Internal Parallelism of Flash Memory-Based Solid-State Drives

FENG CHEN and BINBING HOU, Louisiana State University
RUBAO LEE, Ohio State University

A unique merit of a solid-state drive (SSD) is its *internal parallelism*. In this article, we present a set of comprehensive studies on understanding and exploiting internal parallelism of SSDs. Through extensive experiments and thorough analysis, we show that exploiting internal parallelism of SSDs can not only substantially improve input/output (I/O) performance but also may lead to some surprising side effects and dynamics. For example, we find that with parallel I/Os, SSD performance is no longer highly sensitive to access patterns (random or sequential), but rather to other factors, such as data access interferences and physical data layout. Many of our prior understandings about SSDs also need to be reconsidered. For example, we find that with parallel I/Os, write performance could outperform reads and is largely independent of access patterns, which is opposite to our long-existing common understanding about slow random writes on SSDs. We have also observed a strong interference between concurrent reads and writes as well as the impact of physical data layout to parallel I/O performance. Based on these findings, we present a set of case studies in database management systems, a typical data-intensive application. Our case studies show that exploiting internal parallelism is not only the key to enhancing application performance, and more importantly, it also fundamentally changes the equation for optimizing applications. This calls for a careful reconsideration of various aspects in application and system designs. Furthermore, we give a set of experimental studies on new-generation SSDs and the interaction between internal and external parallelism in an SSD-based Redundant Array of Independent Disks (RAID) storage. With these critical findings, we finally make a set of recommendations to system architects and application designers for effectively exploiting internal parallelism.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.2 [**Operating Systems**]: Performance—*Measurements*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Flash memory, solid state drive, internal parallelism, storage systems

ACM Reference Format:
Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. ACM Trans. Storage 12, 3, Article 13 (April 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2818376

1. INTRODUCTION

High-speed data processing demands high storage input/output (I/O) performance. Having dominated computer storage systems for decades, magnetic disks, due to their

An earlier version of this article was published in *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, Texas. This work was supported in part by National Science Foundation under grants CCF-1453705, CCF-0913150, OCI-1147522, and CNS-1162165, Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, as well as generous support from Intel Corporation.
Authors' addresses: F. Chen and B. Hou, Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803; email: {fchen, bhou}@csc.lsu.edu; R. Lee, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210; email: liru@cse.ohio-state.edu.

DOI: http://dx.doi.org/10.1145/2818376

mechanical nature, are becoming increasingly unsuitable for handling high-speed data processing jobs. In recent years, flash memory-based solid-state drives (SSDs) have received strong interest in both academia and industry [Chen et al. 2009; Kgil et al. 2008; Dirik and Jacob 2009; Agrawal et al. 2008; Birrell et al. 2005; Park et al. 2006]. As a type of semiconductor devices, SSDs are completely built on flash memory chips with no moving parts. Such an architectural difference enables us to address fundamentally the technical issues of rotating media. Researchers have invested extensive efforts to adopt SSD technology in the existing storage systems and proposed solutions for performance optimizations (e.g., Chen [2009], Tsirogiannis et al. [2009], Canim et al. [2009], Do and Patel [2009], Agrawal et al. [2009], Lee and Moon [2007], Lee et al. [2008], Lee et al. [2009b], Li et al. [2009], Nath and Gibbons [2008], Koltsidas and Viglas [2008], Shah et al. [2008], Graefe [2007], Lee et al. [2009b], Bouganim et al. [2009], Stoica et al. [2009], and Pritchett and Thottethodi [2010]). These prior research has focused mostly on leveraging the high random data access performance of SSDs and addressing the slow random write issues. However, an important factor of SSDs has not received sufficient attention and has been rarely discussed in the existing literature, which is *internal parallelism*, a unique and rich resource embedded inside SSDs. In this article, we show that the impact of internal parallelism is far beyond the scope of the basic operations of SSDs. We believe that fully exploiting internal parallelism resources can lead to a fundamental change in the current sequential-access-oriented optimization model, which has been adopted in system and application designs for decades. This is particularly important for data-intensive applications.

1.1. Internal Parallelism in SSDs

Internal parallelism is essentially a result of optimizations for addressing several architectural challenges in the design of SSD. SSDs have several inherent architectural limitations. (1) Due to current technical constraints, one single flash memory package can only provide limited bandwidth (e.g., 32 to 40MB/sec [Agrawal et al. 2008]), which is unsatisfactory for practical use as a high-performance storage. (2) Writes in flash memory are often much slower than reads. For example, Samsung K9LBG08U0M [Samsung 2007] requires 800μs for writing a page. (3) The Flash Translation Layer (FTL) running at the device firmware level needs to perform several critical operations, such as garbage collection and wear leveling [Chen et al. 2009; Agrawal et al. 2008; Dirik and Jacob 2009]. These internal operations can incur a high overhead (e.g., milliseconds).

In order to address these limitations, SSD architects have built an ingenious structure that enables rich internal parallelism: An array of flash memory packages integrated in an SSD and connected through multiple (e.g., 2–10) channels to flash memory controllers. A sequence of logical block addresses (LBA) is provided by the SSD as a logical interface to the host. Since logical blocks can be striped over multiple flash memory packages, data accesses can be performed independently in parallel. Such a highly parallelized design yields two benefits: (1) Transferring data from/to multiple flash memory packages in parallel can provide high bandwidth in aggregate. (2) High-latency operations, such as *erase*, can be effectively hidden behind other concurrent operations, because when a flash chip is performing a high-latency operation, the other chips can still service incoming requests simultaneously. These design optimizations effectively address the above-mentioned issues. Therefore, internal parallelism, in essence, is not only an *inherent functionality* but also a *basic requirement* for SSDs to deliver high performance.

Exploiting I/O parallelism has been studied in conventional disk-based storage, such as RAID [Patterson et al. 1988], a storage based on multiple hard disks. However, there are two fundamental differences between the SSD and RAID internal structures.

(1) *Different logical / physical mapping mechanisms*: For an RAID storage, logical blocks are *statically* mapped to "fixed" physical locations, which is determined by its logical block number (LBN) [Patterson et al. 1988]. For an SSD, a logical block is *dynamically* mapped to physical flash memory, and this mapping changes during runtime. A particular attention should be paid to a unique data layout problem in SSDs. In a later section, we will show that an ill-mapped data layout can cause undesirable performance degradation, while such a problem would not happen in RAID with its static mapping. (2) *Different physical natures*: RAID is built on magnetic disks, whose random access performance is often one order of magnitude lower than sequential access. In contrast, SSDs are built on flash memories, in which such a performance gap is much smaller, and thus are less sensitive to access patterns. This physical difference has a strong system implication to the existing sequentiality-based application designs: Without any moving parts, parallelizing I/O operations on SSDs could make random accesses capable of performing comparably to or even better than sequential accesses, while this is unlikely to happen in RAID. Considering all these factors, we believe a thorough study on internal parallelism of SSDs is highly desirable and will help us understand its performance impact, both the benefits and side effects, as well as the associated implications to system designers and data-intensive application users. Interestingly, we also note that using multiple SSDs to build a RAID system can form another layer of parallelism. In this article, we will also perform studies on the interaction between the internal parallelism and the RAID-level external parallelism.

1.2. Research and Technical Challenges

Internal parallelism makes a single SSD capable of handling multiple incoming I/O requests in parallel and achieving a high bandwidth. However, internal parallelism cannot be effectively exploited unless we address several critical challenges.

(1) *Performance gains from internal parallelism are highly dependent on how the SSD internal structure is insightfully understood and effectively utilized.* Internal parallelism is an architecture-dependent resource. For example, the mapping policy directly determines the physical data layout, which significantly affects the efficiency of parallelizing data accesses. In our experiments, we find that an ill-mapped data layout can cause up to 4.2 times higher latency for parallel accesses on an SSD. Without knowing the SSD internals, it is difficult to achieve the anticipated performance gains. Meanwhile, uncovering the low-level architectural information without changing the strictly defined device/host interface, such as Small Computer System Interface (SCSI), is highly challenging. In this article we present a set of simple yet effective experimental techniques to achieve this goal.
(2) *Parallel data accesses can compete for critical hardware resources in SSDs, and such interference would cause unexpected performance degradation.* Increasing parallelism is a double-edged sword. On one hand, high concurrency would improve resource utilization and increase I/O throughput. On the other hand, sharing and competing for critical resources may cause undesirable interference and performance loss. For example, we find mixing reads and writes can cause a throughput decrease as high as 4.5 times for writes, which also makes the runtime performance unpredictable. Only by understanding both benefits and side effects of I/O parallelism can we effectively exploit its performance potential while avoiding its negative effects.
(3) *Exploiting internal parallelism in SSDs demands fundamental changes to the existing program design and the sequential-access-oriented optimization model adopted in software systems.* Applications, such as database systems, normally assume that the underlying storage devices are disk drives, which perform well with sequential

I/Os but lack support for parallel I/Os. As such, the top priority for application design is often to *sequentialize* rather than *parallelize* data accesses for improving storage performance (e.g., Jiang et al. [2005]). Moreover, many optimization decisions embedded in application designs are implicitly based on such an assumption, which would unfortunately be problematic when being applied to SSDs. In our case studies on the PostgreSQL database system, we find that parallelizing a query with multiple subqueries can achieve a speedup of up to a factor of 5, and, more importantly, we also find that with I/O parallelism, the *query optimizer*, a key component in database systems, would make an *incorrect* decision on selecting the optimal query plan, which should receive increased attention from application and system designers.

1.3. Our Contributions

In this article, we strive to address the above-mentioned three critical challenges. Our contributions in this work are fivefold. (1) To understand the parallel structure of SSDs, we first present a set of experimental techniques to uncover the key architectural features of SSDs without any change to the interface. (2) Knowing the internal architectures of SSDs, we conduct a set of comprehensive performance studies to show both benefits and side effects of increasing parallelism on SSDs. Our new findings reveal an important opportunity to significantly increase I/O throughput and also provide a scientific basis for performance optimization. (3) Based on our experimental studies and analysis, we present a set of case studies in database systems, a typical data-intensive application, to show the impact of leveraging internal parallelism for real-world applications. Our case studies indicate that the existing optimizations designed for HDDs can be suboptimal for SSDs. (4) To study the interactions between internal and external parallelism, we further conduct experiments on an SSD-based RAID storage to understand the interference between the two levels. Our results show that only by carefully considering both levels can all parallelism resources be fully exploited. (5) Seeing the rapid evolution of flash technologies, we finally perform a set of experiments to revisit our studies on a comprehensive set of new-generation SSDs on the market. Our results show that recent innovations, such as 3D Vertical NAND (V-NAND) and advanced FTL designs, create both opportunities and new challenges.

The rest of this article is organized as follows. We first discuss the critical issues in Section 2. Section 3 provides the background. Section 4 introduces our experimental system and methodology. Section 5 presents how to detect the SSD internals. Sections 6 and 7 present our experimental and case studies on SSDs. Section 8 discusses the interaction with external parallelism. Section 9 gives experimental studies on new-generation SSDs. Section 10 discusses the system implications of our findings. Related work is given in Section 11. The last section concludes this article.

2. CRITICAL ISSUES FOR INVESTIGATION

In order to fully understand internal parallelism of SSDs, we strive to answer several critical questions to gain insight into this unique resource of SSDs and also reveal some untold facts and unexpected dynamics in SSDs.

(1) Limited by the thin device/host interface, how can we effectively uncover the key architectural features of an SSD? In particular, how is the physical data layout determined in an SSD?
(2) The effectiveness of parallelizing data accesses depends on many factors, such as access patterns, available resources, and others. Can we quantify the benefit of I/O parallelism and its relationship to these factors?

(3) Reads and writes on SSDs may interfere with each other. Can we quantitatively illustrate such interactive effects between parallel data accesses? Would such interference impair the effectiveness of parallelism?
(4) Readahead improves read performance, but it is sensitive to read patterns [Chen et al. 2009]. How would I/O parallelism affect the readahead? Can the benefits from parallelism offset the negative impact? How should we trade off between increasing parallelism and retaining effective readahead?
(5) The physical data layout on an SSD could change on the fly. How does an ill-mapped data layout affect the effectiveness of I/O parallelism and the readahead mechanism?
(6) Applications can exploit internal parallelism to optimize the data access performance. How much performance benefit can we achieve in a large storage system for data-intensive applications? Can we see some side effects?
(7) Many optimizations in applications are specifically tailored to the properties of hard drives and may be ineffective for SSDs. Can we make a case that parallelism-based optimizations would be important for maximizing data access performance on SSDs and changing the existing application design?
(8) SSD-based RAID provides another level of parallelism across devices. What are the interactions between internal parallelism of SSDs with such external cross-device parallelism? How would the RAID configurations affect the effectiveness of leveraging internal parallelism?
(9) Flash memory and SSD technologies are continuously evolving. Can we effectively uncover the architectural details for new-generation SSDs, which have highly complex internal structures? To what degree can we expose such low-level information without hardware support?

In this article, we will answer these questions through extensive experiments. We hope our experimental analysis and case studies will influence the system and application designers to carefully rethink the current sequential-access-oriented optimization model and treat parallelism as a *top priority* on SSDs.

3. BACKGROUND OF SSD ARCHITECTURE

3.1. NAND Flash Memory

Today's SSDs are built on NAND flash memory. NAND flash can be classified into two main categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. An SLC flash memory cell stores only one bit. An MLC flash memory cell can store two bits or even more. For example, recently Samsung began manufacturing Triple Level Cell (TLC) flash memories, which store 3 bits in one cell. An NAND flash memory package is usually composed of multiple *dies* (chips). A typical die consists of two or more *planes*. Each plane contains thousands (e.g., 2,048) of *blocks* and registers. A block normally consists of dozens to hundreds of pages. Each page has a data part (e.g., 4–8KB) and an associated metadata area (e.g., 128 bytes), which is often used for storing Error Correcting Code (ECC) and other information.

Flash memory has three major operations, *read*, *write*, and *erase*. Reads and writes are normally performed in units of pages. Some support subpage operations. A read normally takes tens of microseconds (e.g., 25μs to 50μs). A write may take hundreds of microseconds (e.g., 250μs to 900μs). Pages in a block must be written sequentially, from the least significant address to the most significant address. Flash memory does not support *in-place overwrite*, meaning that once a page is programmed (written), it cannot be overwritten until the entire block is erased. An erase operation is slow (e.g., 1.5ms) and must be conducted in block granularity. Different flash memories may have various specifications.

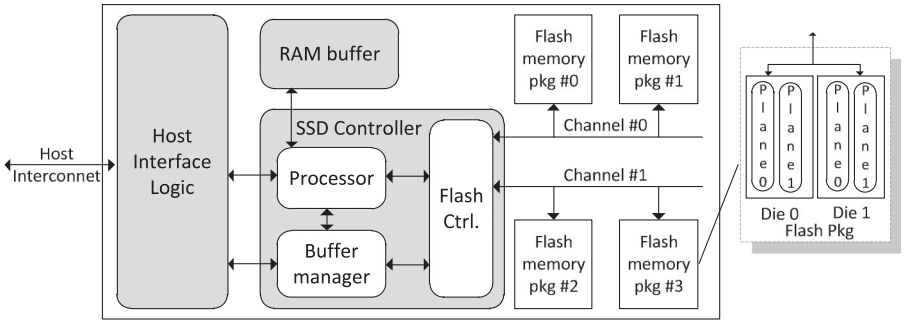


Fig. 1. An illustration of SSD architecture [Agrawal et al. 2008].

