Research on High Performance Database Management Systems with Solid State Disks

A DISSERTATION SUBMITTED TO
THE UNIVERSITY OF TOKYO
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yongkun Wang
December 2010

Abstract

In this information explosion era, data volumes grow drastically, posing great challenge to the data-intensive applications, such as the database management systems. These data-intensive applications are required to process the huge amount of data quickly. However, in the current hard disk-based storage system, the speed gap between the CPU and hard disks becomes the bottleneck to improve the performance. At this time, the Solid State Disk (SSD) is on the spotlight. The SSD, mainly composed of flash memory, has a significant performance advantage over the traditional hard disk. The read performance of SSD is about two orders of magnitude better than that of hard disk. The sequential write performance of SSD is also much better than that of hard disk. However, the random write performance of SSD is comparable or even worse than that of hard disk, because of the "erase-before-write" design of the flash SSD. Therefore, comprehensive study is required to incorporate the flash SSDs into the existing enterprise database management systems.

In this dissertation, I performed a research on the possibility of building high performance database management systems with SSDs. Firstly I provided the basic performance study of the flash SSD. I built a micro benchmark to bypass the operating system buffer cache to get the real performance of flash SSD. With the micro benchmark, I got the performance results of flash SSD. I implemented a flash SSD measurement and simulation system. Secondly I had the performance evaluation of database system with TPC-C benchmark. The IO behavior in the TPC-C experimental system was analyzed along the IO path. Next, I described the SSD-oriented scheduling methods, confirmed the potential performance improvement by static ordering and merging IO trace, and verified the expected performance gain through online IO replaying. The evaluation of this scheduling system showed that it can significantly improve the IO performance of database system on flash SSDs. I summarized the findings, and drew a conclusion that the write defering and coalescing, address converting and aligning was very effective with little resource in the scheduling system. Therefore, the proposed SSD-oriented scheduling was effective to improve the database performance. Finally, I concluded the dissertation and described the future work.

Acknowledgements

First of all, thanks Professor Dr. Masaru Kitsuregawa for choosing me into this great lab. Prof. Kitsuregawa drew the blueprint for me about my dissertation, told me the guideline of research, and categorized the research topics for me when I was sinking in a lot of topics. In addition, Professor Kitsuregawa adviced me to play with my strong point. Professor Kitsuregawa also financially supported me for the study. I had been benefited a lot by Professor Kitsuregawa's well-founded research and fame in this area.

Thanks Associate Professor Dr. Miyuki Nakano. Prof. Nakano not only instructed me directly on my research and papers, but also handled a lot of troublesome administration things for me required by the university. As for the research, Professor Nakano's effort laid in each of my papers and experiments, the discussion had taken her a lot of time and it was always very helpful for me to go forward.

Thanks Dr. Kazuo Goda. Dr. Goda gave me the most detailed advice of each of my paper and experiments, helping to examine the results in detail and provided a lot of insightful advices to help to advance continuously.

Thanks to Professor Dr. Jun Adachi, Professor Dr. Takashi Chikayama, Professor Dr. Hitoshi Aida, and Associate Professor Dr. Masashi Toyoda for reviewing my dissertation.

Thanks to Professor Dr. Nemoto, Dr. Itoh, Dr. Yokoyama, Dr. Zhenglu Yang and other lab members and Ms. secretaries for the nice help during my study.

Thanks to my wife, Dr. Xin Li.

Thanks to my parents.

For all those who helped me along the way, this would not have been possible without you...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the information explosion era, the data volume grows drastically, posing great challenge to the data processing applications in current system, especially the system for data-intensive applications, such as the database systems. The performance of current system is mainly limited by the storage system, because the CPU speed and multi-core technology have been fully developed, while the speed of storage system, mainly composed of the rotating hard disks, has not been improved proportionally. Therefore, the speed gap between the CPU and disks limits the performance of the data-intensive applications, especially when the size of the datasets exceeds the amount of main memory, the processing of the massive dataset cannot satisfy the requirements.

With the emerging of new storage device, such as the solid state disk (SSD), the speed gap between CPU and disk is expected to be reduced. The SSDs, especially flash memory SSDs, are drawing more and more attention in the storage world. Jim Gray once said "Flash is disk, disk is tape", and in the book "The 4th Paradigm" he envisaged so-called "CyberBricks" using the SSD to act as node for the distribute computing of the massive dataset [25]. With the performance and volume increases quickly, the SSDs are being incorporated into the enterprise storage systems and expected to play a vital role.

While the SSD is viewed as a promising storage alternative for storage system, it is also recognized that the access performance of SSDs is quite different from that of the traditional rotating hard disks, and existing systems are mainly designed and tuned based on the hard disks for decades. It is very necessary to carefully examine and re-consider the existing systems in order to maximize the performance benefits of SSDs. Therefore, I performed the study on flash SSD and examined the techniques for the existing database systems on SSDs to fully utilize the performance advantage of SSDs.

1.2 Contributions

My contributions are summarized as follow:

1. I built a micro benchmark, verified the performance characteristics of flash SSD.

2. I studied the database performance on SSDs with the analysis along the IO path. I used TPC-C benchmark, several "high-end" SSDs on the market, two file systems with contrary write strategies (in-place update vs. non-in-place update), and two widely used DBMSs for the evaluation. I examined

the IO behaviors along IO path in the OS kernel and provided my analysis on the usage of flash SSDs.

3. I designed a SSD-oriented scheduling system. I employed the checkpoint information from application as IO management window to be used by my scheduling system. I built the the validation system with static ordering and merging of IO trace to verify the effectiveness of this SSD-oriented scheduling system, and implemented an online scheduling system with flexible configurations and validated the effectiveness.

1.3 Outline

The following chapters will be organized as follow: chapter 2 will give a brief introduction to the flash SSD. Chapter 3 will list the existing works in this field. chapter 4 will present my study on the basic performance of flash SSD. Chapter 5 will provide the performance evaluation of SSD-based database system using the TPC-C benchmark, as well as the IO analysis along the IO path. Chapter 6 will describe the scheduling system with various IO scheduling techniques. The evaluation of this scheduling system will be provided in chapter 7. Finally, I provided the conclusion in chapter 8.

Chapter 2

Flash SSD

2.1 Flash SSD

Flash SSD is composed of NAND flash memory. NAND flash memory is a kind of EEPROM (Electrically Erasable Programmable Read-Only Memory). There are two types of NAND flash memory; SLC and MLC. SLC flash memory provides better performance and endurance, while MLC flash memory makes large capacity available. As for the price, SLC flash memory is more expensive than MLC flash memory.

