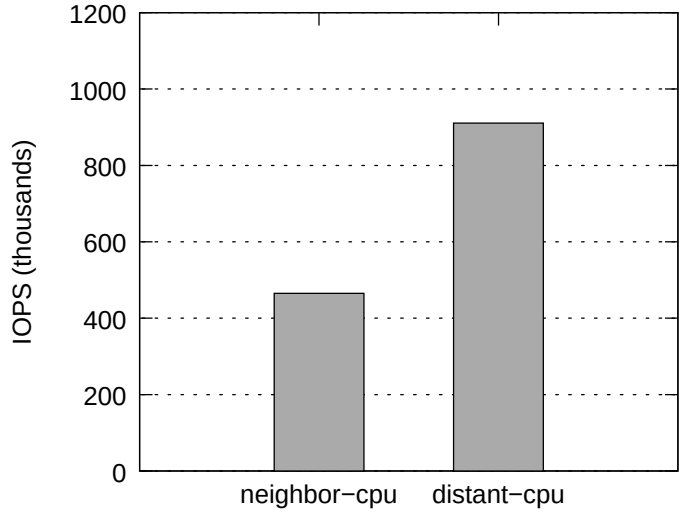

Figure 3.1: Interrupt routing effects on read performance

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 memory data structures related to that I/O that are handled during request initiation as well as during retirement. 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 3.1 shows the result of running a 128-thread 512-byte read workload on one physical CPU package (four CPU cores sharing an L2 cache) while the resulting device interrupts are sent to one of those cores vs 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.)

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.

Chapter 4

Methodology

4.1 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 is limited.

VDBench [19] 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 [20] supports both raw and filesystem I/O using blocking roads, 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 than fio (less variation from sample to sample). A typical configuration file for VDBench is shown in figure 4.1, using 128 threads to perform 4KB random reads on a raw device.

4.2 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

```
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)
```

Figure 4.1: VDBench configuration script

example of trace-driven performance evaluation with SSDs is given in [21].

Unfortunately, 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.

4.3 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 to support larger command queues.

The prototype PCIe SSD does not support ATA TRIM or similar functionality (also known on Linux as *discard*) that would allow filesystem or application code to mark data regions as discarded. None of the benchmarks used attempt to evaluate TRIM/discard performance.

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.)

4.4 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.

4.5 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 an AHCI/SATA drive, supports a 32-command queue.

4.6 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.

Chapter 5

Results and Discussion

5.1 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 avoided hitting bus bandwidth limits, allowing the full effect of the changes to be visible.

Figure 5.1 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 ("ahci 32-rq" in the figure). A dramatic improvement is seen by moving to a dedicated block driver that does not use the SCSI/ATA layers ("block 32-rq").

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 ("block 32-mr" in figure 5.1). Similarly, there is no benefit to quadrupling the queue size to 128 commands, while leaving the request queue overhead in place ("block 128-rq"). 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

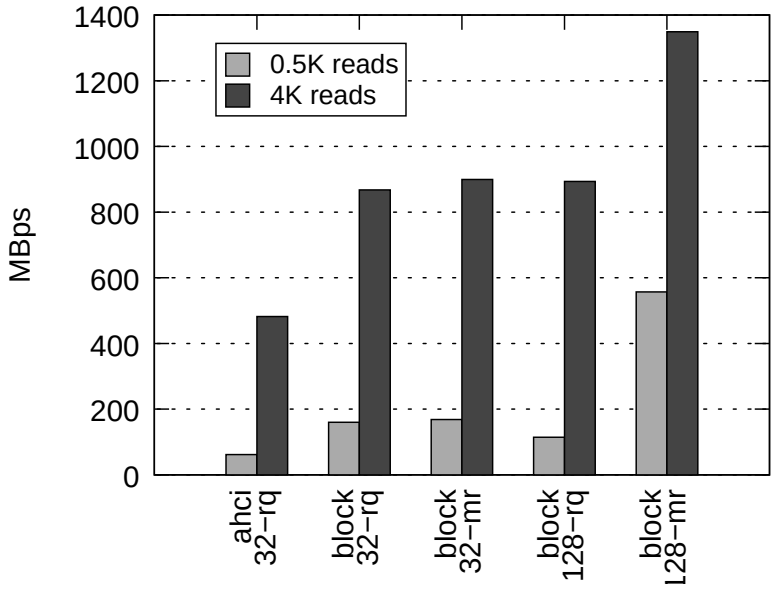

Figure 5.1: Driver performance gains

reads ("block 128-mr").

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.

