# Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing

Feng Chen[2*] Rubao Lee[1,2] Xiaodong Zhang[2]
[1]Institute of Computing Technology
Chinese Academy of Sciences
liru@cse.ohio-state.edu
[2]Dept. of Computer Science & Engineering
The Ohio State University
{fchen, zhang}@cse.ohio-state.edu

## Abstract

*Flash memory based solid state drives (SSDs) have shown a great potential to change storage infrastructure fundamentally through their high performance and low power. Most recent studies have mainly focused on addressing the technical limitations caused by special requirements for writes in flash memory. However, a unique merit of an SSD is its rich internal parallelism, which allows us to offset for the most part of the performance loss related to technical limitations by significantly increasing data processing throughput.*

*In this work we present a comprehensive study of essential roles of internal parallelism of SSDs in high-speed data processing. Besides substantially improving I/O bandwidth (e.g. 7.2x), we show that by exploiting internal parallelism, SSD performance is no longer highly sensitive to access patterns, but rather to other factors, such as data access interferences and physical data layout. Specifically, through extensive experiments and thorough analysis, we obtain the following new findings in the context of concurrent data processing in SSDs. (1) Write performance is largely independent of access patterns (regardless of being sequential or random), and can even outperform reads, which is opposite to the long-existing common understanding about slow writes on SSDs. (2) One performance concern comes from interference between concurrent reads and writes, which causes substantial performance degradation. (3) Parallel I/O performance is sensitive to physical data-layout mapping, which is largely not observed without parallelism. (4) Existing application designs optimized for magnetic disks can be suboptimal for running on SSDs with parallelism. Our study is further supported by a group of case studies in database systems as typical data-intensive applications. With these critical findings, we give a set of recommendations to application designers and system architects for exploiting internal parallelism and maximizing the performance potential of SSDs.*

## 1 Introduction

The I/O performance of Hard Disk Drives (HDDs) has been regarded as a major performance bottleneck for high-speed data processing, due to the excessively *high latency* of HDDs for random data accesses and *low throughput* of HDDs for handling multiple concurrent requests. Recently Flash Memory based Solid State Drives (SSDs) have been incorporated into storage systems. Unlike HDDs, an SSD is built entirely of semiconductor chips with no moving parts. Such an architectural difference provides a great potential to address fundamentally the technical issues of rotating media. In fact, researchers have made extensive efforts to adopt this solid state storage technology in storage systems and proposed solutions for performance optimizations (e.g. [8, 18, 27, 31]). Most of the work has focused on leveraging the high random data access performance of SSDs and addressing their technical limits (e.g. slow random writes). Among these studies, however, an important functionality of SSDs has not received sufficient attention and is rarely discussed in the existing literature, which is the *internal parallelism*, a unique and rich resource provided by SSDs. Parallelism has been reported earlier not to be beneficial to performance [5]. A recent study [20] on the advances of SSD technology reports that with the support of native command queuing (NCQ), the state-of-the-art SSDs are capable of handling multiple I/O jobs simultaneously and achieving a high throughput. In this paper, we will show that the impact of internal parallelism is far beyond the scope of the basic operations of SSDs. We believe that the use of internal parallelism can lead to a fundamental change in the current sequential-access-oriented optimization model adopted in system and application designs, which is particularly important for data-intensive applications.

### 1.1 Internal Parallelism in SSDs

There are two inherent architectural limitations in the design of SSDs. First, due to current technical constraints, one single flash memory package can only provide limited bandwidth (e.g. 32-40MB/sec [3]). Second, writes in flash memory are often much slower than reads, and many critical operations, such as garbage collection and wear-leveling [3, 6, 10], can incur latencies as high as milliseconds.

To address these limitations, SSD architects have built an ingenious structure to provide internal parallelism – Most SSDs are built on an array of flash memory packages, which are connected through multiple (e.g. 2-10) channels to flash

*Currently working at the Intel Labs in Hillsboro, OR.

memory controllers. SSDs provide logical block addresses (LBA) as a logical interface to the host. Since logical blocks can be striped over multiple flash memory packages, data accesses can be conducted 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 can be effectively hidden behind other concurrent operations. Therefore, the 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 HDD-based storage, such as RAID [24], a storage based on multiple hard disks. However, there are two fundamental differences between the SSD and RAID architectures, which demand thorough and new investigations: (1) *Different logical/physical mapping mechanisms* – In RAID, a logical block is *statically* mapped to a "fixed" physical location, which is determined by its logical block number (LBN) [24]. In SSD, a logical block can be *dynamically* mapped to any physical location, which changes with writes during runtime. This has called our particular attention to a unique data layout problem in SSDs, and we will show that an ill-mapped data layout can cause significant performance degradation, while such a problem is unlikely to happen in RAID. (2) *Different physical natures* – RAID is built on magnetic disks, whose random access performance is often one order of magnitude lower than sequential access, while SSDs are built on flash memories, in which such a performance gap is much smaller. The difference in physical natures can strongly impact the existing sequentiality-based application designs, because without any moving parts, parallelizing I/O operations on SSDs can make random accesses capable of performing comparably or even slightly better than sequential accesses, while this is difficult to implement in RAID. Therefore, a careful and insightful study of the internal parallelism of this emerging storage technology is highly desirable for storage architects, system designers, and data-intensive application users.