There are three operations for NAND flash memory: read, write(program), erase. The read and write operations are very fast, while the erase operation is time-consuming. Table 2.1 summarizes necessary time for each operation in a 4GB flash memory chip [54]. The data cannot be written in place. When updating the data, the entire erase-block containing the data must be erased before the updated data is written there. This "erase-before-write" design leads to the relatively poor performance of random write.

Table 2.1: Basic Performance of Flash Memory Chip[54]

Page Read to Register (4KB)	$25\mu s$
Page Write from Register (4KB)	$200\mu s$
Block Erase (256KB)	$1500\mu s$

Recently, the large capacity flash memory is starting to appear in the market. Large capacity flash memory chips are assembled together as the flash SSD (Solid State Drive), with dedicated control system, emulating the traditional block device such as hard disk. The internal structure of the flash SSD can be shown by block diagram in Figure 2.1. The flash SSD can be directly connected to the current system by the SATA interface[1]. Inside the flash SSD, the "On-board System" contains the mapping logic called Flash Translation Layer (FTL) which makes the flash SSD appear to be a block device. The "NAND Flash Memory" packages are assembled with a number of parallel flash memory buses, which are called "Channels" by some manufacturer[30].

2.2 Flash SSD Products

At the time author wrote this dissertation, the flash SSD is mainly with SATA and PCI Express (PCIe) interface. The SATA flash SSD has a good compatibility with

[1]Some flash SSDs are packed with the PCIe interface.

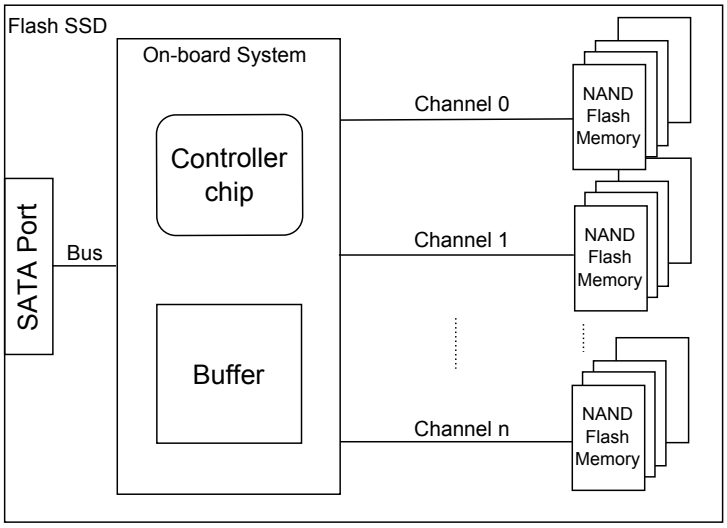


Figure 2.1: An example of internal structure of flash SSD

the existing system and is easy to use for the end user. While the PCIe flash SSD can be seen in some enterprise storage solutions [16].

The flash SSDs can also be divided into to SLC flash SSD and MLC flash SSD, depending on the type of flash memory used inside the SSD. As introduced in section 2.1 about the features of SLC and MLC flash memory, the SLC flash SSD is usually expensive, and purchased by the users with higher performance requirements. The MLC SSD is cheaper with large capacity, can be used for low-end personal computer.

Table 2.2 listed three SLC flash SSDs used in this dissertation.

2.3 Flash SSD and Database System

The performance of database system, especially the online transaction processing system (OLTP), is limited by the performance of disk-based storage system. The small and random IOs in database system is very challenging for the disk heads to seek and response quickly. Because of the innate mechanical characteristics of hard disk, the performance of the hard disk is hard to be improved so much. At this time, the flash SSD seems a good alternative for the database system because there is no moving part for the SSD. Table 2.3 provides a performance comparison between hard disk and flash SSD by the performance value disclosed in the specification. We can see that the seek time of OCZ SSD is about two order of magnitude faster than that of the hard disk.

It should be carefully considered to incorporate the flash SSD into the current

Table 2.2: Basic specifications of the SSDs used in the dissertation

Manufacture Model	SLC/MLC /RPM	Form Factor	Interface	Capacity	Cache Size	Heads/ Channels
Mtron PRO 7500 [45]	SLC	3.5”	SATA 3.0Gbps	32GB	16MB[1]	4 channels[2]
Intel X25-E[30]	SLC	2.5”	SATA 3.0Gbps	64GB	16MB[3]	10 channels [31]
OCZ VERTEX EX [49]	SLC	2.5”	SATA 3.0Gbps	120GB	64MB	Not found

[1] Reported by hdparm[24] in my test system.
[2] Estimated by the number of Flash Bus Controller (FBC) in the block diagram.
[3] Obtained by the memory chip used in the 32GB model.[1]

Table 2.3: Performance specifications of the SSDs and hard disk used in this dissertation

Manufacture Model	Sustained Rate	Performance Value
HGST HDS72107 [26]	300MB/s[1]	Seek time: 8.2ms read (typical) 9.2ms write (typical)
Mtron PRO 7500 [45]	Read: 130MB/s Write: 120MB/s	Sequential Read IOPS(4KB): 12,000 Sequential Write IOPS(4KB): 21,000 Random Read IOPS(4KB): 12,000 Random Write IOPS(4KB): 130
Intel X25-E[30]	Read: 250MB/s Write: 170 MB/s	Random Read IOPS(4KB): 35,000 Random Write IOPS(4KB): 3,300
OCZ VERTEX EX [49]	Read: 260MB/s Write: 100 MB/s	Seek Time: less than 0.1ms

[1] This bandwidth is connection bandwidth. Sustained transfer rate is not disclosed in the data sheet.

system, because the current system has been optimized based on hard disks for a long time. In order to maximize the performance benefit of flash SSD, the existing system should be carefully evaluated. There are a lot of works about optimizing the existing database system for flash SSD, as will be seen in next chapter.

2.4 Summary

The basic characteristics of flash SSDs are introduced. The flash SSD seems a promising alternative to the hard disk in the database system, however, it must be carefully evaluated due to the different characteristics.

Chapter 3

Related Work

3.1 Introduction

A lot of researchers have shown a large amount of contributions on the research of SSDs. I organized some of the existing works as follow.

3.2 SSD

3.2.1 Flash Translation Layer

Flash Translation Layer (FTL) bridges the operating system and flash memory. The main function of FTL is mapping the logical blocks to the physical flash data units, emulating flash memory to be a block device like hard disk. Early FTL used a simple page-to-page mapping[34] with a log-structured architecture[53]. It required a lot of space to store the mapping table. The block mapping scheme was proposed in order to reduce the space for mapping table. The scheme introduced the block mapping table with page offset to map the logical pages to flash pages[5]. However, the block-copy may happen frequently. To solve this problem, Kim improved the block mapping scheme to the hybrid scheme by using a log block mapping table[37]. More works are shown in [51][15][36].