3.2. Flash Translation Layer (FTL)

In SSD firmware, FTL is implemented to emulate a hard disk. FTL exposes an array of logical blocks to the host and implements a set of complex mechanisms to manage flash memory blocks and optimize SSD performance. A previous work [Gal and Toledo 2005] provides a detailed survey on the FTL algorithms, and here we briefly introduce three key roles of FTL as follows: (1) *logical block mapping*, which maps logical block addresses (LBA) to *physical block addresses* (PBA); (2) *garbage collection*, which handles the no-in-place-write issue, and asynchronously recycles dirty blocks in a way similar to Log-Structured File System [Rosenblum and Ousterhout 1992]; and (3) *wear leveling*, which shuffles cold blocks (read intensive) with hot blocks (write intensive) to even out writes over flash memory blocks.

3.3. SSD Architecture and Internal Parallelism

A typical SSD includes four major components (see Figure 1). A *host interface logic* connects to the host via an interface connection (e.g., Serial ATA bus). An *SSD controller* manages flash memory, translates I/O requests, and issues I/O commands to flash memory via a *flash memory controller*. A dedicated Dynamic Random-Access Memory (DRAM) *buffer* holds metadata or data. Some SSDs use a small integrated Static Random-Access Memory (SRAM) buffer to lower production cost. In most SSDs, multiple (e.g., 2–10) channels connect the controller with flash memory packages. Each channel may be shared by multiple packages. Actual implementations may vary across different models. Prior research work [Agrawal et al. 2008; Dirik and Jacob 2009] gives detailed descriptions about the architecture of SSDs.

With such an architecture design, we can find that parallelism is available at different levels. Operations at each level can be parallelized or interleaved, which provides rich opportunities of internal parallelism.

(1) *Channel-Level Parallelism*: Since the data bus is the main performance bottleneck, SSDs usually incorporate multiple channels to connect the controller to the flash memory packages. Each channel can operate independently and simultaneously. Resource redundancy is provided. For example, some SSDs adopt an independent ECC engine for each channel to further speedup I/Os [Park et al. 2006], which improves performance but also increases production cost.
(2) *Package-Level Parallelism*: In order to optimize resource utilization, a channel is usually shared among multiple flash memory packages. Since each flash memory package can be operated independently, operations on flash memory packages attached to the same channel can be interleaved. As so, the bus utilization can be optimized [Dirik and Jacob 2009; Agrawal et al. 2008].

(3) *Die-Level Parallelism*: A flash memory package often includes two or more dies (chips). For example, a Samsung K9LBG08U0M flash memory package is composed of two K9GAG08U0M dies. Each die can be selected individually and execute a command independent of the others, which increases the throughput [Samsung 2007].
(4) *Plane-Level Parallelism*: A flash memory chip is typically composed of two or more planes. Flash memories (e.g., Samsung [2007] and MIC [2007]) support performing the same operation (e.g., read/write/erase) on multiple planes simultaneously. Some flash memories (e.g., MIC [2007]) provide `cache mode` to further parallelize medium access and bus transfer.

Such a highly parallelized architecture provides rich parallelism opportunities. In this article we will present several simple yet effective experimental techniques to uncover the internal parallelism at various levels.

3.4. Command Queuing

Native command queuing (NCQ) is a feature introduced by the SATA II standard [SAT 2011]. With NCQ support, the device can accept multiple incoming commands from the host and schedule the jobs internally. For a hard drive, by accepting multiple incoming commands from the host, it can better schedule jobs internally to minimize disk head movement. However, only one request can be executed each time due to its mechanical nature. For SSDs, NCQ is especially important, because the highly parallelized internal structure of an SSD can be effectively utilized only when the SSD is able to accept multiple concurrent I/O jobs from the host (operating system). NCQ enables SSDs to achieve real parallel data accesses internally. Compared to hard drives, NCQ on SSDs not only allows the host to issue multiple commands to the SSD for better scheduling but, more importantly, it also enables multiple data flows to concurrently exist inside SSDs. Early generations of SSDs do not support NCQ and thus cannot benefit from parallel I/Os [Bouganim et al. 2009].

The SATA-based SSDs used in our experiments can accept up to 32 jobs. New interface logic, such as NVMe [NVM Express 2015; Ellefson 2013], can further enable a much deeper queue depth (65,536) than the 32 parallel requests supported by SATA. Based on the PCI-E interface, NVMe technology can provide an extremely high bandwidth to fully exploit internal parallelism in large-capacity SSDs. Although in this article we mostly focus on the SATA-based SSDs, the technical trend is worth further study.

4. MEASUREMENT ENVIRONMENT

4.1. Experimental Systems

Our experimental studies have been conducted on a Dell Precision T3400. It is equipped with an Intel Core 2Duo E7300 2.66GHz processor and 4GB main memory. A 250GB Seagate 7200RPM hard drive is used for maintaining the OS and home directories (`/home`). We use Fedora Core 9 with the Linux Kernel 2.6.27 and Ext3 file system. The storage devices are connected through the on-board SATA connectors.

We select two representative, state-of-the-art SSDs fabricated by a well-known SSD manufacturer for our initial studies (see Table I). The two SSDs target different markets. One is built on multi-level cell (MLC) flash memories and designed for the mass market, and the other is a high-end product built on faster and more durable single-level cell (SLC) flash memories. For commercial reasons, we refer to the two SSDs as *SSD-M* and *SSD-S*, respectively. Delivering market-leading performance, both SSDs provide full support for NCQ and are widely used in commercial and consumer systems. Our experimental results and communication with SSD manufacturers also show that

Table I. SSD Specification

	SSD-M	SSD-S
Capacity	80GB	32GB
NCQ	32	32
Interface	SATA	SATA
Flash memory	MLC	SLC
Page Size (KB)	4	4
Block Size (KB)	512	256
Read Latency (μs)	50	25
Write Latency (μs)	900	250
Erase Latency (μs)	3,500	700

their designs are consistent with general-purpose SSD designs [Agrawal et al. 2008; Dirik and Jacob 2009] and are representative in the mainstream technical trend on the market. Besides the two SSDs, we also use a PCI-E SSD to study the SSD-based RAID in Section 8 and another set of five SSDs to study the new-generation SSDs with various flash chip and FTL technologies in Section 9. The hardware details about these SSDs will be presented in Sections 8 and 9.

In our experiments, we use the Completely Fair Queuing (CFQ) scheduler, the default I/O scheduler in the Linux kernel, for the hard drives. We use the No-optimization (*noop*) scheduler for the SSDs to leave performance optimization handled directly by the SSD firmware, which minimizes the interference from upper-level components and helps expose the internal behavior of the SSDs for our analysis. We also find *noop* outperforms the other I/O schedulers on the SSDs.

4.2. Experimental Tools and Workloads

For studying the internal parallelism of SSDs, we have used two tools in our experiments. We use the Intel Open Storage Toolkit [Mesnier 2011], which has also been used in prior work [Chen et al. 2009; Mesnier et al. 2007], to generate various types of I/O workloads with different configurations, including read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent I/O jobs), and others. The toolkit reports bandwidth, IOPS, and latency. The second one is our custom-built tool, called *replayer*. The replayer accepts a pre-recorded trace file and replays I/O requests to a storage device. This tool facilitates us to precisely repeat an I/O workload directly at the block device level. We use the two tools to generate workloads with the following three access patterns.

(1) *Sequential*: Sequential data accesses using a specified request size, starting from sector 0.
(2) *Random*: Random data accesses using a specified request size. Blocks are randomly selected from the first 1,024MB of the storage space, unless otherwise noted.
(3) *Stride*: Strided data accesses using a specified request size, starting from sector 0 with a stride distance between two consecutive data accesses.

Each workload runs for 30 seconds in default to limit trace size while collecting sufficient data. Unlike prior work [Polte et al. 2008], which was performed over an Ext3 file system, all workloads in our experiments directly access the SSDs as raw block devices. We do not create partitions or file systems on the SSDs. This facilitates our studies on the device behavior without being intervened by the OS components, such as page cache and file system (e.g., no file system level readahead). All requests are issued to the devices synchronously with no think time. Since writes to SSD may change the physical data layout dynamically, similarly to prior work [Bouganim et al. 2009; Chen et al. 2009], before each experiment we fill the storage space using sequential writes

with a request size of 256KB and pause for 5 seconds. This re-initializes the SSD status and keeps the physical data layout remaining largely constant across experiments.

4.3. Trace Collection

In order to analyze I/O traffic in detail, we use `blktrace` [Blktrace 2011] to trace the I/O activities at the block device level. The tool intercepts I/O events in the OS kernel, such as queuing, dispatching, and completion, and so on. Among these events, we are most interested in the *completion* event, which reports the latency for each individual request. The trace data are first collected in memory and then copied to the hard disk drive to minimize the interference caused by tracing. The collected data are processed using `blkparse` [Blktrace 2011] and our post-processing scripts off line.

5. UNCOVERING SSD INTERNALS

Before introducing our performance studies on SSD internal parallelism, we first present a set of experimental techniques to uncover the SSD internals. Two reasons have motivated us to detect SSD internal structures. First, internal parallelism is an architecture-dependent resource. Understanding the key architectural characteristics of an SSD is required to study and understand the observed device behavior. For example, our findings about the write-order based mapping (Section 5.4) has motivated us to further study the ill-mapped data layout issue, which has not been reported in prior literature. Second, the information detected can also be used for many other purposes. For example, knowing the number of channels in an SSD, we can set a proper concurrency level and avoid over-parallelization.

Obtaining such architectural information is particularly challenging for several reasons. (1) SSD manufacturers often regard the architectural design details as critical intellectual property and commercial secrets. To the best of our knowledge, certain information, such as the mapping policy, is not available in any datasheet or specification, although it is important for us to understand and exploit the SSD performance potential. (2) Although SSD manufacturers normally provide standard specification data (e.g., peak bandwidth), much important information is absent or obsolete. In fact, hardware/firmware changes across different product batches are common in practice, and, unfortunately, such changes are often not reflected in a timely manner in public documents. (3) SSDs on the market carefully follow a strictly defined host interface standard (e.g., SATA [SAT 2011]). Only limited information is allowed to pass through such a thin interface. As so, it is difficult, if not impossible, to directly get detailed internal information from the hardware. In this section, we present a set of experimental approaches to expose the SSD internals.

5.1. A Generalized Model

Despite various implementations, almost all SSDs strive to optimize performance essentially in a similar way—evenly distributing data accesses to maximize resource usage. Such a principle is implemented through the highly parallelized SSD architecture design and can be found at different levels. For the sake of generality, we define an abstract model to characterize such an organization based on open documents (e.g., Agrawal et al. [2008] and Dirik and Jacob [2009]): A *domain* is a set of flash memories that share a specific type of resources (e.g., channels). A domain can be further partitioned into *subdomains* (e.g., packages). A *chunk* is a unit of data that is continuously allocated within one domain. Chunks are interleavingly placed over a set of N domains, following a *mapping policy*. We call the chunks across each of N domains a *stripe*. One may notice that this model is, in principle, similar to RAID [Patterson et al. 1988]. In fact, SSD architects often adopt a RAID-0-like striping mechanism [Agrawal et al. 2008; Dirik and Jacob 2009]. Some SSD hardware even directly integrates a RAID

controller inside an SSD [PC Perspective 2009]. However, as compared to RAID, the key difference here is that SSDs map data dynamically, which determines the physical data layout on the fly and may cause an ill-mapped data layout, as we will see later.

In an SSD, hardware resources (e.g., pins, ECC engines) are shared at different levels. For example, multiple flash memory packages may share one channel, and two dies in a flash memory package share a set of pins. A general goal is to minimize resource sharing and maximize resource utilization, and, in this process, three *key factors* directly determine the internal parallelism.

(1) *Chunk size*: the size of the largest unit of data that is continuously mapped within an individual domain.
(2) *Interleaving degree*: the number of domains at the same level. The interleaving degree is essentially determined by the resource redundancy (e.g., channels).
(3) *Mapping policy*: the method that determines the domain to which a chunk of logical data is mapped. This policy determines the physical data layout.

We present a set of experimental approaches to *infer indirectly* the three key characteristics. Basically, we treat an SSD as a "black box" and we assume the mapping follows some repeatable but unknown patterns. By injecting I/O traffic with carefully designed patterns to the SSD, we observe the reactions of the device, measured with several key metrics, for example, latency and bandwidth. Based on this probing information, we can speculate the internal architectural features and policies adopted in the SSD. In essence, our solution shares a similar principle with characterizing RAID [Denehy et al. 2014], but characterizing SSDs needs to explore their unique features (e.g., dynamic mapping). We also note that, due to the complexity and diversity of SSD implementations, the retrieved information may not precisely uncover all the internal details. However, our purpose is not to reverse engineer the SSD hardware. We try to characterize the key architectural features of an SSD that are most related to internal parallelism, from the outside in a simple yet effective way. We have applied this technique to both SSDs and we found that it works pleasantly well for serving the purposes of performance analysis and optimization. For brevity, we only show the results for SSD-S. SSD-M behaves similarly. In Section 9, we will further study the effectiveness of our probing techniques on new-generation SSDs.

5.2. Chunk Size

As a basic mapping unit, a chunk can be mapped in only one domain, and two continuous chunks are mapped in two separate domains. Assuming the chunk size is S, for any read that is aligned to S with a request size no larger than S, only one domain would be involved. With an offset of $\frac{S}{2}$ from an aligned position, a read would split into two domains equally. Since many NAND flash memories support subpage access, the latter case would be faster due to the parallel I/Os. Based on this feature, we have designed an experiment to identify the chunk size.

We use read-only requests to probe the SSD. Before experiments, we first initialize the SSD data layout by sequentially overwriting the SSD with request size of 256KB to evenly map logical blocks across domains [Chen et al. 2009]. We estimate a maximum possible chunk size,[1] M (a power of 2) sectors, and we select a request size, n sectors, which must be no larger than M. For a given request size, we conduct a set of tests, as follows: We first set an offset, k sectors (512 bytes each), which is initialized to 0 at the beginning. Then we generate 100,000 reads to the SSD, each of which reads n sectors at a position which is k sectors (offset) from a random position that is aligned to

[1]In practice, we can start with a small M (e.g., 64 sectors) to reduce the number of tests.

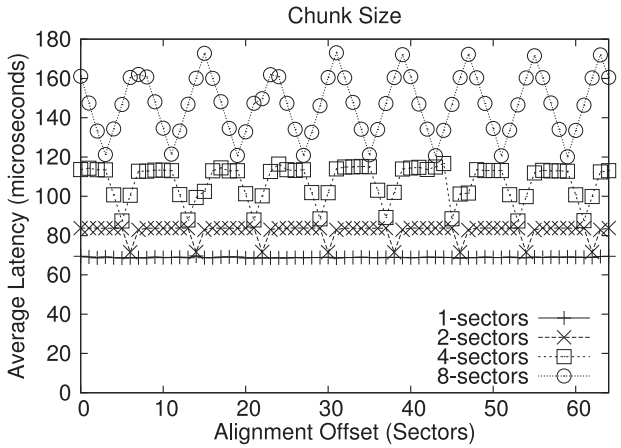


Fig. 2. Detecting the chunk size on SSD-S. Each curve corresponds to a specified request size.