## 1.2 Research and Technical Challenges

On one hand, the internal parallelism makes a single SSD capable of handling multiple incoming I/O requests in parallel and achieving a high bandwidth. On the other hand, the internal parallelism cannot be effectively exploited unless we address the three following critical challenges.

- **The 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 data layout across flash memory packages, 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, the anticipated performance gains are difficult to achieve. Meanwhile, uncovering such a low-level architectural information without changing the strictly defined interface is very challenging. In this paper we will present a set of simple yet effective experimental techniques to achieve this goal.
- **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 generally 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. Only by understanding both benefits and side effects of I/O parallelism on an SSD can we effectively exploit its performance potential while avoiding its negative effects.
- **Exploiting internal parallelism in SSDs demands fundamental changes to the existing program design and the sequential-access-oriented optimization model adopted in software systems.** Most existing applications, such as database systems, assume that the underlying storage device is a hard disk drive. As a result, they mostly focus on how to *sequentialize* rather than *parallelize* data accesses for improving storage performance (e.g. [16]). 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 not only show that speedups by up to a factor of 5 can be achieved by parallelizing a query with multiple sub-queries, but more importantly, we also show that with I/O parallelism, the *query optimizer*, a key component in database systems, would make an *incorrect* decision when selecting the optimal query plan, which should receive a particular attention from application and system designers.

## 1.3 Critical Issues for Investigation

In this work, we strive to address the above three challenges by answering several critical questions related to the internal parallelism of SSDs and revealing some untold facts and unexpected dynamics in SSDs.

- Limited by the 'thin' interface between the storage device and the host, how can we effectively uncover the key architectural features of an SSD? Of the most interest, how is the physical data layout determined in an SSD?
- The effectiveness of parallelizing data access depends on many factors, such as workload access patterns, resource redundancy, and others. Can we quantify the benefit of I/O parallelism and its relationship to these factors?

- Reads and writes on SSDs both generate many asynchronous background operations and thus may interfere with each other. Can we quantitatively show such interactive effects between parallel data accesses? Would such interference impair the effectiveness of parallelism?
- Readahead improves read performance on SSDs but it is sensitive to read patterns [6]. Would I/O parallelism affect the readahead? How should we choose between increasing parallelism and retaining effective readahead?
- The physical data layout on an SSD could change on the fly. How does an ill-mapped data layout affect the efficiency of I/O parallelism and the readahead?
- Applications can leverage internal parallelism to optimize the data access performance. How much performance benefit can we achieve in practice?
- Many optimizations in applications are specifically tailored to the properties of hard disk drives and may be ineffective for SSDs. Can we make a case that parallelism-based optimizations would become a critical consideration for maximizing data access performance on SSDs?

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

The rest of this paper is organized as follows. Section 2 provides the background. Section 3 introduces our experimental system and methodology. Section 4 presents how to detect the SSD internals. Section 5 and 6 present our experimental and case studies on SSDs. Section 7 discusses the system implications of our findings. Related work is given in Section 8. The last section concludes this paper.

## 2 Background of SSD Architecture

### 2.1 SSD Internals

A typical SSD includes four major components (Figure 1). A *host interface logic* connects to the host through an interface connection (e.g. SATA or IDE bus). An *SSD controller* manages flash memory space, translates incoming requests, and issues commands to flash memory packages via a *flash memory controller*. Some SSDs have a dedicated DRAM *buffer* to hold metadata or data, and some SSDs only use a small integrated SRAM buffer to lower production cost. In most SSDs, multiple (e.g. 2-10) channels are used to connect the controller with flash memory packages. Each channel may be shared by more than one package. Actual implementations may vary across different models, and previous work [3, 10] gives detailed descriptions about the architecture of SSDs.

By examining the internal architectures of SSDs, we can find that parallelism is available at different levels, and operations at each level can be parallelized or interleaved.

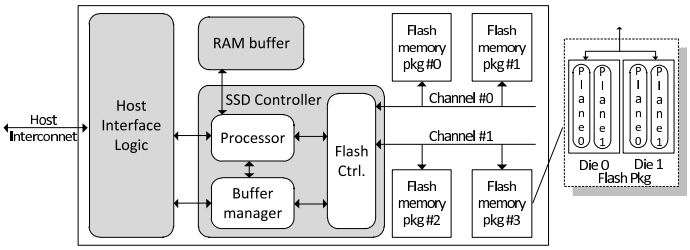



**Figure 1.** An illustration of SSD architecture [3].

- **Channel-level Parallelism** – In an SSD, flash memory packages are connected to the controller through multiple channels. Each channel can be operated independently and simultaneously. Some SSDs adopt multiple Error Correction Code (ECC) engines and flash controllers, each for a channel, for performance purposes [23].
- **Package-level Parallelism** – In order to optimize resource utilization, a channel is usually shared by multiple flash memory packages. Each flash memory package can be operated independently. Operations on flash memory packages attached to the same channel can be interleaved, so the bus utilization can be optimized [3, 10].
- **Die-level Parallelism** – A flash memory package often includes two or more dies (chips). Each die can be selected individually and execute a command independent of the others, which increases the throughput [28].
- **Plane-level Parallelism** – A flash memory chip is typically composed of two or more planes. Most flash memories (e.g. [2, 28]) support performing the same operation (e.g. read/write/erase) on multiple planes simultaneously. Some flash memories (e.g. [2]) provide `cache mode` to further parallelize medium access and bus transfer.