5.2 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 (16 logical CPUs with hyper-threading enabled). This test demonstrates the overhead added to command handling by the Linux kernel under the test configurations.

Figure 5.2 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

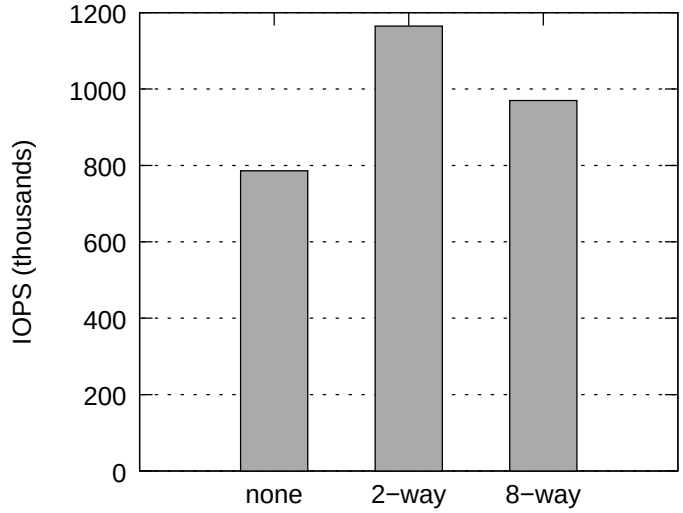

Figure 5.2: IOPS improvement with interrupt handler division

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 (interprocessor interrupt) has significant latency, so the 8-way split has inherently higher overhead. Second, an L3 cache hit (which replaces a miss in the 2-way split handler) 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 handler.

The improvements shown in this experiment might suggest that further improvement might be achieved with hardware that supports multiple independent interrupt requests. A multiple-interrupt design[1] such that interrupt load could be more easily 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 [22]. We performed experiments with different interrupt mitigation settings on the PCIe SSD. Normally, interrupt mitigation is used to reduce the interrupt load so

[1] Multiple interrupts per device are possible today. PCI Express devices allow multiple MSI-X interrupts to be directed to different CPUs.

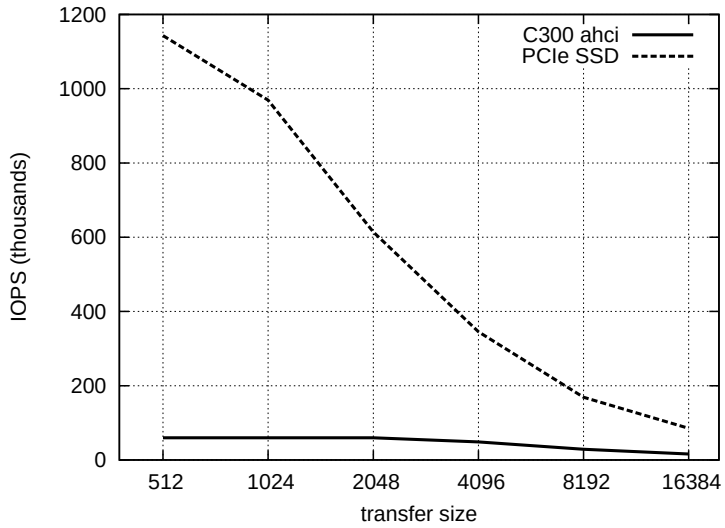

Figure 5.3: Read IOPS

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.

5.3 Performance Measurement and Improvements

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.[2] Figure 5.3 shows IOPS when benchmarking random reads using 128 threads. Figure 5.4 shows the total read throughput. Figure 5.5 shows the average latency.

Write performance is also quite impressive: Figure 5.6 shows the steady-state IOPS when performing random writes. Figure 5.7 shows the total write throughput. Figure 5.8 shows the write latency. These measurements show little variability; different test

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

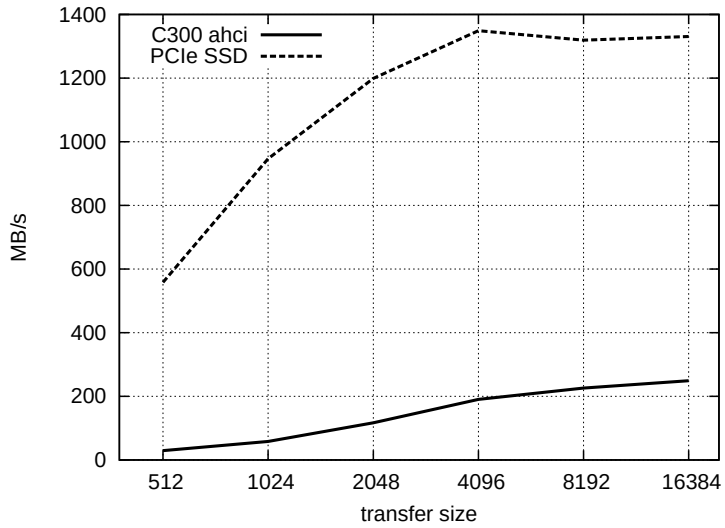

Figure 5.4: Read throughput

runs show changes of less than 1%.