PROGRAM 1: Pseudocode of uncovering SSD internals

```
init_SSD():             sequentially write SSD with 256KB requests
rand_pos(A):            get a random offset aligned to A sector
read(P, S):             read S sectors at offset P sect.
stride_read(J,D):       read 1 chunk with J jobs from offset 0, each read skips over D chunks
plot(X,Y,C):            plot a point at (X,Y) for curve C
M:                      an estimated maximum chunk size
D:                      an estimated maximum interleaving degree
(I) detecting chunk size:
  init_SSD();                                   /* initialize SSD space */
  for (n = 1 sector; n <= M; n *= 2):           /* request size */
    for (k = 0 sector; k <= 2*M; k ++):         /* offset */
      for (i = 0, latency=0; i < 100000; i ++):
        pos = rand_pos(M) + k;
        latency += read (pos, n);
      plot (k, latency/100000, n);              /* plot average latency */

(II) detecting interleaving degree:
  init_SSD();                                   /* initialize SSD space */
  for (j=2; j <=4; j*=2):                       /* number of jobs */
    for (d = 1 chunk; d < 4*D; d ++):           /* stride distance */
      bw = stride_read (j, d);
      plot (d, bw, j);                          /* plot bandwidth */
```

the estimated maximum chunk size M. By selecting a random position, we can reduce the possible interference caused by on-device cache. We calculate the average latency of the 100,000 reads with an offset of k sectors. Then we increment the offset k by one sector and repeat the same test for no less than $2M$ times and plot a curve of latencies with different offsets. Then we double the request size, n, and repeat the whole set of tests again until the request size reaches M. The pseudocode is shown in Program 1(I).

Figure 2 shows an example result on SSD-S. Each curve represents a request size. For brevity, we only show results for request sizes of 1–8 sectors with offsets increasing from 0 to 64 sectors. Except for the case of request size of one sector, a dip periodically appears on the curves as the offset increases. *The chunk size is the interval between the bottoms of two consecutive valleys.* In this case, the detected chunk size is 8 sectors (4KB), the flash page size, but note that a chunk can consist of multiple flash pages in some implementations [Dirik and Jacob 2009]. For a request size of 1 sector (512 bytes),

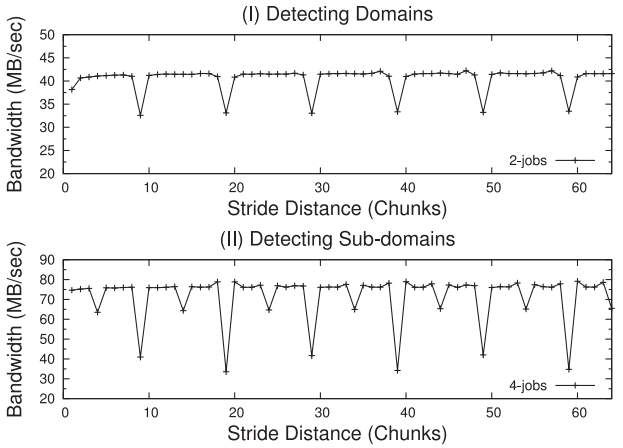


Fig. 3. Detecting the interleaving degree on SSD-S. Figures (I) and (II) use two jobs and four jobs, respectively.

we see a flat curve, because the smallest read/write unit in the OS kernel is one sector and cannot be mapped across two domains. Also note that since this approach relies on the hardware properties, some devices may show a distinct or weak pattern. In Section 9, we will show our studies on new-generation SSDs, and we find that the two SSDs adopting V-NAND technologies do exhibit very weak patterns.

5.3. Interleaving Degree

In our model, chunks are organized into domains based on resource redundancy. Interleaving degree (i.e., the number of domains) represents the degree of resource redundancy. Parallel accesses to multiple domains without resource sharing can achieve a higher bandwidth than doing that congested in one domain. Based on this feature, we have designed an experiment to determine the interleaving degree.

We use read-only requests with size of 4KB (the detected chunk size) to probe the SSD. Similarly to the previous test, we first initialize the SSD data layout and estimate a maximum possible interleaving degree, D. We conduct a set of tests as follows: We first set a stride distance of d chunks, which is initialized to 1 chunk (4KB) at the beginning. Then we use the toolkit to generate a read-only workload with *stride* access pattern to access the SSD with multiple (2–4) jobs. This workload sequentially reads a 4KB chunk each time, and each access skips over d chunks from the preceding access position. We run the test to measure the bandwidth. Then we increment the stride distance d by one chunk and repeat this test for no less than $2D$ times. Program 1(II) shows the pseudocode.

Figure 3(I) and (II) shows the experimental results with two jobs and four jobs on SSD-S, respectively. In Figure 3(I), we observe a periodically appearing dip. *The interleaving degree is the interval between the bottoms of two consecutive valleys, in units of chunks.* For SSD-S, we observe 10 domains, each of which corresponds to one channel.

The rationale behind this experiment is as follows. Suppose the number of domains is D and the data are interleavingly placed across the domains. When the stride distance, d, is $D \times n - 1$, where $n \geq 1$, every data access would exactly skip over $D - 1$ domains from the previous position and fall into the same domain. Since we issue two I/O jobs simultaneously, the parallel data accesses would compete for the same resource and the bandwidth is only around 33MB/sec. With other stride distances, the two parallel jobs

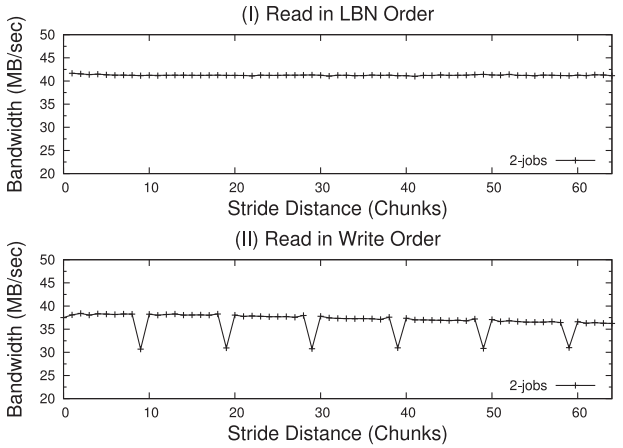


Fig. 4. Detecting the mapping policy on SSD-S. Figures (I) and (II) represent two test cases.

would be distributed across two domains and thus could achieve a higher bandwidth (around 40MB/sec).

By extending the queue depth to four jobs, we can further observe m subdomains in each of D domains. As shown in Figure 3(II), we see two deep dips (33MB/sec) appearing at stride distance of $D \times m \times n - 1$ chunks, where $n, m \geq 1$. At those points, all requests are directed to the same subdomain. A moderate dip (40MB/sec) appears in the middle of two deep dips, because the requests are directed to one domain but in different subdomains. The two shallow dips (64MB/sec) represent the case where two domains are accessed by the four jobs. In this case, we observed two subdomains, each of which corresponds to one package.

5.4. The Mapping Policy

Although all SSDs strive to evenly distribute data accesses across domains, the mapping policies may differ. The mapping policy determines the physical page to which a logical page is mapped. According to open documents (e.g., Dirik and Jacob [2009] and Agrawal et al. [2008]), two mapping policies are widely used in practice. (1) *A LBA-based static mapping*: For a given logical block address (*LBA*) with an interleaving degree, *D*, the block is mapped to a domain number ($LBA \bmod D$). (2) *A write-order-based dynamic mapping*: Differing from static mapping, dynamic mapping is determined by the order in which the blocks are written. That is, for the ith write, the block is assigned to a domain number ($i \bmod D$). We should note here that the write-order-based mapping is *not* the log-structured mapping policy [Agrawal et al. 2008], which appends data in a flash block to optimize write performance. Considering the wide adoption of the two mapping policies in SSD products, we will focus on these two policies in our experiments.

To determine which mapping policy is adopted, we first *randomly* overwrite the first 1024MB data with a request size of 4KB, the chunk size. Each chunk is guaranteed to be overwritten *once and only once*. After such a randomization, the blocks are relocated across the domains. If the LBA-based static mapping is adopted, then the mapping should not be affected by such random writes. Then, we repeat the experiment in Section 5.3 to see if the same pattern (repeatedly appearing dips) repeats. We find, after randomization, that the pattern (repeatedly appearing dips) disappears (see Figure 4(I)), which means that the random overwrites have changed the block mapping, and the LBA-based mapping is not used.

We then conduct another experiment to confirm that the write-order-based dynamic mapping is adopted. We use `blktrace` [Blktrace 2011] to record the order of random writes. We first randomly overwrite the first 1024MB space, and each chunk is written *once and only once*. Then we follow the same order in which blocks are written to issue reads to the SSD and repeat the same experiments in Section 5.3. For example, for the LBNs of random writes in the order of (91, 100, 23, 7, ...), we will read data in the same order (91, 100, 23, 7, ...). If the write-order based mapping is used, then the blocks should be interleavingly allocated across domains in the order of writes (e.g., blocks 91, 100, 23, 7 are mapped to domains 0, 1, 2, 3, respectively), and reading data in the same order would repeat the same pattern (dips) that we saw previously. Our experimental results have confirmed this hypothesis, as shown in Figure 4(II). The physical data layout and the block mapping are *strictly* determined by the order of writes. We have also conducted experiments by mixing reads and writes, and we find that the layout is *only* determined by writes.

5.5. Discussions

Levels of Parallelism: The internal parallelism of SSDs is essentially a result of redundant resources shared at different levels in the SSD. Such a resource redundancy enables the channel-, package-, die-, and plane-level parallelism. Our experimental studies have visualized such potential parallelism opportunities at different levels and also proved that they may impact performance to different extents. For example, Figure 3 shows that there exist at least two levels of parallelism in the SSD, which are corresponding to the channel and package levels. We also can see that leveraging the channel-level parallelism is the most effective (the top points). In contrast, leveraging the package-level parallelism (the middle points) can bring additional benefits but at a limited scale, and congesting all I/O requests in one package would cause the lowest performance. In Section 9, we will further study five new-generation SSDs, and one exhibits three levels of parallelism. In general, due to the interference and overhead in the system, the deeper the level of internal parallelism, the weaker the effect we can see. Nevertheless, we can largely conclude that regarding different types of internal parallelism, the top-level parallelism is the most important resource. We should parallelize our applications to maximize the use of such most shared resources in design optimizations.

Limitations of our approach: Our approach for uncovering the SSD internals has several limitations. First, as a black-box approach, our solution assumes certain general policies adopted in the device. As the hardware implementation becomes more and more complex, it is increasingly challenging to expose all the hardware details. For example, on-device cache in new-generation SSDs could be large (512MB) and interfere our results. In experiments, we attempt to avoid such a problem by flooding the cache with a large amount of data before the test and accessing data at random locations. Second, new flash memory technologies, such as TLC and 3D V-NAND technology, may change the property of storage medium itself, and some observed patterns could be differ. Also, sophisticated FTL designs, such as on-device compression and deduplication [Chen et al. 2011b], may further increase the difficulty. In Section 9, we further study a set of new-generation SSDs, which are equipped with complex FTL and new flash memory chips, to revisit the effectiveness of the hardware probing techniques presented in this section.

6. PERFORMANCE STUDIES

Prepared with the knowledge about the internal structures of SSDs, we are now in a position to investigate the performance impact of internal parallelism. We strive to answer several related questions on *the performance benefits of parallelism*, *the*

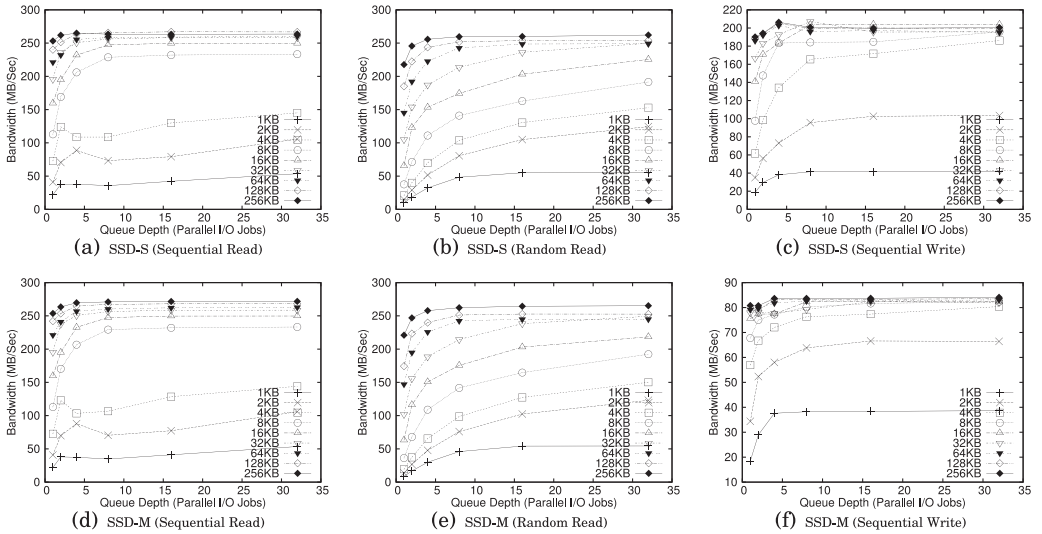


Fig. 5. Bandwidths of the SSDs with increasing queue depth (the number of concurrent I/O jobs).

interference between parallel jobs, *the readahead mechanism with I/O parallelism*, and *the performance impact of physical data layout*.

6.1. What Are the Benefits of Parallelism?

In order to quantitatively measure the potential benefits of I/O parallelism in SSDs, we run four workloads with different access patterns, namely *sequential read*, *sequential write*, *random read*, and *random write*. For each workload, we increase the request size from 1KB to 256KB,[2] and we show the bandwidths with a queue depth (i.e., the number of concurrent I/O jobs) increasing from 1 job to 32 jobs. In order to compare workloads with different request sizes, we use bandwidth (MB/s), instead of I/O per second (IOPS), as the performance metric. Figure 5 shows the experimental results for SSD-M and SSD-S. Since the write-order-based mapping is used, as we see previously, random and sequential writes on both SSDs show similar patterns. For brevity, we only show the results of sequential writes here.

Differing from prior work [Bouganim et al. 2009], our experiments show a great performance benefit from parallelism on SSDs. The significance of performance gains depends on several factors, and we present several key observations here.

(1) *Workload access patterns determine the performance gains from parallelism.* In particular, small and random reads yield the most significant performance gains. In Figure 5(b), for example, increasing queue depth from 1 to 32 jobs for random reads on SSD-S with a request size 4KB achieves a *7.2-fold* bandwidth increase. A large request (e.g., 256KB) benefits relatively less from parallelism, because the continuous logical blocks are often striped across domains and it already benefits from internal parallelism. This implies that in order to exploit effectively internal parallelism, we can either increase request sizes or parallelize small requests.
(2) *Highly parallelized small/random accesses can achieve performance comparable to large sequential accesses without parallelism.* For example, with only 1 job, the

[2]Increasing request size would weaken the difference between random and sequential workloads. However, in order to give a complete picture, we show the experimental results of changing the request size from 1KB to 256KB for all the workloads here.

bandwidth of random reads of 16KB on SSD-S is only 65.5MB/sec, which is 3.3 times lower than that of sequential reads of 64KB (221.3MB/sec). With 32 jobs, however, the same workload (16KB random reads) can reach a bandwidth of 225.5MB/s, which is comparable to the single-job sequential reads. This indicates that SSDs provide us an alternative approach for optimizing I/O performance, *parallelizing small and random accesses*, in addition to simply organizing sequential accesses. This unlocks many new opportunities. For example, database systems traditionally favor a large page size, since hard drives perform well with large requests. On SSDs, which are less sensitive to access patterns, if parallelism can be utilized, a small page size can be a sound choice for optimizing buffer pool usage [Lee et al. 2009b; Graefe 2007].