Such a highly parallelized structure provides rich opportunities for parallelism. In this paper we will present several simple yet effective experimental techniques to uncover the internal parallelism on various levels.

### 2.2 Native Command Queuing (NCQ)

Native command queuing (NCQ) is a feature introduced by the SATA II standard [1]. With NCQ support, the device can accept multiple incoming commands from the host and schedule the jobs internally. NCQ is especially important to SSDs, 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). Early generations of SSDs do not support NCQ and thus cannot benefit from parallel I/O [5]. The two SSDs used in our experiments can accept up to 32 jobs.

## 3 Measurement Environment

### 3.1 Experimental Systems

Our experiments have been conducted on a Dell™ Precision™ T3400. It features an Intel® Core™2Duo E7300 2.66GHz processor and 4GB main memory. A 250GB Seagate 7200RPM hard disk drive is used to hold 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.

| | SSD-M | SSD-S |
|---|---|---|
| Capacity | 80GB | 32GB |
| NCQ | 32 | 32 |
| 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) | 3500 | 700 |

**Table 1.** Specification data of the SSDs.

We have selected two representative, state-of-the-art SSDs fabricated by a well-known SSD manufacturer for our studies. 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. Both SSDs deliver market-leading performance with full support of NCQ. Their designs are consistent with general-purpose SSD designs (e.g. [3, 10]) and representative in the mainstream technical trend on the market. PCI-E based flash devices (e.g. Fusion-io's ioDrive [11]) are designed for special-purpose systems with a different structure [17]. In this paper we focus on the SATA-based devices. Table 1 shows more details about the two SSDs.

Similar to our prior work [6], we use the *CFQ* (Completely Fair Queuing) scheduler, the default I/O scheduler in the Linux kernel, for the hard disk. We use the *noop* (No-optimization) scheduler for the SSDs to expose the internal behavior of the SSDs for our analysis.

### 3.2 Experimental Tools and Workloads

To study the internal parallelism of SSDs, two tools are used in our experiments. We use the Intel® Open Storage Toolkit [21] to generate various types of I/O workloads with different configurations, such as read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent I/O jobs), and others. It reports bandwidth, IOPS, and latency. We also designed a tool, called *replayer*, which accepts a pr-recorded trace file for input and replays data accesses to a storage device. It facilitates precisely repeating a workload directly at the block device level. We use the two tools to generate three access patterns.

- **Sequential** - Sequential data accesses using specified request size, starting from sector 0.
- **Random** - Random data accesses using specified request size. Blocks are randomly selected from the first 1024MB of the storage space, unless otherwise noted.
- **Stride** - Strided data accesses using specified request size, starting from sector 0 with a stride distance.

In our experiments, each workload runs for 30 seconds in default to limit trace size while collecting sufficient data. No partitions or file systems are created on the SSDs. Unlike previous work [26], which was performed over an Ext3 file system, all workloads in our experiments directly access the SSDs as raw block devices. All requests are issued to the devices synchronously with no think time.

Similar to the methods in [6], 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 largely remain constant across experiments [6].

### 3.3 Trace Collection

To analyze I/O traffic in detail, we use *blktrace* [4] to trace the I/O activities at the block device level. 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* [4] and our post-processing scripts and tools off line.

## 4 Uncovering SSD Internals

Before introducing our performance studies on the internal parallelism of SSDs, 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. Knowing the key architectural features of an SSD is required to study and understand the observed device behavior. For example, our findings about the write-order based mapping (see Section 4.4) motivate 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 in practice. For example, knowing the number of channels in an SSD, we can set a proper concurrency level and avoid over-parallelization.

On the other hand, obtaining such architectural information is particularly challenging for several reasons. (1) Architectural details about the SSD design are often regarded as critical intellectual property of manufacturers. To the best of our knowledge, certain information, such as the mapping policy, is not available in any datasheet or specification, though it is critical 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, across different product batches, hardware/firmware change is common and often cannot be timely reflected in public documents. (3) Most SSDs on the market carefully follow a strictly defined host interface standard (e.g. SATA [1]). Only limited information is allowed to pass through such a 'thin interface'. As a result, 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.

### 4.1 A Generalized Model

Despite various implementations, most SSDs strive to optimize performance essentially in a similar way – evenly distributing data accesses to maximize resource usage. Such a principle is applicable to the parallel structure of SSDs at different levels. Without losing generality, we define an abstract model to characterize such an organization based on open documents (e.g. [3, 10]): A *domain* is a set of flash memories that share a specific set of resources (e.g. channels). A domain can be further partitioned into *sub-domains* (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 by following a *mapping policy*. A set of chunks across each of $N$ domains are called a *stripe*. One may notice that this model is in principle similar to RAID [24]. In fact, SSD architects often adopt a RAID-0 like striping mechanism [3, 10], some even directly integrate a RAID controller inside an SSD [25]. However, as mentioned previously, the key difference is that SSDs use dynamic mapping, which may cause an ill-mapped data layout as we will see later.