Figure 5.9 shows the throughput of several filesystems with the PCIe SSD using different numbers of client threads 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 5.10 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 as or better than multi-threaded workloads. When writing, the journaling filesystems ext3, ext4 and xfs have lower effective throughput than ext2, which performs no journaling.

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.

5.4 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.

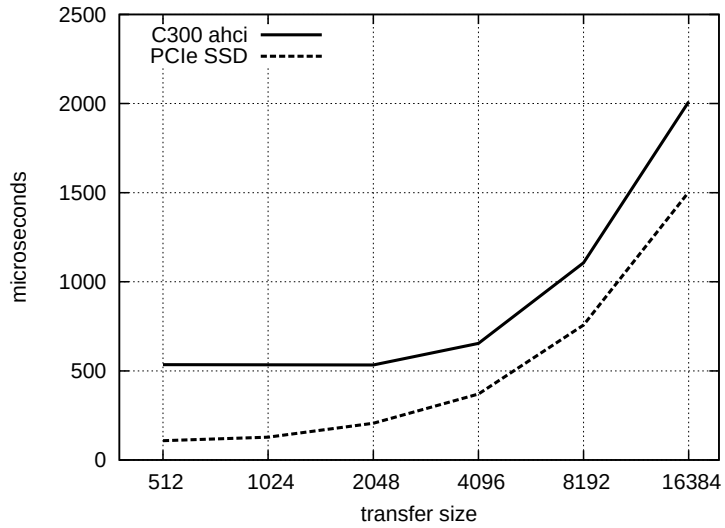

Figure 5.5: Read latency

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 miss 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.

5.5 Additional Discussion

5.5.1 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 prefetching

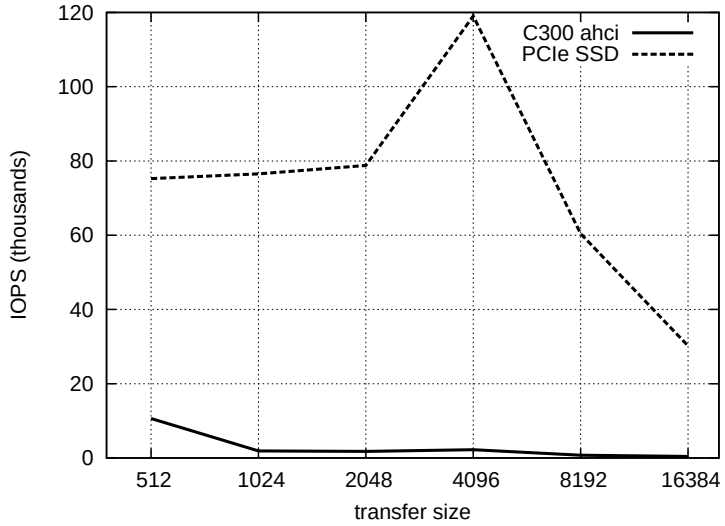

Figure 5.6: Write IOPS

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.

5.5.2 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.