(3) *The performance potential of increasing I/O parallelism is physically limited by the redundancy of available resources.* We observe that when the queue depth reaches over 8–10 jobs, further increasing parallelism receives diminishing benefits. This observation echoes our finding made in Section 5 that the two SSDs have 10 domains, each of which corresponds to a channel. When the queue depth exceeds 10, a channel has to be shared by more than 1 job. Further parallelizing I/O jobs can bring additional but smaller benefits. On the other hand, we also note that such an over-parallelization does not result in undesirable negative effects.

(4) *The performance benefits of parallelizing I/Os also depend on flash memory mediums.* MLC and SLC flash memories provide noticeable performance difference, especially for writes. In particular, writes on SSD-M, the MLC-based lower-end SSD, quickly reach the peak bandwidth (only about 80MB/sec) with a small request size at a low queue depth. SSD-S, the SLC-based higher-end SSD, shows much higher peak bandwidth (around 200MB/sec) and more headroom for parallelizing small writes. In contrast, less difference can be observed for reads on the two SSDs, because the main performance bottleneck is transferring data across the serial I/O bus rather than reading the flash medium [Agrawal et al. 2008].

(5) *Write performance is insensitive to access patterns, and parallel writes can perform faster than reads.* The uncovered write-order-based mapping policy indicates that the SSDs actually handle incoming writes in the same way, regardless of write patterns (random or sequential). This leads to the observed similar patterns for sequential writes and random writes. Together with the parallelized structure and the on-device buffer, write performance is highly optimized and can even outperform reads in some cases. For example, writes of 4KB with 32 jobs on SSD-S can reach a bandwidth of 186.1MB/sec, which is even 28.3% higher than reads (145MB/sec). Although this surprising result contrasts with our common understanding about slow writes on SSDs, we need to note that for sustained intensive writes, the cost of internal garbage collection will dominate and lower write performance eventually, which we will show in Section 9.

On both SSD-S (Figure 5(a)) and SSD-M (Figure 5(d)), we can also observe a slight *dip* for sequential reads with small request sizes at a low concurrency level (e.g., four jobs with 4KB requests). This is related to interference to the readahead mechanism. We will give a detailed analysis on this in Section 6.3.

6.2. How Do Parallel Reads and Writes Interfere with Each Other and Cause Performance Degradation?

Reads and writes on SSDs can interfere with each other for many reasons. (1) Both operations share many critical resources, such as the ECC engines and the lock-protected mapping table, and so on. Parallel jobs accessing such resources need to be serialized. (2) Both writes and reads can generate background operations internally, such

Table II. Bandwidths (MB/sec) of Co-Running Reads and Writes

	Sequential Write	Random Write	None
Sequential Read	109.2	103.5	72.6
Random read	32.8	33.2	21.3
None	61.4	59.4	

as readahead and asynchronous writeback [Chen et al. 2009]. These internal operations could negatively impact foreground jobs. (3) Mingled reads and writes can foil certain internal optimizations. For example, flash memory chips often provide a `cache mode` [MIC 2007] to pipeline a sequence of reads or writes. A flash memory plane has two registers, *data register* and *cache register*. In the cache mode, when handling a sequence of reads or writes, data can be transferred between the cache register and the controller, while concurrently moving another page between the flash medium and the data register. However, such pipelined operations must be performed in one direction, so mingling reads and writes would interrupt the pipelining and cause performance loss.

In order to illustrate such an interference, we use the toolkit to generate a pair of concurrent workloads, similar to that in the previous section. The two workloads access data in two separate 1,024MB storage spaces. We choose four access patterns, namely *random reads*, *sequential reads*, *random writes*, and *sequential writes*, and enumerate the combinations of running two workloads simultaneously. Each workload uses request size of 4KB and one job. We report the aggregate bandwidths (MB/s) of two co-running workloads on SSD-S in Table II. Co-running with "none" means running a workload individually.

We have observed that for all combinations, the aggregate bandwidths cannot reach the optimal result, the sum of the bandwidths of individual workloads. We also find that *reads and writes have a strong interference with each other, and the significance of this interference highly depends on read access patterns*. If we co-run sequential reads and writes in parallel, the aggregate bandwidth exceeds that of each workload running individually, which means parallelizing workloads obtains benefits, although the aggregate bandwidths cannot reach the optimal (i.e., the sum of the bandwidths of all workloads). However, when running random reads and writes together, we see a strong negative impact. For example, sequential writes can achieve a bandwidth of 61.4MB/s when running individually; however, when running with random reads, the bandwidth drops by a factor of 4.5 to 13.4MB/sec. Meanwhile, the bandwidth of random reads also drops from 21.3MB/sec to 19.4MB/sec. Apparently the co-running reads and writes strongly interfere with each other.

A recent research [Wu and He 2014] by Wu and He has studied the conflict between program/erase operations and read operations in NAND flash and found that the lengthy P/E operations can significantly interfere with the read performance, which well explains our observation here. In their study, they also propose a low-overhead P/E suspension scheme to service pending reads with a priority, which shows significant performance benefits and points to the direction for eventually resolving this challenge.

6.3. How Does I/O Parallelism Impact the Effectiveness of Readahead?

Prior study [Chen et al. 2009] shows that with no parallelism, sequential reads on SSD can substantially outperform random reads, because modern SSDs implement a readahead mechanism at the device level to detect sequential data accesses and prefetch data into the on-device buffer. However, parallelizing multiple sequential read streams could result in a sequence of mingled reads, which can affect the detection of sequential patterns and invalidate readahead. In the meantime, parallelizing reads

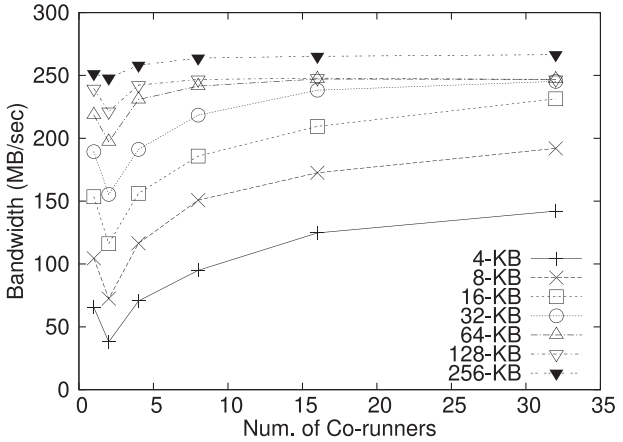


Fig. 6. Performance impact of parallelism on readahead. Each curve represents a request size.

can improve bandwidths, as we see before. So there is a tradeoff between *increasing parallelism* and *retaining effective readahead*.

In order to examine the impact of parallelism on readahead, we have designed a workload to generate multiple jobs, each of which sequentially reads an individual 1024MB space (i.e., the *i*th job reads the *i*th 1,024MB space) simultaneously. Before the experiments, we sequentially overwrite the storage space. We increase the concurrency from 1 to 32 jobs and vary the size from 4KB to 256KB. We compare the aggregate bandwidths of the co-running jobs on SSD-S. SSD-M shows similar results.

Figure 6 shows that in nearly all curves, there exists a *dip* when the queue depth is two jobs. For example, the bandwidth of 4KB reads drops by *43%* from one job to two jobs. At that point, readahead is strongly inhibited due to mingled reads, and such a negative impact cannot be offset at a low concurrency level (two jobs). When we further increase the concurrency level, the benefits coming from parallelism quickly compensate for impaired readahead. It is worth noting that we have verified that this effect is not due to accidental channel conflicts. We added a 4KB shift to the starting offset for each parallel stream, which makes requests mapped to different channels, and we observed the same effect. This case indicates that *readahead can be impaired by parallel sequential reads, especially at low concurrency levels and with small request sizes.* SSD architects can consider including a more sophisticated sequential pattern detection mechanism to identify multiple sequential read streams to avoid this problem.

Another potential hardware limit is the on-device buffer size. For optimizations like readahead, their effectiveness is also limited by the buffer capacity. Aggressively preloading data into the buffer, although it could remove future access cost, may also bring a side effect: buffer pollution. Therefore, a large buffer size in general is beneficial for avoiding such a side effect and accommodating more parallel requests for processing. As flash devices are moving toward supporting very high I/O parallelism, the buffer size must be correspondingly increase. On the other hand, this could lead to another problem: how to keep data safe in such volatile memory. New technologies, such as non-volatile memory (NVM), could be valid solutions [Chen et al. 2014].

6.4. How Does an Ill-Mapped Data Layout Impact I/O Parallelism?

The dynamic write-order-based mapping enables two advantages for optimizing SSD designs. First, high-latency writes can be evenly distributed across domains, which not only guarantees load balance but also naturally balances available free flash blocks and

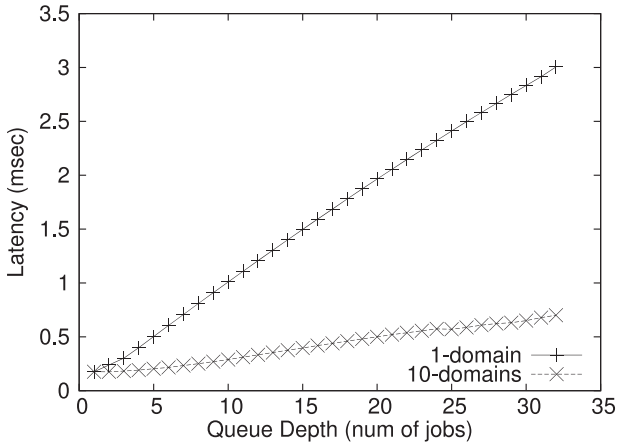


Fig. 7. Performance impact of data layout.

helps even out wears across domains. Second, writes across domains can be overlapped with each other, and the high latency of writes can be effectively hidden behind parallel operations.

On the other hand, such a dynamic write-order-based mapping may result in negative effects in certain conditions. An outstanding one is that since the data layout is completely determined by the order of incoming writes on the fly, logical blocks could be mapped to only a subset of domains and result in an *ill-mapped data layout*. In the worst case, if a set of blocks is mapped to the same domain, then data accesses would be congested and have to compete for the shared resources, which would impair the effectiveness of parallel data accesses.

As discussed before, the effectiveness of increasing I/O parallelism is highly dependent on the physical data layout. Only when the data blocks are physically located in different domains is high I/O concurrency meaningful. An ill-mapped data layout can significantly impair the effectiveness of increasing I/O parallelism. To quantitatively show the impact of an ill-mapped physical data layout, we design an experiment on SSD-S as follows. We first sequentially overwrite the first 1,024MB of storage space to map the logical blocks evenly across domains. Since sequential reads benefit from the readahead, which would lead to unfair comparison, we create a random read trace with request size of 4KB to access the 10 domains in a round-robin manner. Similarly, we create another trace of random reads to only one domain. Then we use the *replayer* to replay the two workloads and measure the average latencies for each. We vary the queue depth from 1 job to 32 jobs and compare the average latencies of parallel accesses to data that are concentrated in one domain and that are distributed across 10 domains. Figure 7 shows that when the queue depth is low, accessing data in one domain is comparable to doing it in 10 domains. However, as the I/O concurrency increases, the performance gap quickly widens. In the worst case, with a queue depth of 32 jobs, accessing data in the same domain (3ms) incurs *4.2 times* higher latency than doing that in 10 domains (0.7ms). This case shows that *an ill-mapped data layout can significantly impair the effectiveness of I/O parallelism*.

6.5. Readahead on Ill-Mapped Data Layout

An ill-mapped data layout may impact readahead, too. To show this impact, we map the blocks in the first 1,024MB storage space into 1, 2, 5, and 10 domains. Manipulating data layout is simple: We sequentially write the first 1,024MB data with request size

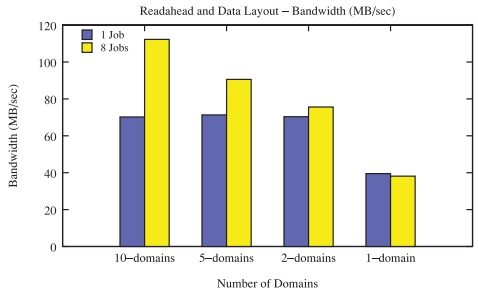


Fig. 8. The impact of ill-mapped data layout for readahead with queue depths of one and eight jobs.

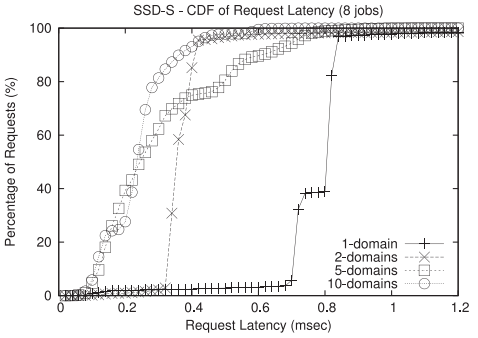


Fig. 9. CDF of latencies for sequential reads with 8 jobs to data mapped in 1–10 domains.

of 4KB, but between each two writes, we insert different numbers of random writes of 4KB data. For example, if we attempt to map all blocks into 1 domain, we insert 9 random writes of 4KB data. Then we use the toolkit to generate sequential reads with request size of 4KB and queue depth of one job andeight jobs on different physical layouts.

Figure 8 shows that as the logical blocks are increasingly concentrated into a subset of domains, the bandwidth decreases significantly (up to 2.9×). It is apparent that the ill-mapped data layout weakens the effectiveness of the readahead mechanism, since asynchronous reads are congested in one domain and data cannot be preloaded into the cache quickly enough to service the next read. We also can see that sequential reads with a high concurrency level (eight jobs in Figure 8) is more sensitive to the ill-mapped physical data layout. We collected the trace for sequential reads with eight jobs and compare the cumulative distribution function (CDF) of request latencies on different data layouts in Figure 9. In the worst case (one domain), a read incurs a latency of over 700μs, which is much worse than accessing data allocated in 10 domains.

6.6. Summary

In this section, we use microbenchmarks to show that parallelism can significantly improve performance, especially for random and small data accesses. In the meantime, our experimental results also show many dynamics and surprising results. We find that with parallelism, random reads can outperform sequential reads, and reads and writes have a strong interferences with each other and cause not only significant performance loss but also unpredictable performance. It is also evident in our study that the ill-mapped data layout can impair the effectiveness of I/O parallelism and readahead mechanism. Our results show that I/O parallelism can indeed provide significant performance improvement, but, in the meantime, we should also pay specific attention to the surprising results that come from parallelism, which provides us with a new understanding of SSDs.

7. CASE STUDIES IN DATABASE SYSTEMS

In this section, we present a set of case studies in database systems as typical data-intensive applications to show the opportunities, challenges, and research issues brought by I/O parallelism on SSDs. Our purpose is not to present a set of well-designed solutions to address specific problems. Rather, we hope that through these case studies, we can show that exploiting internal parallelism of SSDs can not only yield significant

performance improvement in large data-processing systems, but more importantly, it also introduces many emerging challenges.