In this work, we are particularly interested in examining three *key factors* that are directly related to internal parallelism. In our future work we will further extend the techniques to uncover other architectural features.

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

In the following, we will present a set of experimental approaches to *infer indirectly* the three key factors. 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, we observe the 'reactions' of the SSD, measured in several key metrics, e.g. latency and bandwidth. Based on this probing information, we can speculate the internal architectures and policies adopted in the SSD. Our solution shares a similar principle on characterizing RAID [9], 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 picture all the internal details, and our purpose is not to reverse-engineer the SSD hardware. Instead, we try to characterize the key architectural features of an SSD 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 in serving our performance analysis and optimization purposes. For brevity we only show the results for SSD-S. SSD-M behaves similarly.

### 4.2 Chunk Size

**Program 1** Pseudocode of uncovering SSD internals

```
init_SSD():  sequentially write SSD w/ 256KB req.
rand_pos(A): get a random offset aligned to A sect.
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 max. possible chunk size
D:  an estimated max. possible interleaving degree

(I) detecting chunk size:
  init_SSD();              // initialize SSD space
  for (n = 1 sector; n <= M; n *= 2): //req. 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 avg. lat.

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

As a basic mapping unit, a chunk can be mapped in only one domain, and two continuous chunks are mapped in two separate domains. Suppose 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. The latter case would be faster. Based on this feature, we designed an experiment to identify the chunk size. 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 [10]. We see a flat curve for 1 sector (512 bytes), because the smallest read/write unit in the OS kernel is one sector and cannot be mapped across two domains.

### 4.3 Interleaving Degree

In our model, chunks are organized into domains based on resource redundancy. Parallel accesses to data in multiple domains without resource sharing can achieve a higher bandwidth than doing that congested in one domain. Based on this feature, we designed an experiment to determine the interleaving degree, as shown in Program 1(II).

Figure 3(I) and (II) show the experimental results with 2 jobs and 4 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.* In this case, we observe 10 domains, each of which actually corresponds to one channel.

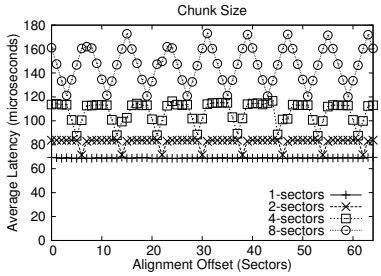



**Figure 2.** Detecting the chunk size on SSD-S. Each curve corresponds to a specified request size.

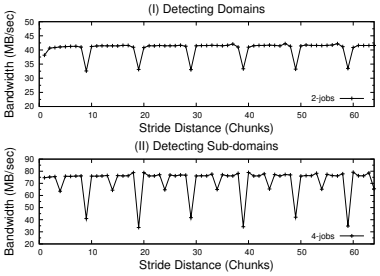



**Figure 3.** Detecting the interleaving degree on SSD-S. Figure (I) and (II) use 2 jobs and 4 jobs, respectively.

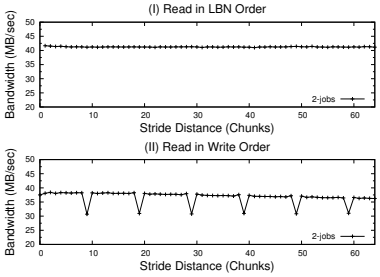



**Figure 4.** Detecting the mapping policy on SSD-S. Figure (I) and (II) represent two test cases.

The rationale is as follows. Suppose the number of domains is $D$ and the data is 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 would be distributed across two domains and thus could achieve a higher bandwidth (around 40MB/sec). We can also observe 2 sub-domains using a similar approach, as shown in Figure3 (II).

### 4.4 The Mapping Policy

The mapping policy determines the physical page to which a logical page is mapped. According to open documents (e.g. [3, 10]), two mapping policies are widely used in practice. (1) *LBA-based mapping* – for a given logical block address ($LBA$) with an interleaving degree, $D$, the block is mapped to a domain number ($LBA$ mod $D$). (2) *Write-order-based mapping* – for the $i_{th}$ write, the block is assigned to a domain number ($i$ mod $D$). We should note here that write-order-based mapping is *not* the log-structured mapping policy [3], 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. After such a randomization, the blocks are relocated across the domains. If LBA-based mapping is adopted, the mapping should not be affected by such random writes. Thus, we repeat the experiment in Section 4.3, and we find after randomization, the pattern (repeatedly appearing dips) disappears (see Figure 4(I)), which means that the random overwrites have changed the block mapping, and LBA-based mapping is not used.

We then conduct another experiment to confirm that write-order-based mapping is adopted. 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 4.3. For example, for the LBNs of random writes in the order of (91, 100, 23, 7, ...), we read data in the same order (91, 100, 23, 7, ...). If write-order based mapping is used, the blocks should be interleavingly allocated across domains in the order of writes (e.g. blocks 91, 100, 23, 7 are mapped to domain 0, 1, 2, 3, respectively), and reading data in the same order would repeat the same pattern (dips) we see before. Our experimental results confirm this hypothesis (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 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 interference between parallel jobs*, *the readahead mechanism with I/O parallelism*, and *the performance impact of physical data layout*.