3.2.2 Evaluation

Agrawal et al.[3] studied the internal design trade-offs that will have impact on the performance. A comprehensive evaluation of the flash devices can be seen in uFLIP[9][7].

[20] studied the overhead of SSD in existing system by Direct IO. They used the random read request (random write is an opposite but identical operation) to measure the overhead. The overhead was measured in response time and CPU clocks of each phase along the IO path. Compared to the service time of SSD, the overhead of the platform was still small. So that authors acclaimed that the current platform with SATA interface was sufficient for the SSD. Excluding the overhead of SSD, the major overhead of the rest functions along the IO path came from the generic OS/Device functions (e.g. Driver, Interrupts, Context-switching), not from the storage specific functions such as SCSI or ATA processing.

[52] studied the performance of Intel X25-E flash SSDs in the RAID system. The experiment results showed that the performance of SSD in a RAID (0,5,10) system did not scales well; rather, the random IOPS performance of the RAID0 system degraded more than 30% than that of the single SSD. However, the random IOPS performance scaled well with the number of RAID controllers. Moreover,

author found that the software RAID was better than the hardware RAID. All this confirmed that the RAID controllers were the performance bottleneck.

[44] provided performance evaluation of several flash SSDs, specially the evaluation of three Enterprise Class PCIe SLC flash SSDs: Virident tachIOn 400GB PCIe-x8, FusionIO ioDrive Duo 2x 160GB PCIe-x4, Texas Memory Systems RamSan-20 450GB PCIe-x4. The performance of of PCIe SLC flash SSDs was significant compared to the SATA MLC flash SSDs. The comparison showed that no device outperformed others consistently in all the cases, that is, none of these device was one size fit all device. The device should be carefully evaluated by the customer's IO pattern.

3.3 File Systems

Most of the file systems for flash memory exploited the design of Log-structured file system[53] to overcome the write latency caused by the slow erase operations. JFFS2[32] is a journaling file system for flash with wear-leveling. YAFFS[63] is a flash file system for embedded devices. DFS[33] provided a file system design on flash storage layer instead of the FTL layer. The DFS was designed to bypass the traditional file system buffer and perform direct access to the SSDs via the flash storage layer. Author argued that the common FTL based interface, combined with the traditional file system, would have much overhead. Because the traditional file system was designed for hard disk system, the buffer and access pattern were not optimized for flash SSD. Since flash SSD has much fast access performance, the file system buffer and access pattern should be re-considered. Therefore, they advocated to build the file system on *flash storage layer* instead of the FTL layer. Based on the flash storage layer, the newly designed file system, Direct File System (DFS) could perform the direct access. Several applications were evaluated on DFS with the comparison to EXT3.

3.4 Database Systems

3.4.1 Embedded Systems

Early design for database system on flash memory mainly focused on the embedded systems. FlashDB[48] was a self-tuning database system optimized for sensor networks, with two modes; disk mode for infrequent write and log mode for frequent write. LGeDBMS[35], was a relational database system for mobile phone.

3.4.2 Large Database Systems

As flash SSDs are being used in enterprise storage platforms, many researchers are focusing on the performance of the flash SSDs instead of the raw flash memory. A good summary about the usage of flash for DBMS can be seen in [4], in which the usage of flash for DBMS had been categorized into three contexts: (1) as a log device for transaction processing system with all data residing in memory, (2) as the main storage media for transaction processing, (3) as an update cache for the data warehouse applications. Authors provided the techniques and evaluation for each context. FlashLogging[14] gave further information about (1).

Some work with log-structured file system for database system is as following. Myers[46] had a study on the usage of flash SSD in database systems and provided the IO throughput comparison between the LFS and the conventional file system. The evaluation of database systems on SSD using TPC-C benchmark with log-structured file system can be found in [60] and [61]. Comparison of flash SSD and hard disk-based Raid-0 system with TPC-C can be seen in [42].

[47] studied migrating the server storages to SSDs, and showed that it was not a cost-efficient solution at that time. For large enterprise database, the SSD may not be cost-efficient to store the whole database even when author wrote this dissertation. Therefore, using the SSD as a cache tier to the hard disk is considered. To use flash SSD as a cache for the large storage system seems to be a trend for large system providers, such as the Oracle Exadata Version 2 [50]. [11] used the DB2 snapshot utility to gather the IO information, then decided which objects should be put into the “limited” SSD. Another data placement scheme for SSD was proposed in [12]; unlike previous solution, it cached region-based data instead of page-level or object-level data[11]. Different regions on disk were logically divided by the access frequency and marked with different “regional temperatures”. The SSD behaved as a write-through cache. When reading, read the SSD firstly. When updating, update the pages on SSD (if cached) and HDD together. This design performed random writes to the SSD at page granularity, because author assumed that there were no noticeable performance difference between the random writes and sequential writes on *high-end* SSD such as FusionIO. Holloway [27] proposed the DD which combined the SSD with HDD to buffer the random write. The elapsed time of DD was compared with that of NILFS with a read-intensive workload. A reversed solution, using the hard disk as the write cache to extend the SSD life time, can be seen in [55]. Besides extending the SSD lifetime, it could also reduce the IO latency as acclaimed.

A block level optimization, Page-Differential Logging (PDL)[39] proposed to accumulate the page differentials in buffer then writes to flash memory once. The logical page may be updated many times before it was written to flash. When a logical page was requested to write to flash memory, PDL will firstly read the

base page from flash memory, then got the differentials by comparing the logical page and the base page, only wrote the differentials to device. When the page was read from flash memory again, the base page and the differential page were read out together to reconstruct the logical page. Prior work, in-Page Logging[41], proposed to co-locate a data page and its log records in the same physical location.

As for the utilization of the high read performance of SSD, [59] studied the algorithms in query processing and provided a join algorithm, FlashJoin, with the TPC-H evaluation.

3.4.3 Index

Lazy-Adaptive Tree[2] used buffers for nodes to host the updates and write to flash devices in batch. [11] proposed the FD-Tree, which using a small B+-tree (head tree) to hold many random writes within a small space, because on flash SSDs the performance of random writes on a small address space was close to that of sequential writes. When the head tree was full, the data were merged into the below levels of the head tree in batch, hereby the random writes were converted into sequential write by merging in batch. An B-tree implementation on FTL can be found in [62] .

3.4.4 Key-Value Store

[17] used flash SSD as cache between main memory and disk for the specific application, key-value store. The in-memory hash table used the compact key signature to save the space and keep more keys in memory. Two applications were chosen to be evaluated: online multi-player gaming and storage de-duplication. In multi-player gaming, the states of the player was stored as key-value store and accessed frequently. The storage de-duplication was a techniques to remove the redundancy of backup system. Files in the backup systems were divided into chunks, and hash signature algorithms were used to determine whether two chunks were identical. The main data structure was hash.