7.1. Data Accesses in a Database System

Storage performance is crucial to query executions in database management systems (DBMS). A key operation of query execution is *join* between two *relations* (tables). Various *operators* (execution algorithms) of a join can result in completely different data access patterns. For warehouse-style queries, the focus of our case studies, two important join operators are *hash join* and *index join* [Graefe 1993]. Hash join sequentially fetches each *tuple* (a line of record) from the driving input relation and probes an in-memory hash table. Index join, in contrast, fetches each tuple from the driving input relation and starts index lookups on B^+ trees of a large relation. Generally speaking, hash join is dominated by sequential data accesses on a huge fact table, while index join is dominated by random accesses during index lookups.

Our case studies are performed on the PostgreSQL 8.3.4, which is a widely used open-source DBMS in Linux. The working directory and the database are located on SSD-S. We select Star Schema Benchmark (SSB) queries [O'Neil et al. 2009] as workloads (scale factor: 5). SSB workloads are considered to be more representative in simulating real warehouse workloads than TPC-H workloads [Stonebraker et al. 2007], and they have been used in recent research work (e.g., Abadi et al. [2008] and Lee et al. [2009a]). The SSB has one huge fact table (LINEORDER, 3340MB) and four small dimension tables (DATE, SUPPLIER, CUSTOMER, PART, up to 23MB). The query structures of all SSB queries are the same, that is, aggregations on equal joins among the fact table and one or more dimension tables. The join part in each query dominates the query execution. When executing SSB queries in PostgreSQL, hash join plans are dominated by sequential scans on LINEORDER, while index join plans are dominated by random index searches on LINEORDER.

7.2. Case 1: Parallelizing Query Execution

We first study the effectiveness of parallelizing query executions on SSDs. Our query-parallelizing approach is similar to the prior work [Tsirogiannis et al. 2009]. A star schema query with aggregations on multiway joins on the fact table and one or more dimension tables naturally has intraquery parallelism. Via data partitioning, a query can be segmented to multiple subqueries, each of which contains joins on partitioned data sets and preaggregation. The subqueries can be executed in parallel. The final result is obtained by applying a final aggregation over the results of subqueries.

We study two categories of SSB query executions. One is using the index join operator and dominated by random accesses, and the other is using the hash join operator and dominated by sequential accesses. For index join, we partition the dimension table for the first-level join in the query plan tree. For hash join, we partition the fact table. For index join, we select query Q1.1, Q1.2, Q2.2, Q3.2, and Q4.3. For hash join, besides Q1.1, Q1.2, Q2.2, Q4.1, and Q4.2, we have also examined a simple query (Q0), which only scans LINEORDER table with no join operation. We use Q0 as an example of workloads with little computation. Figure 10 shows the speedup (execution time normalized to the baseline case) of parallelizing the SSB queries with subqueries.

We have the following observations. (1) For index join, which features intensive random data accesses, parallelizing query executions can speed up an index join plan by up to *a factor of 5*. We have also executed the same set of queries on a hard disk drive and observed no performance improvement, which means the factor-of-5 speedup is not due to computational parallelism. This case again illustrates the importance of parallelizing random accesses (e.g., B^+-tree lookups) on an SSD. When executing an index lookup dominated query, since each access is small and random, the DBMS

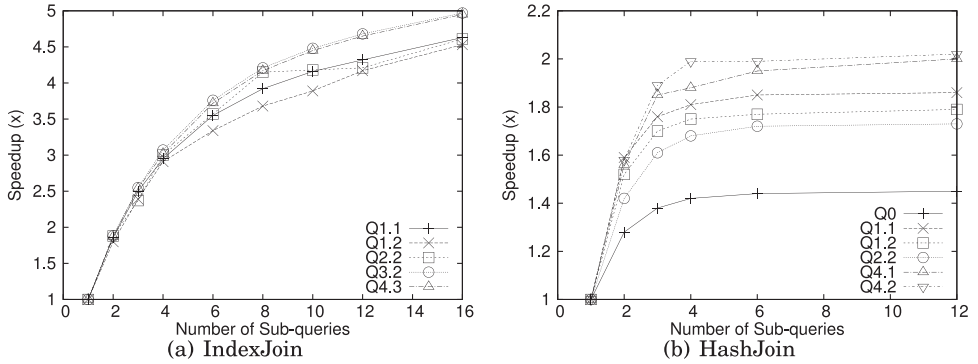


Fig. 10. Execution speedups of parallelizing SSB queries with index and hash join plans.

engine cannot fully utilize the SSD bandwidth. Splitting a query into multiple subqueries effectively reduces the query execution time. (2) For hash join, which features sequential data accesses, parallelizing query executions can speed up a hash join plan by up to *a factor of 2*. This is mainly because hash join is dominated by sequential accesses, which already benefit from internal parallelism to some extent. However, it still cannot fully exploit the SSD bandwidth, and parallelizing queries can achieve a lower but decent performance improvement. (3) Parallelizing query execution with no computation receives relatively less benefits. For example, parallelizing Q0 provides a speedup of *a factor of 1.4*, which is lower than other queries. Since Q0 simply scans a big table with no computation, the workload is very I/O intensive. In such a case, the SSD is fully saturated, which leaves little room for overlapping I/O and computation and limits further speedup through parallelism. In summary, this case clearly shows that in a typical data-processing application like database, I/O parallelism can truly provide substantial performance improvement on SSDs, especially for operations like index-tree lookups.

7.3. Case 2: Database-Informed Prefetching

The first case shows a big advantage of parallelizing a query execution over conventional query execution on SSDs. However, to achieve this goal, the DBMS needs to be extended to support query splitting and merging, which involves extra overhead and effort. Here we present a simpler alternative, *Database-Informed Prefetching*, to achieve the same goal for sequential-access queries, such as hash join.

We designed a user-mode daemon thread (only around 300 lines of C code) to asynchronously prefetch file data on demand. We added a hook function in the PostgreSQL server with only 27 lines of code to send prefetching commands to the daemon thread through the UNIX domain socket. When a query sequentially scans a huge file in hash join, the daemon is notified by the PostgreSQL server to initiate an asynchronous prefetching to load data from user-specified positions (e.g., each 32MB, in our settings). The daemon thread generates 256KB read requests to fully exploit the SSD bandwidth. Prefetched data are loaded into the OS page cache, rather than the database buffer pool, to minimize changes to the DBMS server code.

Figure 11 shows that, with only minor changes to the DBMS, our solution can keep the SSD as busy as possible and reduce query execution times by up to 40% (Q1.1). As expected, since the four queries used in our experiments are all dominated by large sequential reads, the performance improvement is relatively less significant than index joins but still substantial. An important merit of such a database-informed prefetching over the default OS-level prefetching is that with detailed application-level knowledge,

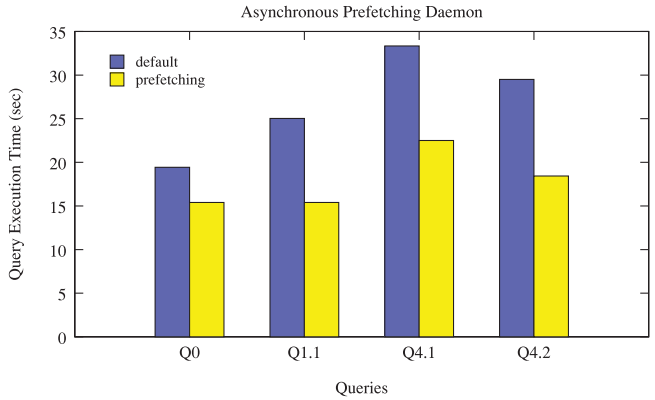


Fig. 11. Asynchronous prefetching in DBMS. *Default* and *prefetching* refer to the cases without and with prefetching, respectively.

the DBMS knows the exact data access patterns before executing a query, and thus aggressive prefetching can be safely applied without worrying about a low hit ratio that may be caused by mis-prefetching. In contrast, the default OS-level prefetching has to use a conservative prefetching, which cannot fully exploit the high bandwidth of SSDs. This case clearly indicates that even with only minor changes to DBMS, we can effectively speedup certain operations through parallelism.

7.4. Case 3: Interference between Parallel Queries

Although parallelizing I/O operations can improve system bandwidth in aggregate, concurrent workloads may interfere with each other. In this case, we give a further study on such an interference in macrobenchmarks by using four workloads as follows.

(1) *Sequential Read*: the SSB query Q1.2 executed using hash join, denoted as QS.
(2) *Random Read*: the SSB query Q1.2 executed using index join, denoted as QR.
(3) *Sequential Write*: *pgbench* [Pgbench 2011], a tool of PostgreSQL for testing OLTP performance. It is configured with 2 clients and generates frequent synchronous log flushing with intensive overwrites and small sequential writes.
(4) *Random Write*: *Postmarkm* [Shah and Noble 2007], a widely used file system benchmark. It emulates an email server, and we configure it with 1,000 files, 10 directories, and 20,000 transactions. It is dominated by synchronous random writes.

We run each combination of a pair of the workloads on SSD, except *pgbench*, which is measured using the reported transaction/second, the other three workloads are measured directly using execution times. When evaluating performance for running a target workload with another workload, we repeatedly execute the co-runner until the target workload completes. We report the performance degradations (%), compared to the baseline of running them alone, as shown in Figure 12.

As expected, Figure 12 shows a strong interference between I/O intensive workloads. Similarly to Section 6.2, we find that running two random database workloads (QR/QR) has the least interference (only 6%). However, we also have observed that the sequential read workload (QS) has a strong performance impact to its co-runners, both reads and writes, by up to 181% (QR/QS), which differs from what we have seen in microbenchmarks. This is because sequential reads in practical systems often benefit from the OS-level prefetching and form large requests. When running with other workloads, it would consume the most SSD bandwidth and interfere its co-runners. Sequential reads can even obtain some performance improvement (20% to 25%). This

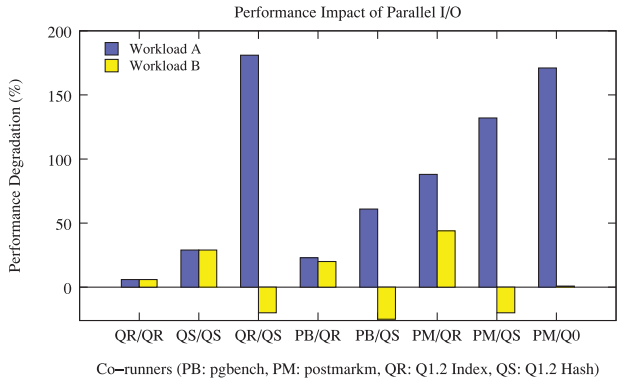


Fig. 12. Interference between workloads. PB, PM, QR, QS refer to pgbench, postmarkm, q1.2-index, and q1.2-hash, respectively.

result is repeatable across many runs and CPU is confirmed not the bottleneck. After examining the I/O trace, we found that without a co-runner, the query Q1.2 scans a large table file in a sequential pattern, which results in a low (14) queue depth with a large request size (512 sectors). With an injection of writes or random reads, the large requests are broken down to several smaller reads (256 sectors), which doubles the queue depth. As a result, reads are further parallelized, which aggressively overlaps computation and I/Os and results in better performance. To confirm this, we run Q0, which only sequentially scans the LINEORDER table without any computation, together with postmarkm (PM/Q0), and we find that without any computation, Q0 cannot benefit from the increased queue depth and becomes 0.9% slower. This case shows that the interference between co-running workloads in practice may lead to complicated results, and we should pay specific attention to minimize such interference.

7.5. Case 4: Revisiting Query Optimizer

A critical component in DBMS is the *query optimizer*, which decides the plan and operators used for executing a query. Traditionally, the query optimizer implicitly assumes the underlying storage device is a hard drive. Based on such an assumption, the optimizer estimates the execution times of various job plans and selects the lowest-cost query plan for execution. On a hard drive, a hash join enjoys an efficient sequential data access but needs to scan the whole table file; an index join suffers random index lookups but only needs to read a table partially. On an SSD, the situation becomes even more complicated, since parallelizing I/Os would weaken the performance difference of various access patterns, as we see previously in Section 6.1.

In this case, we study how the SSD internal parallelism will bring new consideration for the query optimizer. We select a standard Q2.2 and a variation of Q1.2 with a new predicate ("d_weeknuminyear=1") in the DATE table (denoted by $Q1.2_n$). We execute them first with no parallelism and then in a parallel way with subqueries. Figure 13 shows the query execution times of the two plans with 1 to 16 subqueries. One subquery means no parallelism.

In the figure, we find that SSD parallelism can greatly change the *relative strengths* of the two candidate plans. Without parallelism, the hash join plan is more efficient than the index join plan for both queries. For example, the index join for Q2.2 is 1.9 times slower than the hash join. The optimizer should choose the hash join plan. However, with parallelized subqueries, the index join outperforms the hash join for both queries. For example, the index join for Q2.2 is 1.4 times faster than the hash join.

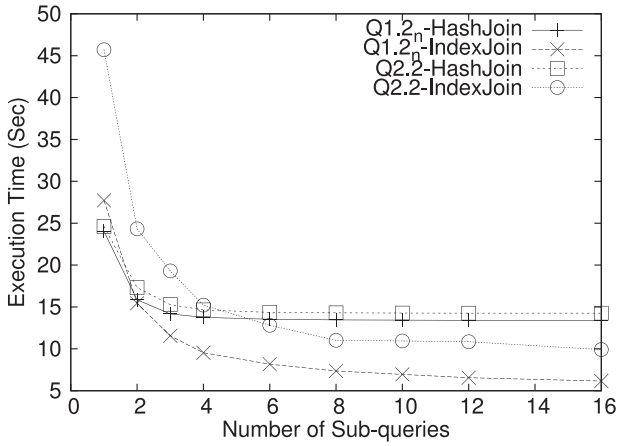


Fig. 13. Hash Join vs. Index Join with parallelism.

This implies that the query optimizer *cannot* make an optimal decision if it does not take parallelism into account when estimating the execution costs of candidate plans on SSDs. This case strongly indicates that when switching to an SSD-based storage, applications designed and optimized for magnetic disks must be carefully reconsidered, otherwise, the achieved performance can be suboptimal.

8. INTERACTION WITH RAID

Internal parallelism of SSD provides great performance potential for optimizing applications. RAID, which organizes an array of storage devices as a single logical device, can provide another level of parallelism to further parallelize I/Os, which we call *external parallelism*. The interaction between the two levels of parallelism is important. For example, if I/O requests are congested in one device, although internal parallelism can be well utilized, cross-device parallelism would be left unexploited. In this section, we study the interaction between the two layers by examining the effect of RAID data layout to the overall performance.

For our experiments, we use a high-end PCI-E SSD manufactured by the same vendor for our prior experiments to study the SSD-based RAID. This 800GB SSD, named SSD-P, is equipped with 25nm MLC NAND flash memories and exposes four independent volumes to the host. Each volume is 200GB and can operate independently. We use the `mdadm` tool in Linux to combine the four volumes and create a single RAID-0 device (`/dev/md0`) for analyzing the interaction between internal parallelism and external parallelism. The high-speed PCI-E (x8) interface allows us to reach a bandwidth of nearly 2GB/s.

8.1. Microbenchmark

We first perform a set of microbenchmarks by using the Intel OpenStorage Toolkit to generate four types of workloads, *random read*, *sequential read*, *random write*, and *sequential write*, with various request sizes (4KB to 256KB), with a queue depth of 32 jobs. We report the aggregate bandwidth as the performance metric.