### 5.1 What are the benefits of parallelism?

In order to quantify 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[1], and we show the bandwidths with a queue depth increasing from 1 job to 32 jobs. In order to compare workloads with different request sizes, we use bandwidth (MB/sec), instead of IOPS (IO per second), as the performance metric. Figure 5 shows the experimental results for SSD-M and SSD-S. Since 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 [5], our experiments show a great performance improvement from parallelism on SSDs. The significance of performance gains depends on several factors, and we present several key observations here.

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

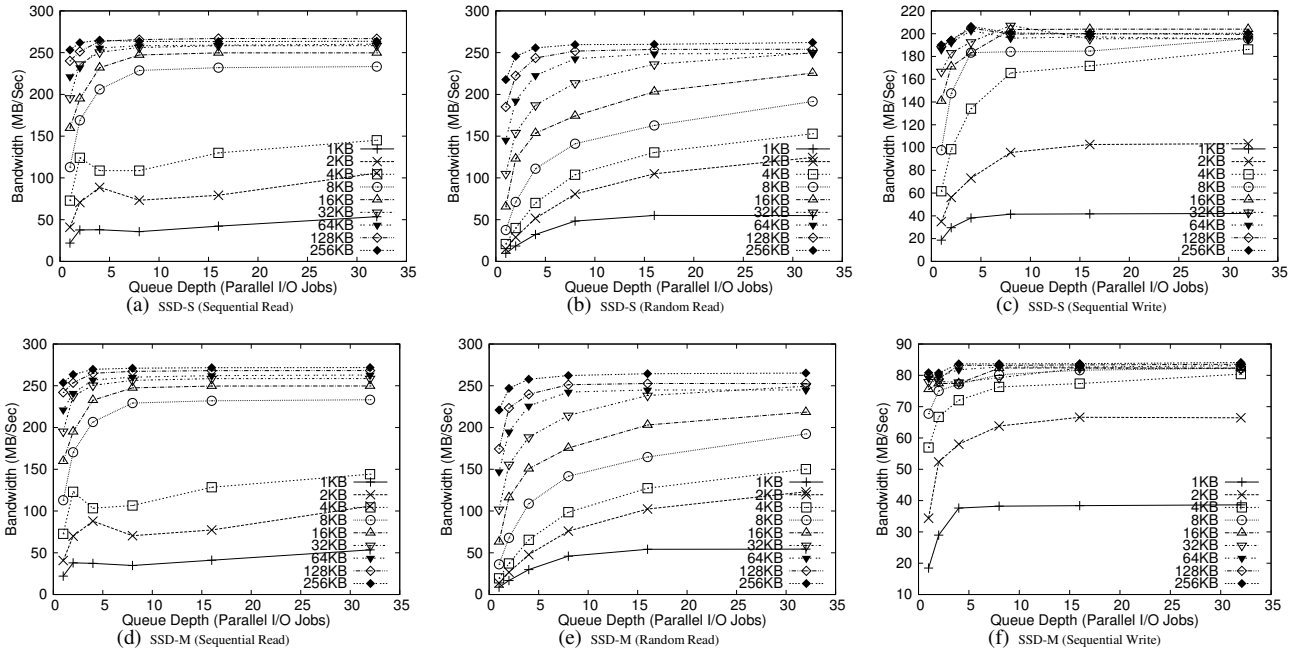



**Figure 5.** Bandwidths of the SSDs with increasing queue depth (the number of concurrent I/O jobs).

(1) *Workload access patterns determine the performance gains from parallelism.* In particular, *small and random reads* yield the most significant performance gains. 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 means 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 or even slightly better than large/sequential accesses without parallelism.* For example, with only one job, the 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/sec, which is even slightly higher than the single-job sequential reads. This indicates that SSDs provide us with an alternative approach for optimizing I/O performance, which is to parallelize small and random accesses. Many new opportunities become possible. For example, database systems traditionally favor a large page size, because hard disk drives perform well with large requests. On SSDs, which are less sensitive to access patterns, a small page size can be a sound choice for optimizing buffer pool usage [13, 20].

(3) *The redundancy of available hardware resources physically limits the performance potential of increasing I/O parallelism.* When the queue depth increases over 8-10 jobs, further increasing parallelism receives diminishing benefits. This actually reflects our finding made in Section 4 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 one job. Further parallelizing I/O jobs can bring additional but smaller benefits.

(4) *Flash memory mediums (MLC/SLC) can provide different performance potential for parallel I/O jobs, especially for writes.* 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, since the main performance bottleneck is transferring data across the serial I/O bus rather than reading the flash medium [3].

(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. 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). This surprising result is actually *opposite* to our long-existing common understanding about slow writes on SSDs.

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. 4 jobs with 4KB requests). This is related to interference in the readahead, and we will give detailed analysis in Section 5.3.

### 5.2 How do parallel reads and writes interfere with each other and cause performance degradation?

| | Seq. Write | Rnd Write | None |
|---|---|---|---|
| **Seq. Read** | 109.2 | 103.5 | 72.6 |
| **Rnd. read** | 32.8 | 33.2 | 21.3 |
| **None** | 61.4 | 59.4 | |

**Table 2.** Bandwidths (MB/sec) of co-running Reads and Writes.

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, etc. Parallel jobs accessing such resources need to be serialized. (2) Both writes and reads can generate background operations internally, such as readahead and asynchronous write-back [6]. (3) Mingled reads and writes can foil certain internal optimizations. For example, flash memory chips often provide a `cache mode` [2] to pipeline a sequence of reads or writes. A flash memory plane has two registers, *data register* and *cache register*. 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.

To show such an interference, similar to that in the previous section, we use the toolkit to generate a pair of concurrent workloads. The two workloads access data in two separate 1024MB 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 show the aggregate bandwidths (MB/sec) of two workloads co-running on SSD-S in Table 2. Co-running with 'none' means running a workload individually.

We 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 any workload running individually, which means parallelizing workloads obtains benefits, although the aggregate bandwidths cannot reach the optimal results, the sum of the bandwidths of individual 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/sec 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. Although we cannot exactly identify which specific reason causes such an effect, this case shows that we should be careful about mixing reads and writes.

### 5.3 How does I/O parallelism impact the effectiveness of readahead?

State-of-the-art SSDs implement a readahead mechanism to detect sequential data accesses and prefetch data into the on-device cache [6]. Parallelizing multiple sequential read streams would result in a sequence of mingled reads, which can interfere with the sequential-pattern-based readahead on SSDs. On the other hand, parallelizing reads can improve bandwidths. So there is a tradeoff between *increasing parallelism* and *retaining effective readahead*.

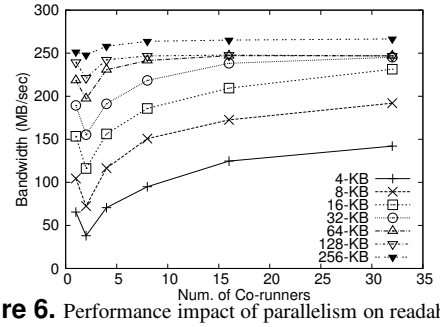