3.5 Enterprise Systems

Gordon[13], was a powerful cluster built with flash memory for data-intensive applications. Oracle used PCIe flash memory inside their enterprise storage solutions[16], and showed great performance improvement by the TPC-C benchmark[57]. EMC incorporated the flash drives into their storage solutions to boost the performance for data-intensive applications [19][18].

Chapter 4

Basic Performance of Flash SSDs

4.1 Introduction

The flash SSD has the capability of disk emulation, but its internal mechanism is different of that of conventional hard disk (HDD). The performance characteristics should be carefully studied when considering about the deployment into the current systems. In this section, I presented my experimental examination with three major SSD products.

Some literatures have disclosed the basic performance of several flash memory SSDs [9][46]. In this section, I developed the micro benchmark, validated some general features of flash SSDs reported by the existing work. I built a measurement system, which was flexible to not only get the performance of the specific flash SSD products, but replay and measure the real workload trace.

4.2 Experimental Environment

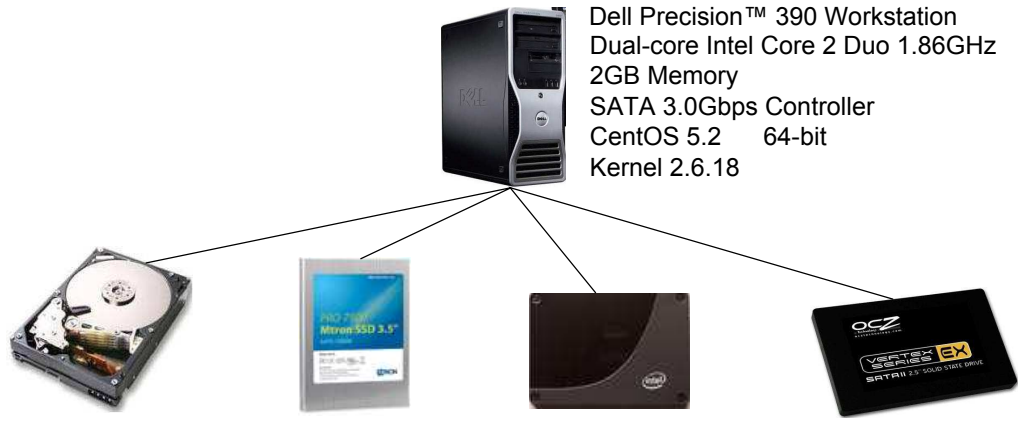


Figure 4.1: Experimental setup

In this section, I presented my basic performance study on three commercialized flash SSDs. Figure 4.1 illustrates my test system with three flash SSDs and a hard disk. In chapter 2, Table 2.2 has provided some basic information from the specification of these high-end (relatively fast and reliable with SLC flash memory chips inside) flash SSDs from the major product lines of Mtron, Intel and OCZ.

I developed a micro benchmark tool running on the Linux system. The benchmark tool has the capability of automatically measuring overall IO performance by issuing several types of IO sequences (e.g. purely sequential reads and 50%

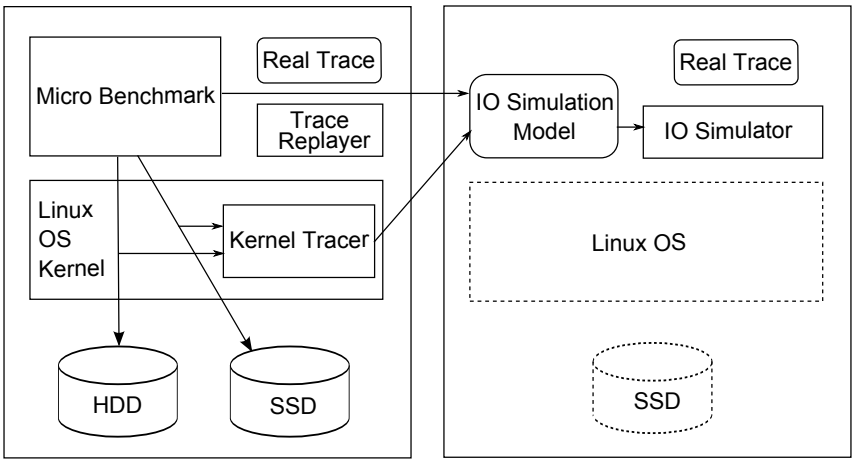


Figure 4.2: Measurement System

random reads plus 50% random writes) to a target SSD. The benchmark tool bypassed the file system and the operating system buffer hereby clarifying the pure performance of the given SSDs. The effects of file systems and operating system buffers will be discussed in later sections. As for the caching inside the SSDs, the SSDs allow me to change their controller configurations, but I employed default settings in all the experiments: read-ahead prefetching and write-back caching were enabled, because the vendors are supposed to ship their products with reasonable configurations.

The entire measurement system is illustrated in Figure 4.2. I built the micro benchmark to get the raw IO performance by bypassing the OS kernel. At the same time, the kernel tracer can obtained the IO trace and replay it for the validation. The real trace can also be replayed by the trace replayer. The micro benchmark results can be used to generate the IO Simulation Model, which enables the IO simulator to replay the real trace.

4.3 Experimental Results

4.3.1 IO Throughput

I firstly examined the IO throughput of sequential accesses. Three cases are compared for each device in Figure 4.3 to 4.6. Higher throughputs were confirmed in most of the cases in the SSDs than the hard disk, but several exceptions were also seen. In Figure 4.4, the read and write throughputs of Mtron's SSD were close to each other and saturated at around 120MB/s, which is consistent with the bandwidth specifications in Table 2.3. In Figure 4.5, the read and write throughputs of