In a RAID-0 storage, logical blocks are evenly distributed across the participating devices in a round-robin manner, and the chunk size determines the size of the unit that is contiguously allocated in each individual device. In other words, the chunk size determines the data layout of RAID storage. By evenly distributing chunks, all participating devices can operate independently in parallel. Built on SSDs, where each

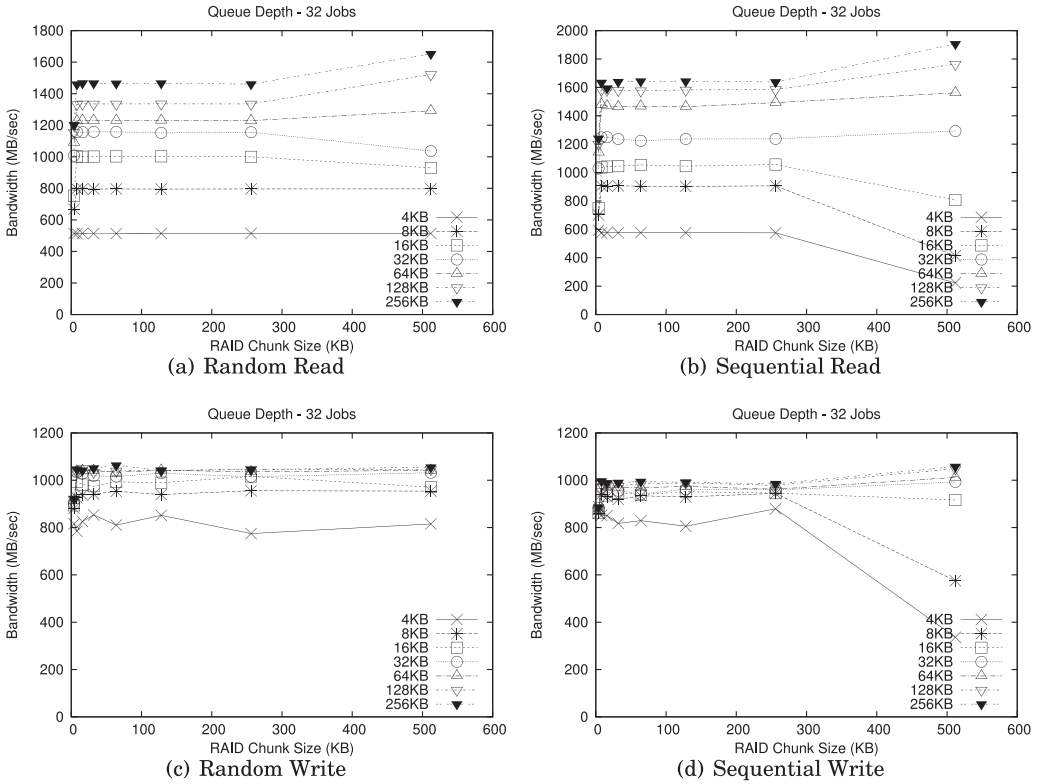


Fig. 14. Interaction of parallelism with RAID (queue depth, 32 jobs).

individual device already shows substantial performance potential for parallel I/Os, various RAID chunk sizes can influence the parallelism across the devices and also the overall performance of an RAID device. Figure 14 presents the results of the four workload patterns. We can see that for random reads with a relatively large request size, increasing chunk size can improve performance. For example, the 256KB workload can be improved from about 1,201MB/sec (chunk size 4KB) to 1,652MB/sec (chunk size 512KB). This is because with a large chunk size, a request is more likely to fall into only one device. For a large request, the internal parallelism of each device can be fully utilized, and a large queue depth ensures that all devices can be fully utilized, which leads to better performance. In contrast, this effect is not observed with small requests (e.g., 4KB), since the small requests cannot fully leverage the internal parallelism of each single device. For sequential reads (Figure 14(b)), as chunk size increases, we see a performance degradation with small requests. The reason is that as we increase the chunk size, despite the parallel I/Os, it is more likely to congest multiple requests in one device, since the access pattern is sequential rather than random. For example, having 32 parallel sequential requests of 4KB, the 32 requests can fall in one 128KB chunk, which leaves the other three devices unused. A similar pattern can be observed with sequential writes (see Figure 14(d)). For random writes, requests experience a performance increase as the chunk size increases, but the trend stops at small chunk size (e.g., 32KB). The reason is similar to random reads: A relatively larger write chunk size can ensure internal parallelism to be fully utilized first. However, writes cannot provide a headroom for performance improvement as large as reads. In general, we

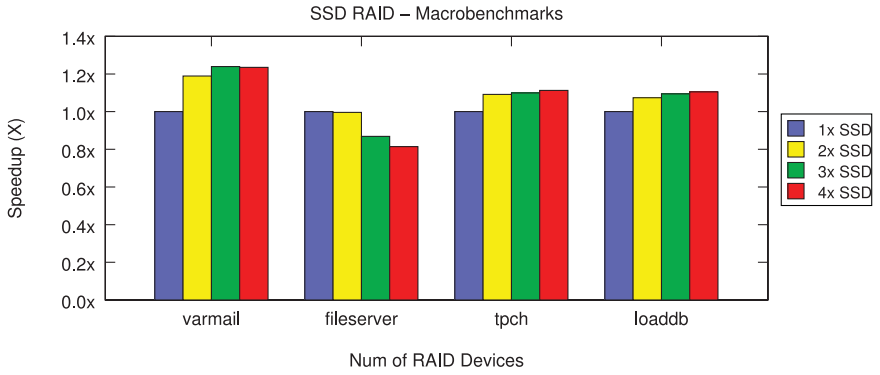


Fig. 15. Macrobenchmarks with SSD-based RAID.

can see that a 32KB chunk size is a sweet spot for this particular case. Comparing random writes and sequential writes, we also can see that both can reach similar peak performance (around 1GB/s), which again indicates that the SSD internally handles random and sequential writes similarly.

8.2. Macrobenchmarks

To show the impact of external parallelism to the end-to-end performance of applications, we select four different workloads as our macrobenchmarks: *Varmail* is a workload in the Filebench suite [Filebench 2015]. This workload simulates a mail server, which generates create-append-sync, read-append-sync, and read-delete file system operations. It has an equal amount of read and write operations. We configure the workload with 1,000,000 files, 16KB mean file size, and 100 threads. This benchmark reports the throughput (operations per second). *Fileserver* is another workload in Filebench. This workload simulates a file server, which generates create, delete, append, read, write, and attribute operations. The read/write ratio is about 1/2. In our experiments, we configure 100,000 files, a mean file size of 128KB, and 50 threads. *TPC-H* is an online analytical processing (OLAP) workload running with PostgreSQL database (v8.4). It is configured with a scale factor of 10, and we use the execution time of completing all 22 queries as our performance metric. *LoadDB* simulates a database loading process. It uses *dbgen* in the TPC-H benchmark to generate raw table files with a scale factor of 10. Then the database is created and data are loaded from the raw table files, and, finally, indexes are created. This workload reports the execution time. For our experiments, we configure four RAID configurations by using one to four devices and we desire to see the performance difference by increasing the opportunities of external parallelism. We use RAID-0 and a moderate chunk size (64KB). In order to show both throughput and execution time data in one figure, we normalize the results to the baseline case, using only one device. Figure 15 shows the results of performance speedups.

As shown in Figure 15, we can see that the impact of creating external parallelism opportunities to applications varies across the four workloads. For highly parallelized workloads, such as *varmail*, although its I/Os are relatively small (mean size is 16KB), increasing the number of RAID devices can effectively distribute the small I/O requests to different devices and bring substantial performance benefit (24%). *Fileserver* is interesting. It shows a decreasing performance as we add in more SSDs into the RAID. Considering our chunk size is 64KB and the mean file size is 128KB, many I/Os will split into multiple devices. In the meantime, since the workload has a mixed and relatively high write ratio, some SSDs could be congested or slowed down for handling write requests due to internal operations, such as garbage collection. As a result, a

Table III. SSD Specification

Name	Vendor	Capacity	Interface	Flash Type	Lithography
SSD-A	#1	240GB	SATA III	TLC NAND	19nm
SSD-B	#2	250GB	SATA III	TLC V-NAND	40nm
SSD-C	#2	256GB	SATA III	MLC V-NAND	40nm
SSD-D	#3	240GB	SATA III	MLC NAND	20nm
SSD-E	#3	400GB	SATA III	MLC NAND	25nm

request that is across multiple devices could be slowed down if any involved device is slow. Prior research, called Harmonia [Kim et al. 2011], has studied this effect in particular by coordinating garbage collections in SSDs. For workloads lacking sufficient parallelism but involving many I/O operations, such as *TPC-H* and *LoadDB*, we find that they only receive relatively moderate performance benefits (10%). For example, *LoadDB* is highly write intensive and most writes are sequential and large (512KB). As a result, increasing the number of devices can help leverage additional device bandwidth, but the low queue depth leaves less space for further improvement. Similarly, *TPC-H* is read intensive but also lacks parallelism. This case indicates that when creating an SSD-based RAID storage, whether a workload can receive benefits depends on the workload patterns. A basic rule is to fully utilize the parallelism delivered at both levels. In general, with sufficiently large and parallel I/Os, a reasonably large chunk size can effectively ensure each device to receive a sufficiently large request to exploit its internal parallelism and keep all device active. On the other hand, we should also be careful about the possible slowdown caused by cross-device dependency.

9. OBSERVATIONS WITH NEW-GENERATION SSDS

SSD technology is rapidly developing. In recent years, the industry has experienced a quick development in fabrication and manufacturing process of NAND flash. Nowadays, the fabrication process is entering a sub-20nm era; the high-density TLC NAND is replacing MLC and SLC NAND flash and becoming the mainstream; the 3D V-NAND flash memory technology further breaks the scale limit of the planar (2D) flash and provides higher reliability and better write speed. Inside SSDs, the FTL design is also becoming more and more sophisticated; most SSDs on the market are now adopting the the third-generation SATA interface, which can provide a peak bandwidth of 6.0Gbps; many devices are integrated with a large on-device DRAM/flash cache for buffering purposes. All these technologies together have enabled a great boost in performance and substantially reduced the average cost per gigabyte. On the other hand, as the SSD internals are increasingly complex, the performance behaviors of SSDs are becoming more and more diverse. In this section, we select five new-generation SSDs of three major brands on the market. These SSDs present a comprehensive set of devices for revisiting our experimental studies. Table III gives more details about these SSDs.

9.1. Performance Benefits of I/O Parallelism

Figure 16 shows the benefits of increasing I/O parallelism with the new SSDs. In comparison to results in Section 6.1, we made the following observations: (1) The new generation of SSDs has exhibited significantly higher performance. Almost all these SSDs can achieve a peak read bandwidth of above 450MB/sec and a peak write bandwidth of more than 400MB/s. Such a performance improvement is a result of both the interface upgrade and other optimizations. (2) Most SSDs show performance benefit from I/O parallelism, similarly to our observations in old-generation SSDs. This indicates that increasing I/O parallelism can enable several times higher speedups. As an exception, SSD-A show interference when increasing the queue depth for sequential reads over four parallel jobs. SSD-A is a product in the low-end range and

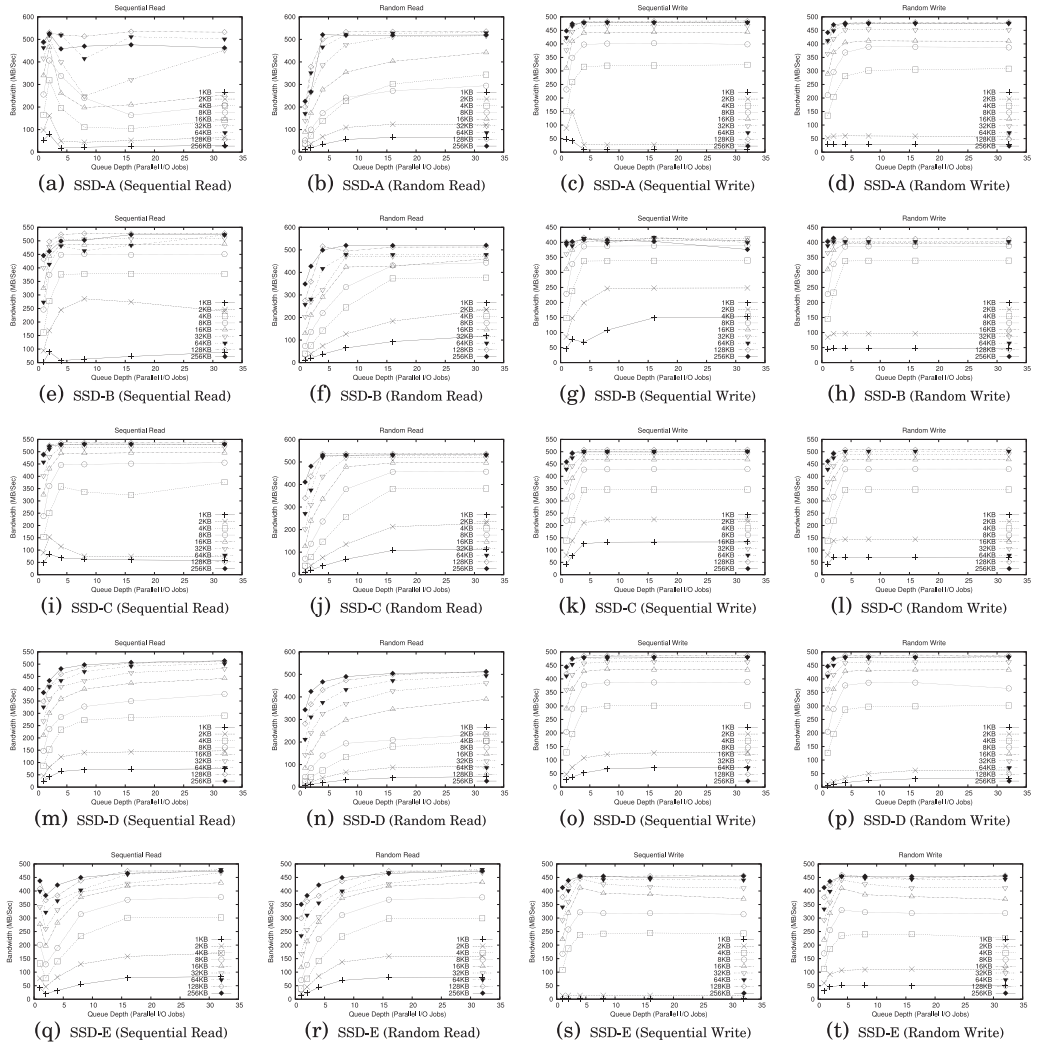
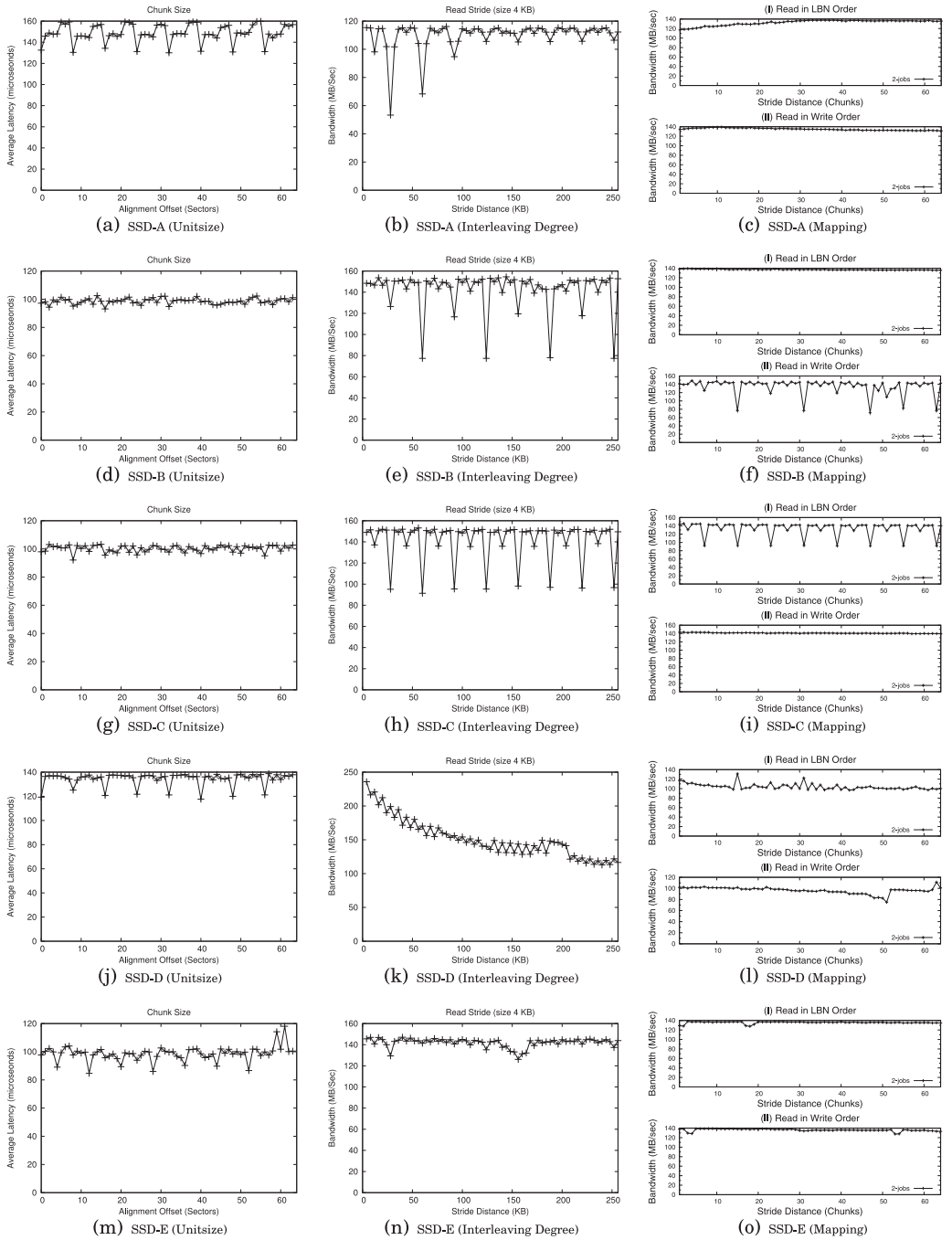