**Figure 6.** Performance impact of parallelism on readahead.

In order to examine the impact of parallelism on readahead, we generate multiple jobs, each of which sequentially reads an individual 1024MB space (i.e. the $i_{th}$ job reads the $i_{th}$ 1024MB space) simultaneously. 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 2 jobs. For example, for request size of 4KB, the bandwidth drops by **43%** (from 65MB/sec to 37MB/sec). At that point, readahead is strongly inhibited due to mingled reads, and such a negative performance impact cannot be offset at a low concurrency level (2 jobs). When we further increase the concurrency level, the benefits coming from parallelism quickly compensate for the impaired readahead. Similarly, increasing the request size can alleviate the impact of interfered readahead by increasing the aggregate bandwidth. 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 to include a more sophisticated sequential pattern detection mechanism to identify multiple sequential read streams to avoid this problem.

### 5.4 How does an ill-mapped data layout impact I/O parallelism?

The write-order-based mapping in SSDs has two advantages. First, high-latency writes can be evenly distributed across domains. This not only guarantees load balance, but also naturally balances available free flash blocks and evens out wear 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 write-order-based mapping may result in some negative effects. The largest one is that

since the data layout is completely determined by the order in which blocks are written on the fly, logical blocks can 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 into the same domain, data accesses would be congested and have to compete for the shared resources, which impairs the effectiveness of parallel data accesses.

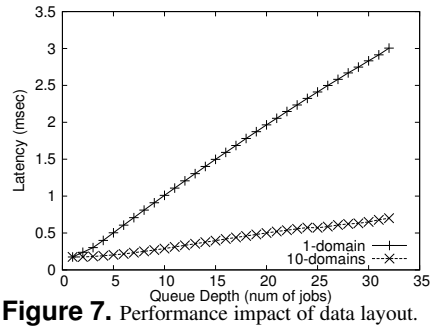



**Figure 7.** Performance impact of data layout.

To show quantitatively the impact of an ill-mapped physical data layout, we designed an experiment on SSD-S as follows. We first sequentially overwrite the first 1024MB of storage space to map the logical blocks evenly across domains. Knowing the physical data layout, we can create a trace of random reads with a 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 traces and we measure the average latencies for the two workloads. 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*. It is also worth mentioning here that we also find the ill-mapped data layout can impair the effectiveness of readahead too.

In summary, our experimental results show that I/O parallelism can indeed provide significant performance improvement, but as the same time, we should also pay attention to the surprising results that come from parallelism, which provides us with a new understanding of SSDs.

# 6 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 the internal parallelism of SSDs can not only yield significant performance improvement in large data-processing systems, but more importantly, it also provides many emerging challenges and new research opportunities.

## 6.1 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, the most important two join operators are *hash join* and *index join* [12]. Hash join sequentially fetches each *tuple* (a line of record) from the driving input relation and probes an in-memory hash table. Index join fetches each tuple from the driving input relation and starts index lookups on $B^+$-trees of a large relation. In general, hash join is dominated by sequential data accesses on a huge fact table, while index join is dominated by random accesses during index lookups.

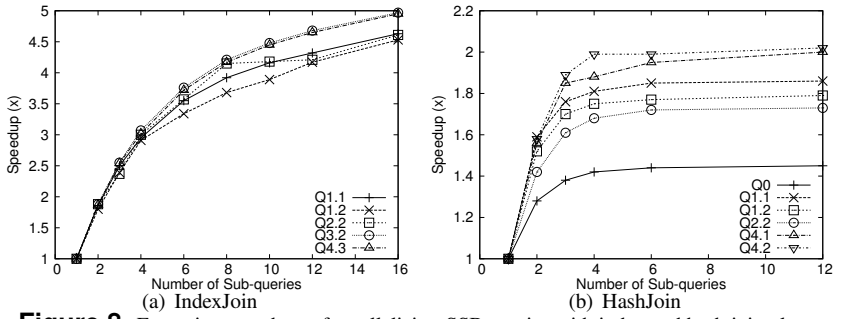
Our case studies are performed on the PostgreSQL 8.3.4. The working directory and the database are located on the SSD-S. We select Star Schema Benchmark (SSB) queries [22] (scale factor 5) as workloads. SSB workloads are considered to be more representative in simulating real warehouse workloads than TPC-H workloads, and they have been used in recent research work, e.g. [19].

## 6.2 Case 1: Parallelizing Query Execution

In this case, we study the effectiveness of parallelizing query executions on SSDs. Our query-parallelizing approach is similar to prior work [31]. Via data partitioning, a query can be segmented to multiple sub-queries, each of which contains joins on partitioned data sets and pre-aggregation. The sub-queries can be executed in parallel. The final result is obtained by applying a final aggregation over the results of sub-queries.

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 selected query Q1.1, Q1.2, Q2.2, Q3.2, and Q4.3. For hash join, besides Q1.1, Q1.2, Q2.2, Q4.1, Q4.2, we also examined a simple query (Q0), which only scans LINEORDER table with no join operation. Figure 8 shows the speedup (execution time normalized to the baseline case) of parallelizing the SSB queries with sub-queries.

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 speedup, which means the factor of 5 speedup is not due to computational parallelism. This case again shows the importance of parallelizing random accesses (e.g. $B^+$-tree lookups) on an SSD. In fact, when executing an index lookup dominated query,




(a) IndexJoin (b) HashJoin

**Figure 8.** Execution speedups of parallelizing SSB queries with index and hash join plans.

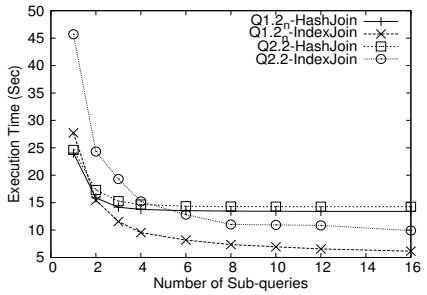