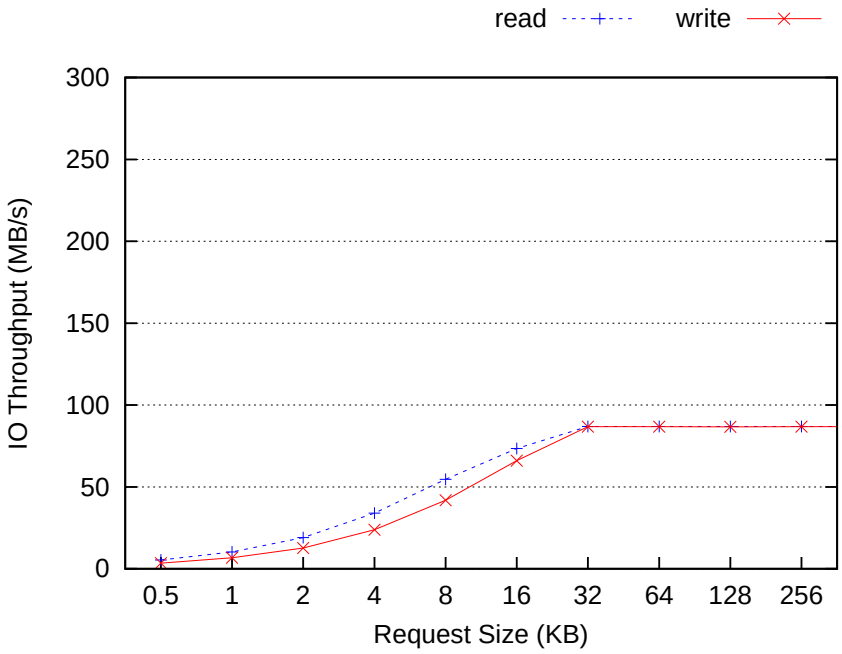


Figure 4.3: IO Throughput for Sequential Access: HGST

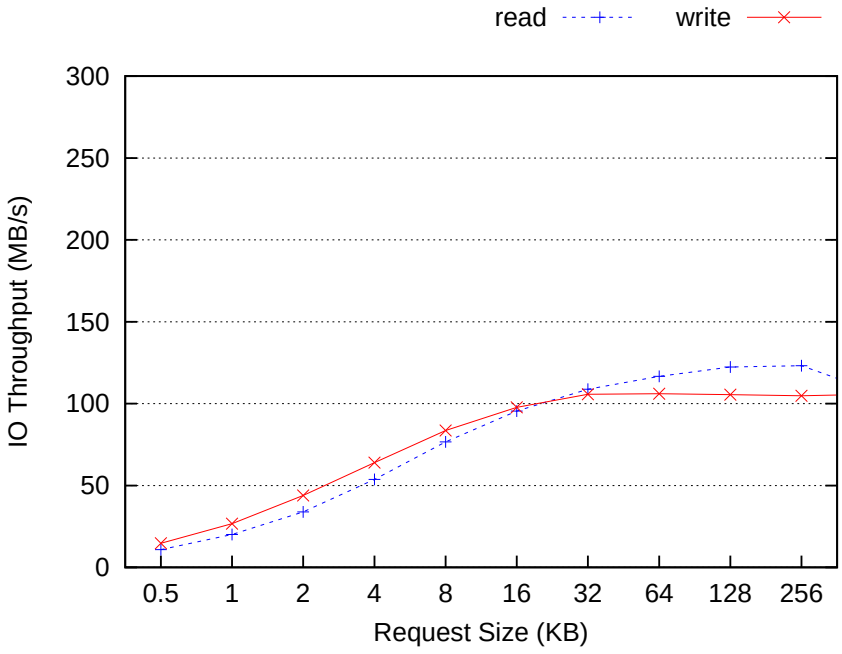


Figure 4.4: IO Throughput for Sequential Access: Mtron

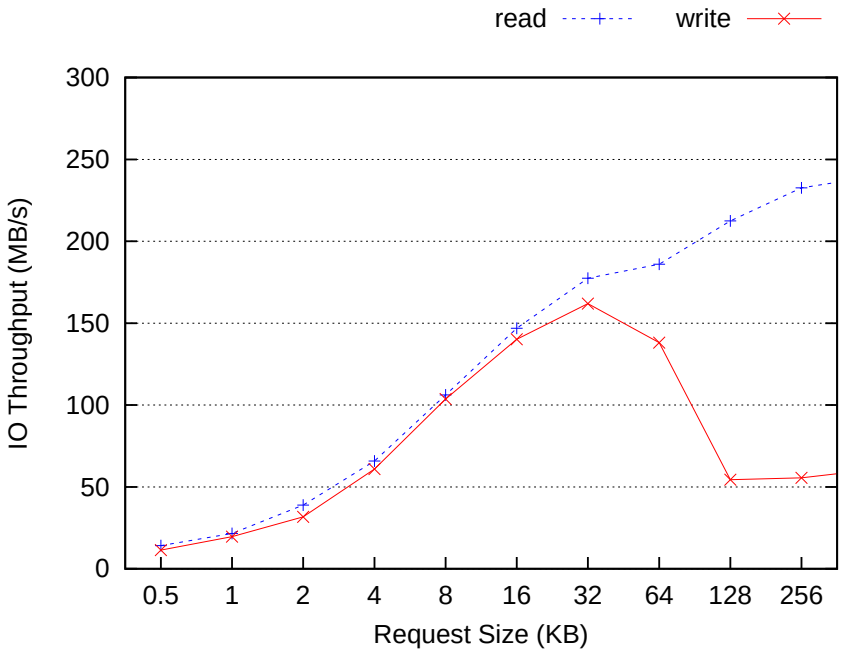


Figure 4.5: IO Throughput for Sequential Access: Intel

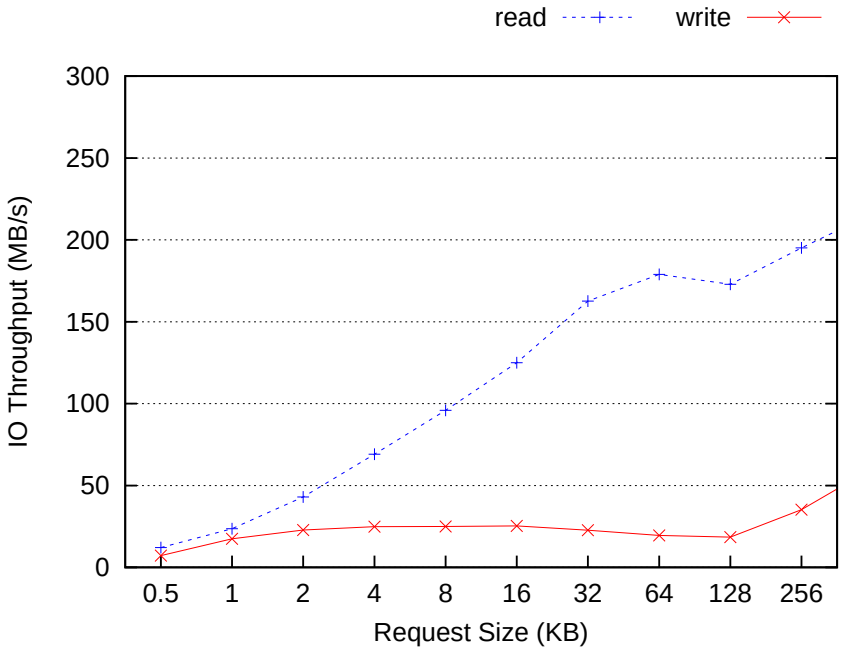


Figure 4.6: IO Throughput for Sequential Access: OCZ

Intel's SSD were much higher than that of Mtron's SSD, but the write throughput decreased when request size became larger than 32KB. The acclaimed bandwidth in Table 2.3 was also confirmed in Figure 4.5 when the request size was set to 1MB[1]. In Figure 4.6, the read throughput of OCZ's SSD was higher too, but the write throughput was the worst. The confirmed bandwidth here was lower than that indicated in the Table 2.3.