Fig. 16. Bandwidths of SSDs with increasing queue depth (the number of concurrent I/O jobs).

adopts low-cost high-density TLC planar NAND flash. Although the SSD includes a large DRAM/SLC cache on device, internal resource congestion still shows a strong interference with high I/O parallelism, and such a performance loss is evident when handling multiple concurrent I/O jobs.

9.2. Characterizing the SSD Internals

The new-generation SSDs have a complex internal structure. Some SSDs adopt new V-NAND flash chips and some have a large DRAM or SLC NAND flash cache (e.g., 512MB). Compared to early-generation SSDs, all these factors make characterizing SSD internals more difficult. In this section, we repeat the same set of experiments in Section 5. Figure 17 shows the results. In the figure, we can see that the curves do exhibit rather different patterns, as expected. We have the following observations: (1) For detecting the chunk size, some SSDs show repeatable and recognizable patterns



(a) SSD-A (Unitsize) (b) SSD-A (Interleaving Degree) (c) SSD-A (Mapping)

(d) SSD-B (Unitsize) (e) SSD-B (Interleaving Degree) (f) SSD-B (Mapping)

(g) SSD-C (Unitsize) (h) SSD-C (Interleaving Degree) (i) SSD-C (Mapping)

(j) SSD-D (Unitsize) (k) SSD-D (Interleaving Degree) (l) SSD-D (Mapping)

(m) SSD-E (Unitsize) (n) SSD-E (Interleaving Degree) (o) SSD-E (Mapping)

Fig. 17. Characterizing SSD Internals on new-generation SSDs.

but some others show no clear patterns. For example, SSD-A, SSD-D, and SSD-E show a regular pattern. SSD-A exhibits a two-step pattern for detecting the chunk size. This is likely to be caused by the different page access cost in high-density TLC NAND flash (LSB, CSB, MSB pages). According to the pattern, we still can roughly guess the logical unit size, which is very likely to be 4KB. SSD-D and SSD-E also show a clear pattern, which implies a 4KB page. SSD-B and SSD-C, however, show no strong patterns for accurately identifying the chunk size. It could be a result of the cache interference or a different implementation for handling a single chunk access. (2) For detecting the interleaving degree, SSD-A, SSD-B, and SSD-C produce regular patterns. SSD-A shows a regular but weakening pattern. According to the unit size, the number of channels is likely to be four channels. For SSD-B and SSD-C, both show a steady and repeatable pattern, based on which we believe the two SSDs have four channels as well. Interestingly, we have observed three repeatedly appearing dips on SSD-B and two dips on SSD-C, which indicates that SSD-B is more aggressively sharing the channels. SSD-D and SSD-E, which are built by the same vendor, exhibit an unclear pattern, which indicates that the internal implementation is fairly complex and interferes our detection based on an assumed architecture. (3) For the mapping policy, only SSD-B and SSD-C respond to our detecting mechanism and clearly shows two patterns: SSD-B uses a write-order based dynamic mapping, while SSD-C uses the LBN-based static mapping. The reason why the two SSDs from the same manufacture adopt two completely different mapping policy is unknown. One possible reason is that SSD-B uses a relatively lower-end TLC V-NAND flash chips, so using the dynamic write-order based mapping could better optimize the write performance. SSD-C, in contrast, uses higher-speed MLC V-NAND flash chips and therefore has a relatively lower pressure for handling writes.

In general, this set of experiments have confirmed that new-generation SSDs show fairly distinct patterns due to the complex internal designs, although some basic internal structure is still roughly characterizable. We expect that as SSD internals continue to be increasingly sophisticated, uncovering SSD internals completely by using a black-box approach would be more and more difficult. For this reason, directly exposing low-level hardware details to the upper-level components is a promising direction for future SSDs [Ouyang et al. 2014].

9.3. Impact of Garbage Collection

Garbage collection operations are the main source of internal overhead that affects observable SSD performance. While moving data internally inside an SSD, parallelizing I/Os may not achieve the same level of effectiveness as the situation without garbage collection. In this section, we illustrate such a situation by running 32 parallel 16KB requests. In order to trigger the garbage collection process, we first randomly write the device with 32 small (4KB) requests for 60s and 600 seconds, respectively, and then immediately start the workload, which generates 32 parallel 16KB requests in four different patterns. Figure 18 shows the comparison between the performance with and without garbage collection. As shown in the figures, after intensive random writes, the impact of garbage collection varies across different devices and workloads. In general, we observe less interference to reads caused by garbage collection than to writes. It is consistent with our understanding that, since most reads are synchronous and performance critical, SSD internals often optimize reads with a high priority. Also, as reads do not rely on garbage collection to refill the pool of clean blocks, garbage collection can be delayed to minimize the potential interference. In contrast, writes, especially random writes, are impacted by garbage collection greatly. With 60-second intensive writes, SSD-A, SSD-B, and SSD-C all have more than 50% bandwidth decrease for random writes. As we further increase the write volume by 10 times, we see that a

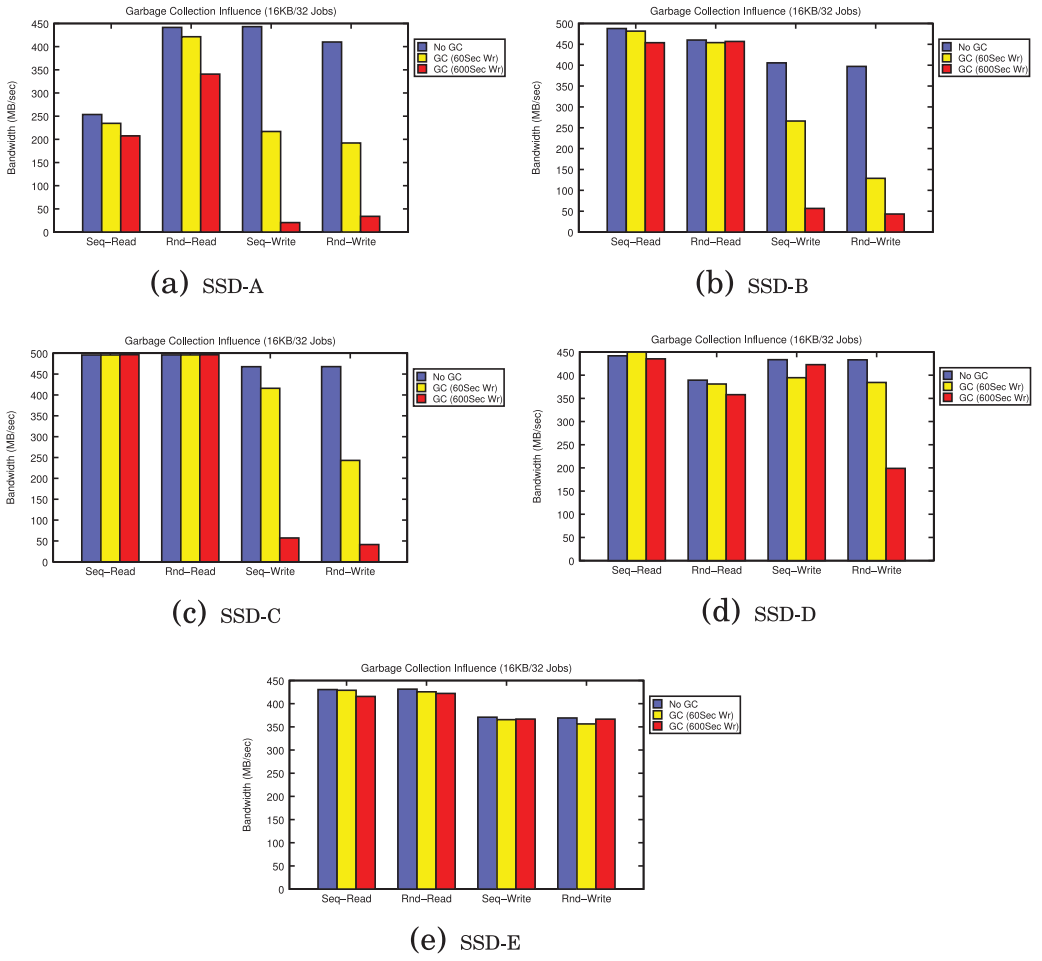


Fig. 18. Influence of garbage collection to the effectiveness of parallelizing I/Os.

significant performance decrease also appears in SSD-D, in addition to SSD-A, SSD-B, and SSD-C. SSD-E still performs well. We believe that this is due to the large over-provisioning ratio in SSD-E. In SSD-E, which is a high-end product designed for data centers, 25% of the NAND flash on SSD-E is used for over-provisioning, which absorbs heavy random write traffic. For mid-range products, such as SSD-B, SSD-C, and SSD-D, the performance impact of garbage collection after a long-time writes is evident. For this reason, some recent research, such as HIOS [Jung et al. 2014] and Harmonia [Kim et al. 2011], has worked on optimizing the efficiency of garbage collections in particular. On the other hand, we also need to note that most SSDs perform reasonably well with a moderate volume of random writes. Considering the fact that most writes are asynchronous, the end-to-end impact to applications would be further reduced by host side page cache.

9.4. Discussions

The implications of our experiments with new-generation SSDs are twofold: First, as SSD technologies advance and internal structure gets increasingly complex and

diverse, it is more and more difficult to accurately uncover the architectural details of the device. Our approach assumes a device as a black box that follows a general architecture design. Each new optimization integrated into the device will unfortunately weaken such an assumption as well as the observable performance patterns. In fact, our methodology is based on observable performance difference for handling various carefully designed workloads. With heavy optimizations, such a performance difference is diminishing or nearly unrecognizable, which is good for end users but bad news for researchers who desire to uncover the internal structures. On the other hand, we can see that many SSDs still can show certain patterns, more or less. This indicates that some general design principles are still in effect. Most importantly, the following key conclusion is still valid: In order to fully exploit the performance potential of flash SSD, we must fully utilize the internal parallelism. We believe this set of experiments with new-generation SSDs assures us the importance of internal parallelism in such new technologies.

10. IMPLICATIONS TO SYSTEM AND APPLICATION DESIGNERS

Having presented our experimental and case studies, we are in a position to present several important implications to system and application designers. Our experimental studies and analysis strongly indicate that we should consider parallelism as a *top priority* and revisit many existing optimizations designed for magnetic disks. We hope that our new findings can provide effective guidance and enhance our understanding on the properties of SSDs in the context of I/O parallelism. This section also summarizes our answers to the questions raised at the beginning of this article.

Benefits of parallelism on SSD: Parallelizing data accesses can provide substantial performance improvement, and the significance of such benefits depends on workload access patterns, resource redundancy, and flash memory mediums. In particular, small random reads benefit the most from parallelism. Large sequential reads achieve less significant but still impressive improvement. Such a property of SSDs provides many opportunities for performance optimization in application designs. For example, many critical database operations, such as index-tree search, feature intensive random reads, which is exactly a best fit in SSDs. Application designers can focus on parallelizing these operations. Meanwhile, sequential-access-dominated operations, such as hash join and sort, can be parallelized through aggressive asynchronous prefetching with proper hints from applications. In our experiments, we did not observe obvious negative effects of over-parallelization. In general, setting the concurrency level slightly over the number of channels is a reasonable choice. Finally, for practitioners who want to leverage the high parallel write performance, we recommend adopting high-end SSDs to provide more headroom for serving workloads with intensive writes. In our experiments, we also found a recent technical breakthrough in flash chips (e.g., 3D V-NAND) that makes MLC or even TLC flash SSDs perform reasonably well.

Random reads: An interesting finding is that with parallelism, random reads performance can be significantly improved to the level of large sequential reads without parallelism. Such a finding indicates that we cannot continue to assume that sequential reads are always better than random reads, which has been a long-existing common understanding for hard drives. This observation has a strong system implication. Traditionally, operating systems often make tradeoffs for the purpose of organizing large sequential reads. For example, the anticipatory I/O scheduler [Iyer and Druschel 2001] intentionally pauses issuing I/O requests for a while and anticipates organizing a larger request in the future. In SSDs, such optimization may become less effective and sometimes may be even harmful due to unnecessarily introduced delays. Issuing random reads as quickly as possible can also fully utilize the internal resources as sequential reads. On the other hand, some application designs could become more complicated.

For example, it is more difficult for the database query optimizer to choose an optimal query plan, because simply counting the number of I/Os and using static equations can no longer satisfy the requirement for accurately estimating the relative performance strengths of different join operators with parallelism. A parallelism-aware cost model is needed to handle such a new challenge.