**Figure 9.** Index join vs. hash join.

the DBMS engine cannot fully utilize the SSD bandwidth, since each access is small and random. Splitting a query into multiple sub-queries 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**. Though less significant than that for index join plans, such a speedup is still impressive. (3) Parallelizing query execution with little 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 sequentially scans a big table with little computation, the SSD is kept busy and there is less room for overlapping I/O and computation, which limits further speedup through parallelism. This case clearly shows that in database systems, a typical data-processing application, I/O parallelism can provide substantial performance improvement on SSDs, especially for operations like index-tree lookups.

### 6.3 Case 2: Revisiting Query Optimizer

A critical component in DBMS is the *query optimizer*, which decides the plan and operator used for executing a query. Implicitly assuming the underlying storage device is an HDD, the optimizer estimates the execution times of a hash join plan and an index join plan, and selects an optimal plan for each query. On an HDD, 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 more complicated, since parallelism weakens the performance difference of various access patterns.

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 sub-queries. Figure 9 shows the query execution times of the two plans with 1 to 16 sub-queries. One sub-query means no parallelism.

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 sub-queries, 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. 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.

## 7 System Implications

Having presented our experimental and case studies, we present several important implications to system and application designers. We hope that our new findings can provide effective guidance and enhance our understanding of 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 paper.

**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. In our experiments, we did not observe obvious negative effects of over-parallelization, however 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 highly recommend adopting high-end SLC-based SSDs to provide more headroom for serving workloads with intensive writes.

**Random reads:** A surprising result is that with parallelism, random reads can perform comparably and sometimes even slightly better than simply sequentializing reads without parallelism. Such a counter-intuitive finding indicates that we cannot continue to assume that sequential reads are always better than random reads. Traditionally, operating systems often make trade-offs for the purpose of organizing large sequential reads. For example, the anticipatory I/O scheduler [15] intentionally pauses issuing I/O

requests for a while and anticipates to organize a larger request in the future. In SSDs, such optimization may become less effective and sometimes may be even harmful, because of the unnecessarily introduced delays. On the other hand, some application designs become more complicated. For example, it becomes more difficult for the database query optimizer to choose an optimal query plan, because simply counting the number of I/Os and using static configuration numbers can no longer satisfy the requirement for accurately estimating the relative performance strengths of different join operators with parallelism.

**Random writes:** Contrary to our common understanding, parallelized random writes can achieve high performance, sometimes even *better* than reads. Moreover, 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 – the SSDs internally handle writes in the same way, regardless of access patterns. The indication is two-fold. On one hand, we can consider how to leverage the high write performance through parallelizing writes, e.g. how to commit synchronous transaction logging in a parallel manner [8]. 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 [29] or under serious fragmentation [6], random writes are still a research issue.

**Interference between reads and writes**: 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. A related research issue is on 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 here 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 write-order-based mapping can significantly impair the effectiveness of parallelism and readahead. 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.

**Revisiting application designs**: Many applications, especially data-processing applications, are often heavily tailored to the properties of hard disk drives and are designed with many implicit assumptions. Unfortunately, these HDD-based optimizations can become sub-optimal on SSDs. This calls 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 design and bring many desirable benefits, such as improved buffer pool usage, which are highly beneficial in practice.

In essence, SSDs represent a fundamental change of storage architecture, and I/O parallelism is *the key* to exploiting the huge performance potential of such an emerging technology. More 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 even in commodity systems. Such a paradigm shift would greatly challenge the optimizations and assumptions made throughout the existing system and application designs. We believe SSDs present not only challenges but also countless new research opportunities.