Random accesses were next studied. Figure 4.7 to 4.10 shows the throughputs that were observed for random accesses when then number of outstanding IOs was set to one. The read throughput kept untouched as that of sequential throughput and much higher than that of the hard disk, while the write and the mixed-access throughputs were drastically degraded on Mtron's SSD and OCZ's SSD. In the case of mixed access, reads and writes were respectively given 50% probability. I had expected that the mixed-access throughput would fall between the read throughput and the write throughput, but the observation was that the mixed-access throughput was comparable with the write throughput, as denoted by "mix(50r50w)" in Figure 4.8 to 4.10. Similar observations are also confirmed by other researchers and this characteristic is sometimes called *bathtub effect*[21]. Only on Intel's SSD, the write and mixed-access throughputs were clearly better than that on hard disk. Section 4.3.3 will present more results.

The same experiment was conducted with 30 outstanding IOs. The results are shown in Figure 4.11 to 4.14. The read throughput improved clearly on the hard disk, Intel's SSD and OCZ's SSD, while significant improvement was not confirmed for the read throughput in Mtron's SSD and the write and mixed-access throughputs in all the SSDs.

I had expected that some clear relation could be seen between the performance of flash SSDs and the internal design information disclosed by the manufacturers. Let us look back at the vendor-disclosed design information such as cache size and channel number cited in Table 2.2. Unfortunately I could not find strong relationship between such design information and the experimental results above presented. It is clear that the overall performance is also impacted by more other design factors, yet vendors only disclosed limited design information. Figure 4.4 and 4.5 shows that Intel's SSD has relatively high transfer rate than Mtron's SSD. This might be caused by design difference of internal channels; Intel's SSD holds ten channels whereas Mtron's SSD has only four. However, additional information such as the bandwidth of internal channels, namely MB/s, is not disclosed. Further analysis is not possible for me due to the lack of information. Experiment presented in this section can be seen as an alternative way for me to understand the basic performance characteristics.

[1] I do not show it here for the brevity.