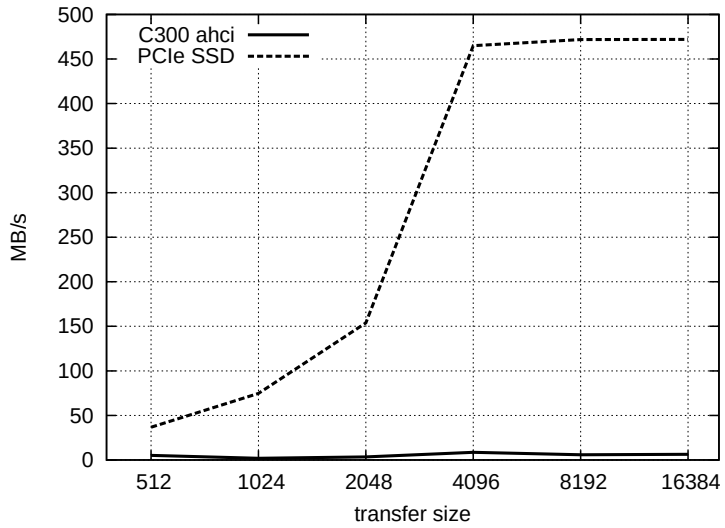

Figure 5.7: Write throughput

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

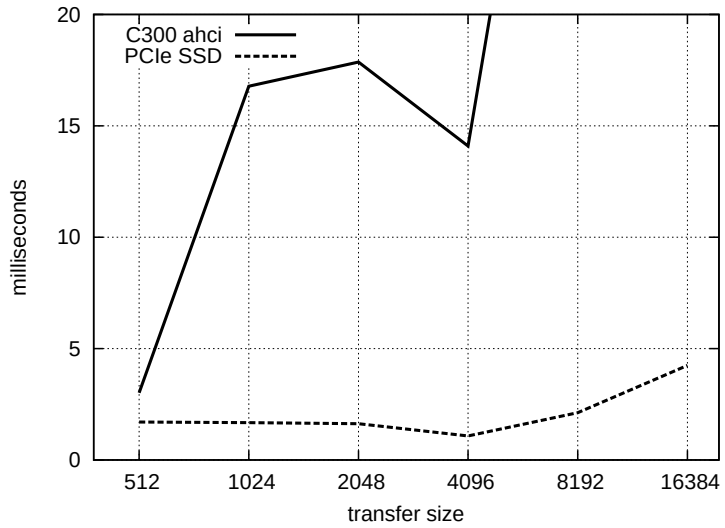

Figure 5.8: Write latency

other CPUs and knowledge of the cache hierarchy between CPUs.

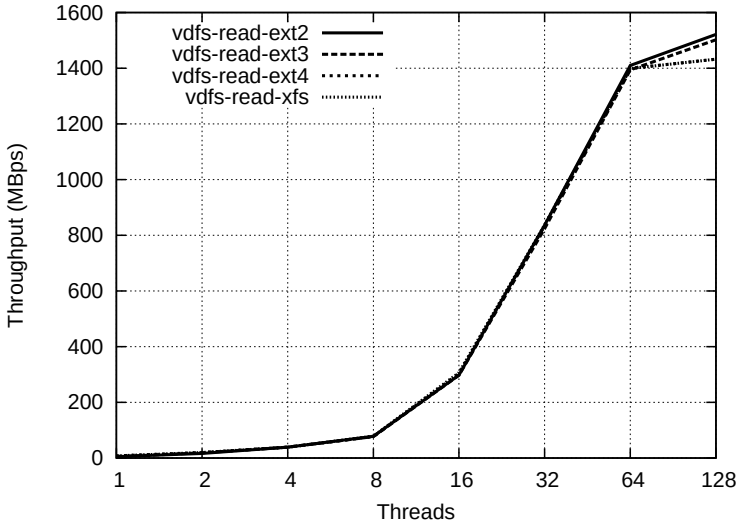

Figure 5.9: Filesystem read throughput

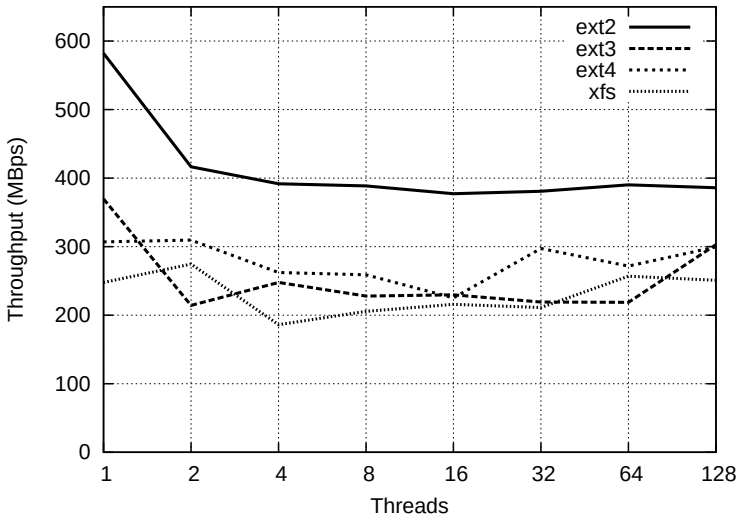

Figure 5.10: Filesystem write throughput

Chapter 6

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.

References

[1] Tn-29-25: Improving performance using two-plane commands. http://download.micron.com/pdf/technotes/nand/tn2925.pdf, September 2008.

[2] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, New York, NY, USA, 2009. ACM.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[4] B. Dees. Native command queuing - advanced performance in desktop storage. *Potentials, IEEE*, 24(4):4–7, Oct.-Nov. 2005.