## 8 Other Related Work

Flash memory based storage technology is an active research area. In the research community, flash devices have received strong interest and been extensively studied (e.g. [3, 5–7, 10, 14, 27, 30]). Due to space constraints, here we only present the work most related to internal parallelism studied in this paper.

A detailed description about the hardware internals of flash memory based SSD has been presented in [3] and [10]. An early work has compared the performance of SSDs and HDDs [26]. We have presented experimental studies on SSD performance by using non-parallel workloads, and the main purpose is to reveal the internal behavior of SSDs [6]. This work focus on parallel workloads. A benchmark tool called uFLIP is presented in [5] to assess the flash device performance. This early study has reported that increasing concurrency cannot improve performance on early generations of SSDs. Researchers have studied the advances of SSDs and reported that with NCQ support, the recent generation of SSDs can achieve high throughput [20]. Our study shows that parallelism is a critical element to improve I/O performance and must be carefully considered in system and application designs. A mechanism called *FlashLogging* [8] adopts multiple low-cost flash drives to improve log processing, and parallelizing writes is an important consideration. Our study further shows 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.

## 9 Conclusions

We have presented a comprehensive study on essential roles of exploiting internal parallelism in SSDs. We would like to conclude the paper with the following three messages, based on our analysis and new findings. First, effectively exploiting internal parallelism indeed can significantly improve I/O performance and largely remove the performance limitations of SSDs. This means that we must treat I/O parallelism as a *top priority* for optimizing I/O performance, even in commodity systems. Second, we must also pay particular attention to some potential side effects related to I/O parallelism on SSDs, such as the strong interference between reads and writes, and minimize their impact. Third, and most importantly, in the scenario of I/O parallelism, many of the existing optimizations specifically tailored to the property of HDDs can become ineffective or even harmful on SSDs, and we must revisit such designs. We hope this work can provide insights into the internal parallelism of SSD architecture and guide the application and system designers to utilize this unique merit of SSDs for achieving high-speed data processing.

## Acknowledgments

We are grateful to the anonymous reviewers for their constructive comments. We also thank our colleague Bill Bynum for reading this paper and his suggestions. This research was partially supported by the US National Science Foundation under grants CCF-0620152, CCF-072380, and CCF-0913050, and Ministry of Industry and Information Technology of China under grant 2010ZX03004-003-03.

## References

[1] Serial ATA revision 2.6. http://www.sata-io.org.

[2] Micron 8, 16, 32, 64gb SLC NAND flash memory data sheet. http://www.micron.com/, 2007.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX'08*, 2008.

[4] Blktrace. http://linux.die.net/man/8/blktrace.

[5] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR'09*, 2009.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of SIGMETRICS/Performance'09*, 2009.

[7] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proc. of FAST'11*, 2011.

[8] S. Chen. FlashLogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD'09*, 2009.

[9] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing storage arrays. In *ASPLOS'04*.

[10] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proc. of ISCA'09*, 2009.

[11] Fusion-io. ioDRIVE DUO datasheet. http://www.fusionio.com/PDFs/Fusion_ioDriveDuo_datasheet_v3.pdf, 2009.

[12] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[13] G. Graefe. The five-minute rule 20 years later, and how flash memory changes the rules. In *DaMon'07*, 2007.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of MICRO'09*, New York, NY, December 2009.

[15] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP'01*, 2001.

[16] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proc. of FAST'05*, 2005.

[17] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *FAST'10*, 2010.

[18] I. Koltsidas and S. Viglas. Flashing up the storage layer. In *Proc. of VLDB'08*, 2008.

[19] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proc. of VLDB'09*, 2009.

[20] S. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *Proc. of SIGMOD'09*, 2009.

[21] M. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[22] P. O'Neil, B. O'Neil, and X. Chen. http://www.cs.umb.edu/poneil/StarSchemaB.PDF.

[23] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. A high performance controller for NAND flash-based solid state disk (NSSD). In *NVSMW'06*, 2006.

[24] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of SIGMOD'88*, 1988.

[25] P. Perspective. OCZ Apex series 250GB solid state drive review. http://www.pcper.com/article.php?aid=661.

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

[27] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10*, 2010.

[28] Samsung Elec. Datasheet (K9LBG08U0M). 2007.

[29] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMon'09*, 2009.

[30] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10*, Jan 2010.

[31] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Query processing techniques for solid state drives. In *Proc. of SIGMOD'09*, 2009.