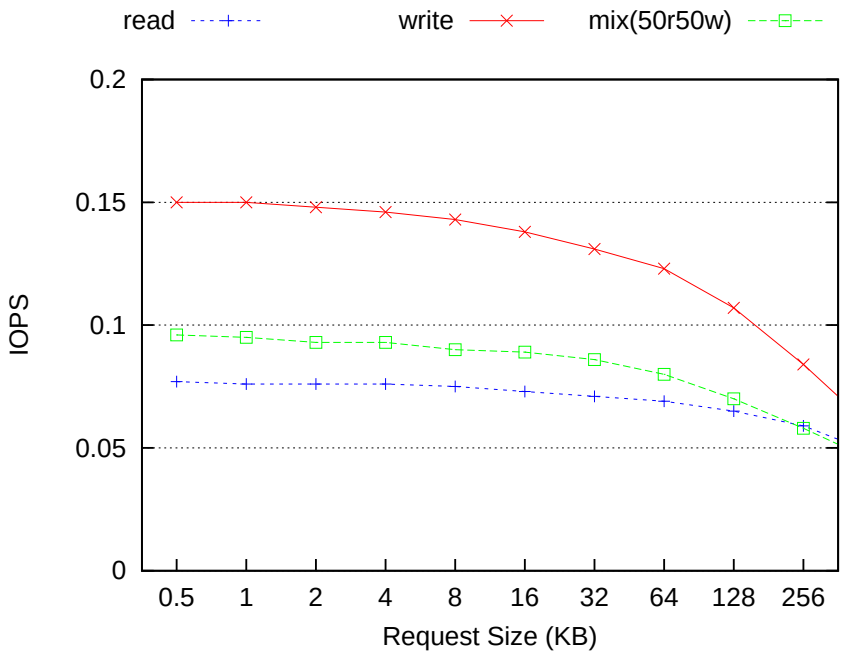


Figure 4.7: IO Throughput for Random Access (Single Thread): HGST

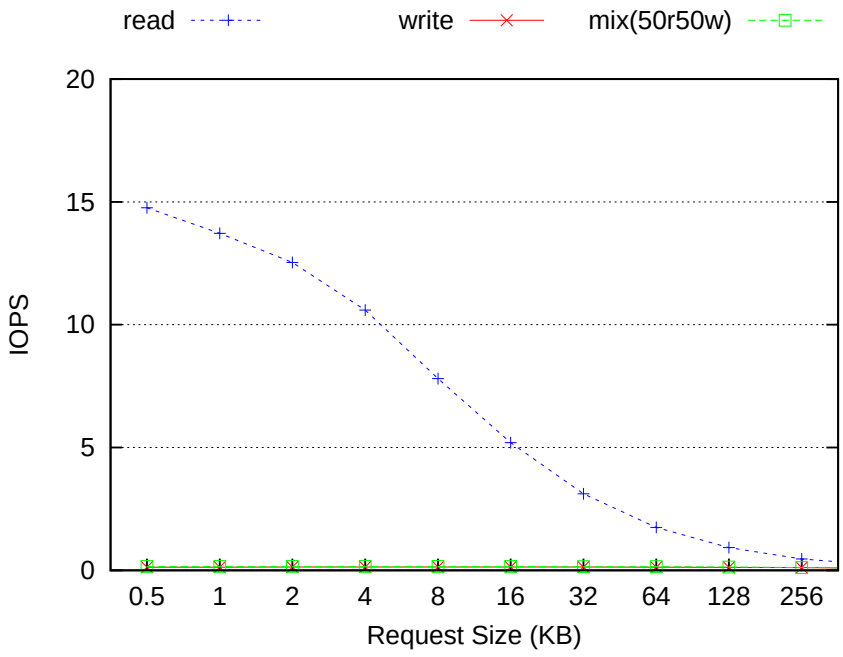


Figure 4.8: IO Throughput for Random Access (Single Thread): Mtron

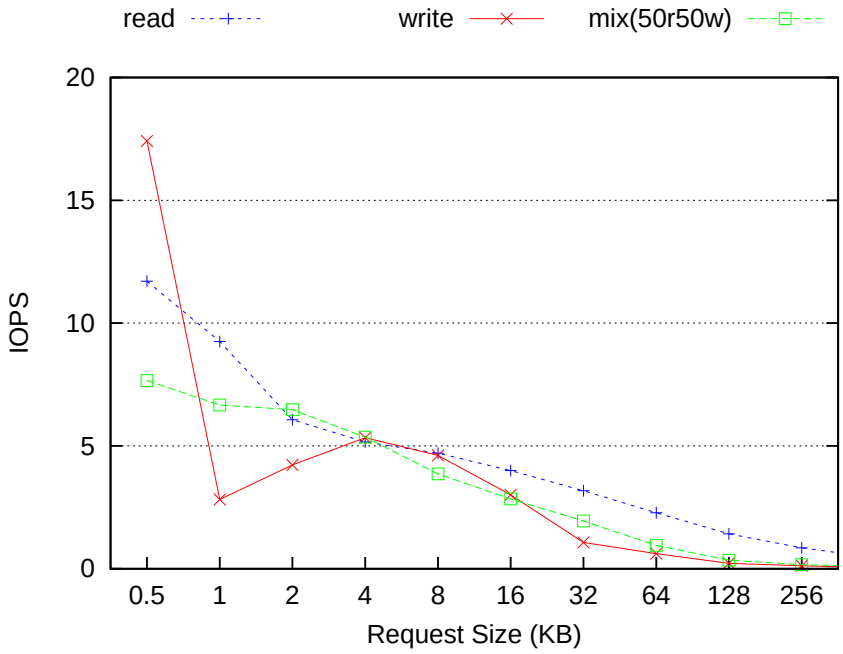


Figure 4.9: IO Throughput for Random Access (Single Thread): Intel

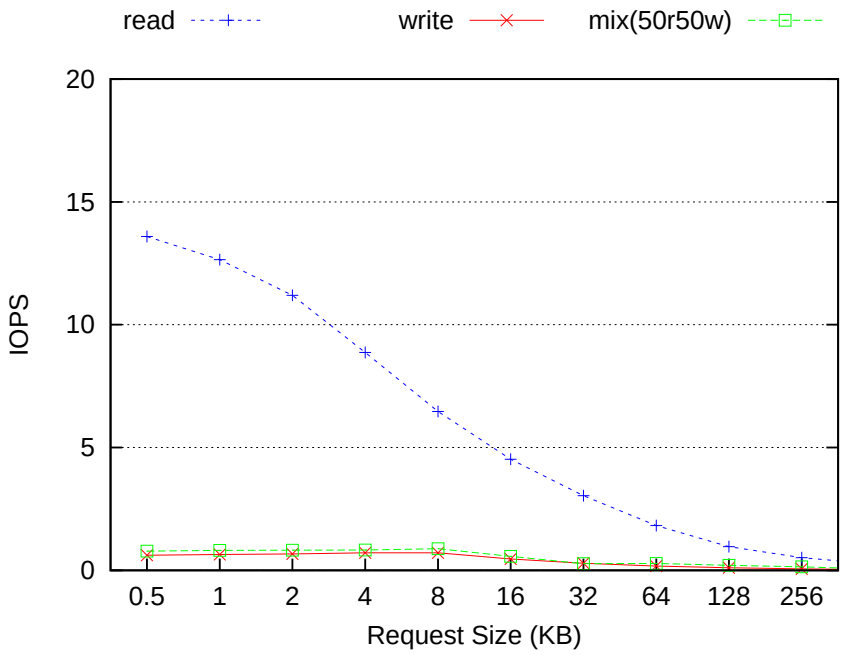


Figure 4.10: IO Throughput for Random Access (Single Thread): OCZ

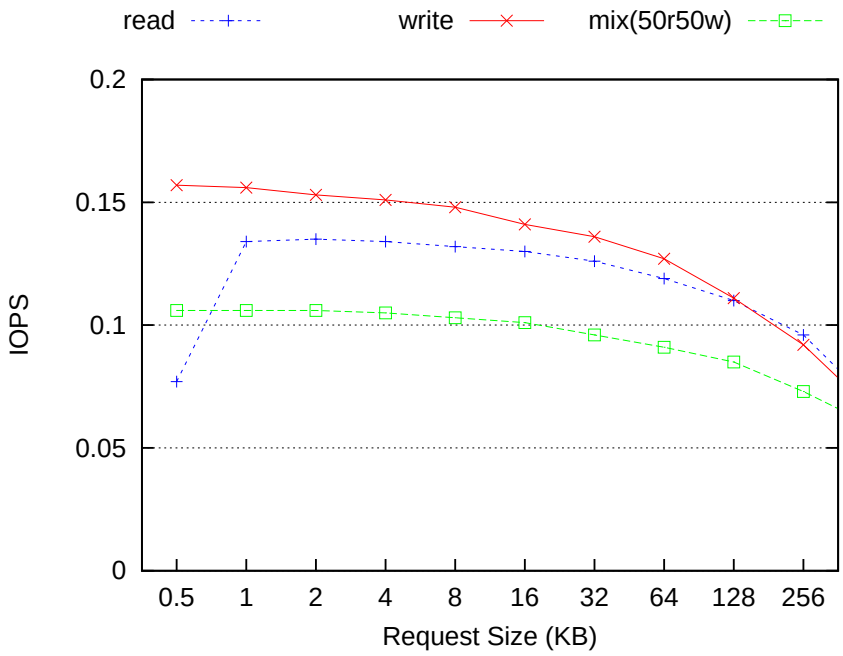


Figure 4.11: IO Throughput for Random Access (Thirty Threads): HGST

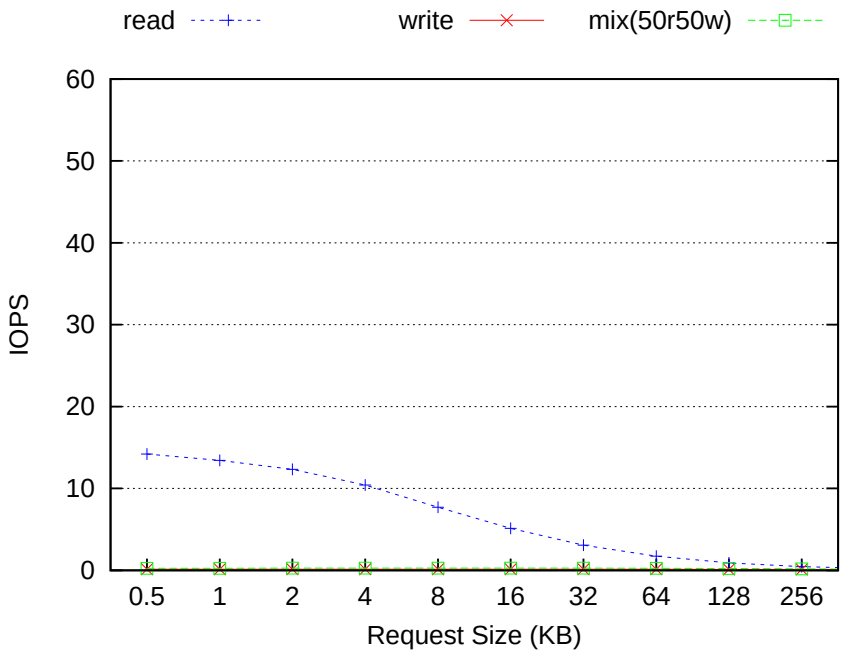


Figure 4.12: IO Throughput for Random Access (Thirty Threads): Mtron

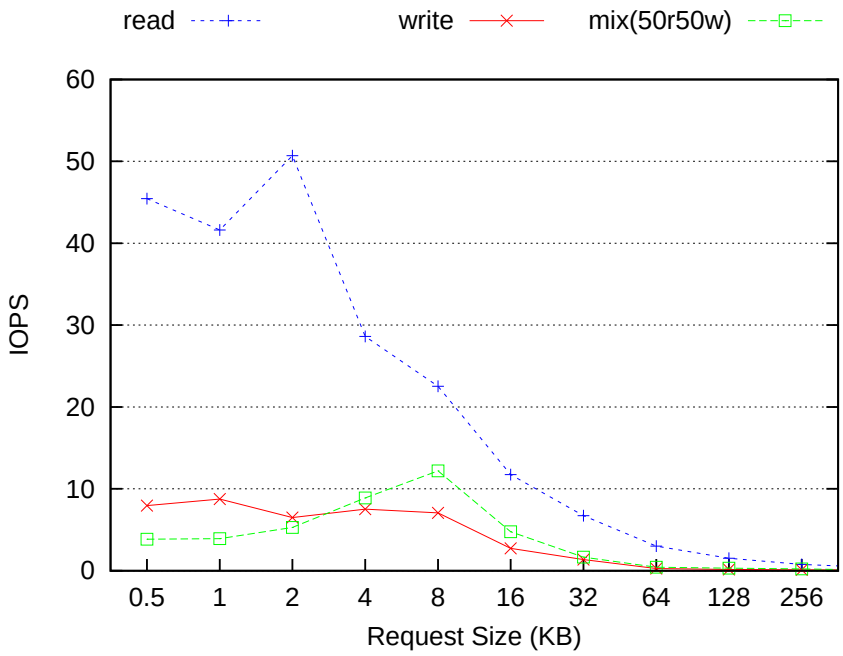


Figure 4.13: IO Throughput for Random Access (Thirty Threads): Intel

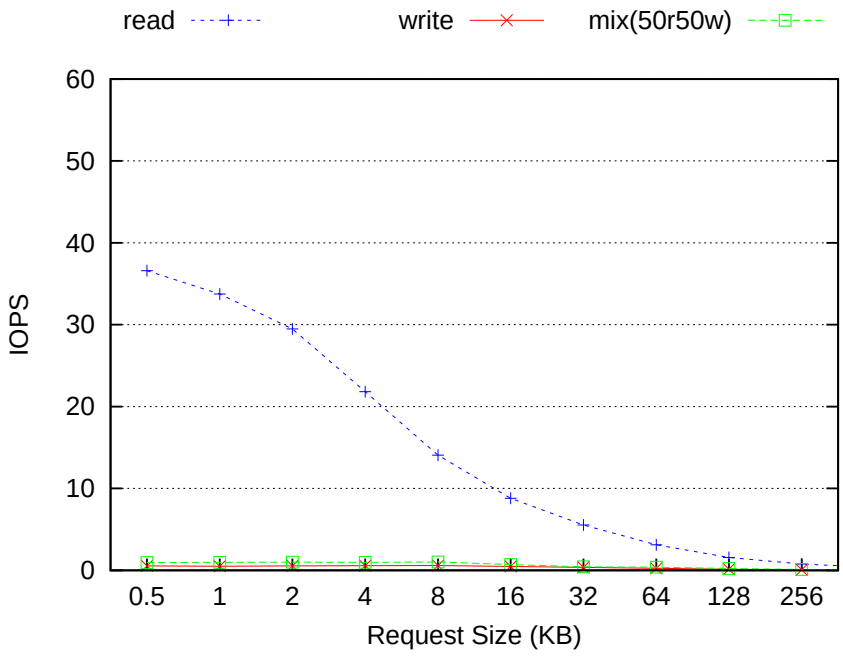


Figure 4.14: IO Throughput for Random Access (Thirty Threads): OCZ

4.3.2 IO Response Time

Since the random access performance is important to typical database applications such as the transaction processing systems, I further studied the response time. The response time distribution are shown in Figure 4.15 to 4.18. I have the following observations:

Random Read The random read response time was close to each other among three SSDs. Note that I saw a single sharp cliff in each cumulative frequency curve for the SSDs. This means that most of random reads could complete in a very small range of response times. Such small variance in response times was not confirmed in conventional hard disk, where the rotational platter always gives unpredictability of response times.