Random writes: Contrary to our common understanding, parallelized random writes can achieve high performance, sometimes even better than reads. We also find that writes are no longer highly sensitive to patterns (random or sequential) as commonly believed. The uncovered mapping policy explains this surprising result from the architectural level: An SSD may internally handle random writes and sequential writes in the same way, regardless of access patterns. The implications are twofold. On one hand, we can consider how to leverage the high write performance through parallelizing writes. Some related problems would emerge, however, for example, how can we commit synchronous transaction logging in a parallel manner while maintaining a strong consistency [Chen 2009]? On the other hand, it means that optimizing random writes specifically for SSDs may not continue to be as rewarding as in early generations of SSDs, and trading off read performance for writes would become a less attractive option. But we should still note that in extreme conditions, such as day-long writes [Stoica et al. 2009] or under serious fragmentation [Chen et al. 2009], random writes are still a research issue.

Interference between reads and writes: In both microbenchmarks and macrobenchmarks, we find that parallel reads and writes on SSDs interfere strongly with each other and can cause unpredictable performance variance. In OS kernels, I/O schedulers should pay attention to this emerging performance issue and avoid mingling reads and writes as much as possible. A related research issue is concerned with how to maintain a high throughput while avoiding such interference. At the application level, we should also be careful of the way of generating and scheduling reads and writes. An example is the hybrid-hash joins in database systems, which have clear phases with read-intensive and write-intensive accesses. When scheduling multiple hybrid-hash joins, we can proactively avoid scheduling operations with different patterns. A rule of thumb is to schedule random reads together and separate random reads and writes whenever possible.

Physical data layout: The physical data layout in SSDs is dynamically determined by the order in which logical blocks are written. An ill-mapped data layout caused by such a dynamic write-order-based mapping can significantly impair the effectiveness of parallelism and readahead on device. In server systems, handling multiple write streams is common. Writes from the same stream can fall in a subset of domains, which would be problematic. We can adopt a simple random selection for scheduling writes to reduce the probability of such a worst case. SSD manufacturers can also consider adding a randomizer in the controller logic to avoid this problem. On the other hand, this mapping policy also provides us a powerful tool to *manipulate* data layout to our needs. For example, we can intentionally isolate a set of data in one domain to cap the usable I/O bandwidth and prevent malicious applications from congesting the I/O bus.

Revisiting application designs: Internal parallelism in SSDs brings many new issues that have not been considered before. Many applications, especially data-processing applications, are often heavily tailored to the properties of hard drives and are designed with many implicit assumptions. Unfortunately, these disk-based optimizations can become suboptimal on SSDs, whose internal physical structure completely differs. This calls for our attention to revisiting carefully the application designs to make them fit well in SSD-based storage. On the other hand, the internal parallelism in SSDs also enables many new opportunities. For example, traditionally DBMS designers often assume an HDD-based storage and favor large request sizes. SSDs can

extend the scope of using small blocks in DBMS and bring many desirable benefits, such as improved buffer pool usage, which is highly beneficial in practice.

External and internal parallelism: As SSDs are organized into an RAID storage, external parallelism interacts with internal parallelism. Our studies show that various workloads have different sensitivities to such an interaction. To effectively exploit parallelism opportunities at both levels, the key issue is to generate large, parallel I/Os to ensure each level have sufficient workload to be fully active. In our experiments, we also have confirmed that the actual effect of increasing external parallelism is application dependent, which needs our careful study on workload pattern.

Architectural complexity in new-generation SSDs: As flash memory and SSD technologies continue to advance, the increasingly diverse flash chip technologies (e.g., TLC, V-NAND) and complex FTL design make uncovering the SSD internals more and more challenging. Our studies on a set of recent products show that completely relying on black-box based techniques to uncover SSD internals will be difficult to accurately characterize all the architectural details. This suggests that SSD vendors should consider to expose certain low-level information directly to the users, which could enable a huge amount of optimization opportunities. On the other hand, our experiments also show that fully exploiting internal parallelism is still the most important goal.

In essence, SSDs represent a fundamental change of storage architecture, and I/O parallelism is *the key* to unlocking the huge performance potential of such an emerging technology. Most importantly, with I/O parallelism a low-end personal computer with an SSD is able to deliver as high an I/O performance as an expensive high-end system with a large disk array. This means that parallelizing I/O operations should be carefully considered in not only high-end systems but also in commodity systems. Such a paradigm shift would greatly challenge the optimizations and assumptions made throughout the existing system and application designs, which demands extensive research efforts in the future.

11. OTHER RELATED WORK

Flash memory-based storage technology is an active research area. In recent years, flash devices have received strong interest and been extensively studied [Chen 2009; Tsirogiannis et al. 2009; Lee et al. 2008; Canim et al. 2009; Do and Patel 2009; Agrawal et al. 2009; Lee and Moon 2007; Li et al. 2009; Nath and Gibbons 2008; Koltsidas and Viglas 2008; Graefe 2007; Lee et al. 2009b; Bouganim et al. 2009; Stoica et al. 2009; Josephson et al. 2010; Soundararajan et al. 2010; Boboila and Desnoyers 2010]. Open-source projects, such as OpenNVM [2015] and OpenSSD [2015], are also under active development. In this section we present the research work that is most related to internal parallelism of SSDs.

A detailed description of the hardware internals of flash memory-based SSD has been presented in prior work [Agrawal et al. 2008; Dirik and Jacob 2009]. Another early work has compared the performance of SSDs and HDDs [Polte et al. 2008]. These studies on architectural designs of SSDs regard parallelism as an important consideration to provide high bandwidth and improve high-cost operations. In our prior work [Chen et al. 2009], we have studied SSD performance by using non-parallel workloads, and the main purpose is to reveal the internal behavior of SSDs. This article focuses on parallel workloads. Bouganim et al. have presented a benchmark tool called uFLIP to assess the flash device performance and found that increasing concurrency cannot improve performance of early generations of SSDs [Bouganim et al. 2009]. Lee et al. have studied the advances of SSDs and reported that with NCQ support, the recent generation of SSDs can achieve high throughput [Lee et al. 2009b]. Hu et al. have studied the multilevel internal parallelism in SSDs and pointed out that different levels of parallelism have distinct impact to the effect of optimization [Hu

et al. 2013]. Jung and Kandemir have revisited SSD design [Jung and Kandemir 2013] and found that internal parallelism resources have been substantially underutilized [Jung and Kandemir 2012]. This article and its early version [Chen et al. 2011a] show that internal parallelism is a critical element to improve I/O performance and must be carefully considered in system and application designs.

Another set of prior studies has applied optimization by leveraging internal parallelism. Jung et al. have proposed a request scheduler, called PAQ, in order to avoid resource contention for reaching full parallelism [Jung et al. 2012]. Sprinkler [Jung and Kandemir 2014] further improves internal parallelism by scheduling I/O requests based on internal resource layout. Roh et al. introduce B+-tree optimization methods by utilizing internal parallelism and also confirm our findings [Roh et al. 2012]. Wang et al. present an LSM-tree-based key-value store on open-channel SSD by exploiting the abundant internal parallelism resources in SSDs [Wang et al. 2014]. Park and Shen present an I/O scheduler, called FIOS, for flash SSDs and exploiting internal parallelism is also a design consideration [Park and Shen 2012]. Ouyang et al. present a software defined flash (SDF) design to directly expose internal parallelism resources to applications for the purpose of fully utilizing available internal parallelism in flash [Ouyang et al. 2014]. Guo et al. have proposed a write buffer management scheme, called PAB, to leverage parallelism inside SSDs [Guo et al. 2013]. Vasudevan et al. also present a vector interface to exploit potential parallelism in high-throughput NVM systems.

The database community is also highly interested in flash SSDs to enhance DB performance. For example, a mechanism called *FlashLogging* [Chen 2009] adopts multiple low-cost flash drives to improve log processing, and parallelizing writes is an important consideration. Das et al. present an application-level compression scheme for flash and adopt a parallelized flush method, called MT-Flush, in the MySQL database, which optimizes the flash storage performance [Das et al. 2014]. Tsirogiannis et al. have proposed a set of techniques including fast scans and join customized for SSDs [Tsirogiannis et al. 2009]. Similarly, our case studies in PostgreSQL show that with parallelism, many DBMS components need to be revisited, and the disk-based optimization model would become problematic and even error prone on SSDs. These related works and our study show that internal parallelism is crucial to enhancing system and application performance and should be treated as a top priority in the design of systems and applications.

12. CONCLUSIONS

This article presents a comprehensive study to understand and exploit internal parallelism in SSDs. Through extensive experiments and analysis, we would like to deliver three key messages: (1) Effectively exploiting internal parallelism of SSDs indeed can significantly improve I/O performance and also largely remove performance limitations of SSDs. We need to treat I/O parallelism as a *top priority* for optimizing I/O performance. (2) We must also pay close attention to some potential side effects, such as the strong interference between reads and writes and the ill-mapped data layout problem, and minimize its impact. (3) Most importantly, we also find that many of the existing optimizations, which are specifically tailored to the properties of disk drives, can become ineffective and sometimes even harmful on SSDs. We must revisit such designs carefully. It is our hope that this work can provide insight into the highly parallelized internal architecture of SSDs and also provide guidance to the application and system designers for effectively utilizing this unique merit of SSDs.

REFERENCES

Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD/PODS Conference*. ACM, New York, NY, 967–980.

D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. 2009. Lazy-adaptive tree: An optimized index structure for flash devices. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX'08)*.

A. Birrell, M. Isard, C. Thacker, and T. Wobber. 2005. A design for high-performance flash disks. In *2005 Microsoft Research Technical Report*.

Blktrace. 2011. Homepage. Retrieved from http://linux.die.net/man/8/blktrace.

S. Boboila and P. Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

L. Bouganim, B. Jónsson, and P. Bonnet. 2009. uFLIP: Understanding flash IO patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems (CIDR'09)*.

M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. 2009. An object placement advisor for DB2 using solid state storage. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

F. Chen, D. A. Koufaty, and X. Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*. ACM, New York, NY.

Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011a. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

F. Chen, T. Luo, and X. Zhang. 2011b. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*.

Feng Chen, Michael P. Mesnier, and Scott Hahn. 2014. A protected block device for persistent memory. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*.

S. Chen. 2009. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. 2014. NVM compression - hybrid flash-aware application level compression. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2014. Deconstructing storage arrays. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*.

C. Dirik and B. Jacob. 2009. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*.

J. Do and J. M. Patel. 2009. Join processing for flash SSDs: Remembering past lessons. In *Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMon'09)*.

Janene Ellefson. 2013. NVM express: Unlock your solid state drives potential. In *2013 Flash Memory Summit*.

Filebench. 2015. Homepage. Retrieved from http://sourceforge.net/projects/filebench/.

E. Gal and S. Toledo. 2005. Algorithms and data structures for flash memories. In *ACM Comput. Surv.* 37, 2 (2005), 138–163.

Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

G. Graefe. 2007. The five-minute rule 20 years later, and how flash memory changes the rules. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware (DaMon'07)*.

Xufeng Guo, Jianfeng Tan, and Yuping Wang. 2013. PAB: Parallelism-aware buffer management scheme for nand-based SSDs. In *Proceedings of IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'13)*.

Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2013. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. In *IEEE Trans. Comput*. 62, 6 (2013), 1141–1155.

S. Iyer and P. Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th Symposium on Operating System Principles (SOSP'01)*.

S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'05)*.

W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'10)*.

Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. 2014. HIOS: A host interface I/O scheduler for solid state disks. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*.

Myoungsoo Jung and Mahmut Kandemir. 2012. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'12)*.

Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'13)*. ACM, New York, NY.

Myoungsoo Jung and Mahmut T. Kandemir. 2014. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

Myoungsoo Jung, Ellis H. Wilson, and Mahmut Kandemir. 2012. Physically addressed queuing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

T. Kgil, D. Roberts, and T. Mudge. 2008. Improving NAND flash based disk caches. In *Proceedings of the 35th International Conference on Computer Architecture (ISCA'08)*.

Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. 2011. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST'11)*.

I. Koltsidas and S. Viglas. 2008. Flashing up the storage layer. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*.

R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. 2009a. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

S. Lee and B. Moon. 2007. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD Conference (SIGMOD'07)*.

S. Lee, B. Moon, and C. Park. 2009b. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. 2008. A case for flash memory SSD in enterprise database applications. In *Proceedings of 2008 ACM SIGMOD Conference (SIGMOD'08)*. ACM, New York, NY,

Y. Li, B. He, Q. Luo, and K. Yi. 2009. Tree indexing on flash disks. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*.

M. Mesnier. 2011. Intel Open Storage Toolkit. http://www.source forge.org/projects/intel-iscsi. (2011).

M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger. 2007. Modeling the relative fitness of storage. In *Proceedings of 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY.

Micron. 2007. Micron 8, 16, 32, 64Gb SLC NAND Flash Memory Data Sheet. Retrieved from http://www.micron.com/.

S. Nath and P. B. Gibbons. 2008. Online maintenance of very large random samples on flash storage. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*.

NVM Express. 2015. Homepage. Retrieved from http://www.nvmexpress.org.

P. O'Neil, B. O'Neil, and X. Chen. 2009. Star Schema Benchmark. Retrieved from http://www.cs.umb.edu/~poneil/StarSchemaB.pdf.

OpenNVM. 2015. Homepage. Retrieved from http://opennvm.github.io.

OpenSSD. 2015. Homepage. Retrieved from http://www.openssd-project.org/.

Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. 2006. A high performance controller for NAND flash-based solid state disk (NSSD). In *Proceedings of the 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW'06)*.

S. Park and K. Shen. 2012. FIOS: A fair, efficient flash I/O scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

D. Patterson, G. Gibson, and R. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*. ACM, New York, NY.

PC Perspective. 2009. OCZ Apex Series 250GB Solid State Drive Review. Retrieved from http://www.pcper.com/article.php?aid=661.

Pgbench. 2011. Homepage. Retrieved from http://developer.postgresql.org/pgdocs/postgres/pgbench.html.

M. Polte, J. Simsa, and G. Gibson. 2008. Comparing performance of solid state devices and mechanical disks. In *Proceedings of the 3rd Petascale Data Storage Workshop*.

T. Pritchett and M. Thottethodi. 2010. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of International Symposium on Computer Architecture (ISCA'10)*.

Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2012. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. In *Proceedings of VLDB Endowment (VLDB'12)*.

M. Rosenblum and J. K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst*. 10, 1 (1992), 26–52.

Samsung. 2007. Datasheet (K9LBG08U0M). Retrieved from http://www.samsung.com.

SATA. 2011. Serial ATA Revision 2.6. Retrieved from http://www.sata-io.org.

M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. 2008. Fast scans and joins using flash drives. In *Proceedings of the 4th International Workshop on Data Management on New Hardware (DaMon'08)*.

S. Shah and B. D. Noble. 2007. A study of e-mail patterns. In *Software Practice and Experience*, Vol. 37(14). 1515–1538.

G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. 2009. Evaluating and repairing write performance on flash devices. In *Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMon'09)*.

Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. 2007. One size fits all? Part 2: Benchmarking studies. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems (CIDR'07)*.

D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. 2009. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. Amsterdam, The Netherlands.

Guanying Wu and Xubin He. 2014. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

Received December 2014; revised June 2015; accepted August 2015