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

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

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

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

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

[10] Hubert Nueckel Ken Chen, Rohit Seth. Improving enterprise database performance on intel itanium. In *Proceedings of the Linux Symposium*, 2003.

[11] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *In Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.

[12] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 15–26, New York, NY, USA, 2010. ACM.

[13] Xiaojian Wu and A.L. Narasimha Reddy. Exploiting concurrency to improve latency and throughput in a hybrid storage system. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:14–23, 2010.

[14] Jinsun Suk and Jaechun No. hybridfs: integrating nand flash-based ssd and hdd for hybrid file system. In *Proceedings of the 10th WSEAS international conference on Systems theory and scientific computation*, ISTASC'10, pages 178–185, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[15] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[16] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th*

USENIX conference on File and storage technologies, FAST'10, page 8, Berkeley, CA, USA, 2010. USENIX Association.

[17] Frank Shu. Notification of deleted data proposal for ata8-acs2. `http://t13.org/Documents/UploadedDocuments/docs2007/e07154r0-Notification_for_Deleted_Data_Proposal_for_ATA-ACS2.doc`, 2007.

[18] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

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

[20] Jens Axboe. Fio - flexible io tester. `http://freshmeat.net/projects/fio/`.

[21] Stan Park and Kai 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.

[22] 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.

Appendix A

Glossary and Acronyms

Care has been taken in this thesis to minimize the use of jargon and acronyms, but this cannot always be achieved. This appendix defines jargon terms in a glossary, and contains a table of acronyms and their meaning.

A.1 Glossary

- **Direct I/O** – I/O that bypasses operating system buffers or caches, going directly to/from a storage device.
- **Discard** – In Linux, a request that a storage device discard some data. See *TRIM*.
- **Elevator** – An algorithm for scheduling I/O requests.
- **Garbage Collection** – In an SSD, finding underutilized flash blocks and migrating any live data remaining to another location so the entire block can be re-used.
- **Hyper-threading** – A technique allowing multiple logical microprocessors to share physical execution hardware, allowing better utilization.
- **make_request** – In Linux, a kernel function used to intercept I/O requests before they enter the request queue.
- **Overprovisioning** – Advertising less total storage space than is physically available; allows more efficient FTL and lower write amplification.

- **PCI Express or PCIe** – A computer bus standard constructed from one or more pairs of high-speed serial links.
- **Request Queue** – In Linux, a software queue used to store, merge, and reorder I/O requests.
- **TRIM** – In ATA devices, a request that the device discard some data. See *Discard*.
- **Write Amplification** – A measurement of the amount of NAND flash write activity required to satisfy a host write. May vary due to the need to migrate live data before re-using a flash block.
- **Wear Leveling** – Balancing write activity over all flash blocks so no block wears out prematurely.

A.2 Acronyms

Table A.1: Acronyms

Acronym	Meaning
AHCI	Advanced Host Controller Interface; a standardized hardware and software interface between a host computer and a mass storage adapter for SATA drives.
AIO	Asynchronous I/O; a software mechanism allowing a single thread to maintain many simultaneous I/O requests.
ATA	AT Attachment; A standardized command interface used on consumer-grade disk drives and SSDs.
FTL	Flash Translation Layer; A part of an SSD that manages data storage, garbage collection, and wear leveling.
HPC	High-Performance Computing; A computer system built with massive parallelism to perform very large tasks.

Continued on next page

Table A.1 – continued from previous page

Acronym	Meaning
IOPS	I/O Operations Per Second; A measurement of a storage device's performance.
IPI	Inter-Processor Interrupt; On a multi-processor computer, an interrupt from one processor to another.
IRQ	Interrupt Request; A hardware signal used to request action by a microprocessor, usually on behalf of an I/O device.
MLC	Multi-Level Cell; A type of NAND Flash storing multiple bits as an analog value in a single gate.
MSI-X	Message Signaled Interrupts; A standard method for implementing interrupts on a PCI or PCI Express bus using in-band bus messages, without an interrupt pin.
NUMA	Non-Uniform Memory Architecture; A multi-processor computer design where banks of memory are local to one or more microprocessors, making latency depend on which microprocessor makes memory requests.
SCSI	Small Computer Systems Interface; A standardized command interface used on enterprise disk drives (and a few SSDs).
SLC	Single-Level Cell; A type of NAND Flash storing one bit per gate.
SMP	Symmetric Mult-Processing; A computer posessing multiple identical microprocessors managed by a single operating system.
SSD	Solid State Drive; A mass storage device using electronic circuits for data storage rather than magnetic media.
VFS	Virtual File System; A kernel layer providing identical user semantics regardless of the filesystem in use.