Random Write Compared with random read, random write gave more complicated characteristics. Two major cliffs were confirmed around 100 microseconds and 10,000 microseconds respectively in Mtron's SSD. Although the inside logic is not documented, my conjecture is that the long response time is caused by the inside flush operations. When the on-disk buffer is full, the control system will flush some pages and make room for the new requests. The flush operation is very time-consuming since it may incur the erase operations. Similarly, multiple cliffs were also confirmed in Intel's SSD and OCZ's SSD.

Random 50% Read 50% Write This pattern (denoted by "mix(50r50w)" in the figures) is close to *Random Write*. Although the pure read performance was very high and almost predictable, the write performance and the mixed-access performance were sometimes much poor and its variance was significant.

It is clearly shown in Figure 4.16 that the random write performance of Mtron SSD is different from the rest SSDs (two cliffs in the write curve in Figure 4.16), so I further presented the detailed IO response time information of Figure 4.16, as shown in Figure 4.19, which shows each response time of the 4KB random write request on Mtron SSD. Two belts of data values are clearly shown in Figure 4.19. Most of the values in the below belts are smaller than $100\mu s$. In the upper belts, data points are scattered between $10000\mu s$ and $30000\mu s$, representing a very long response time. Since the inside logic is not documented, my conjecture is that the long response time may be caused by the flush operation in the "On-board System" described in Figure 2.1. When the "Buffer" is full, the "On-board System" will flush some pages and make room for the new requests. The flush operation is very time-consuming since it may involve many internal activities which may incur the "erase" operations.

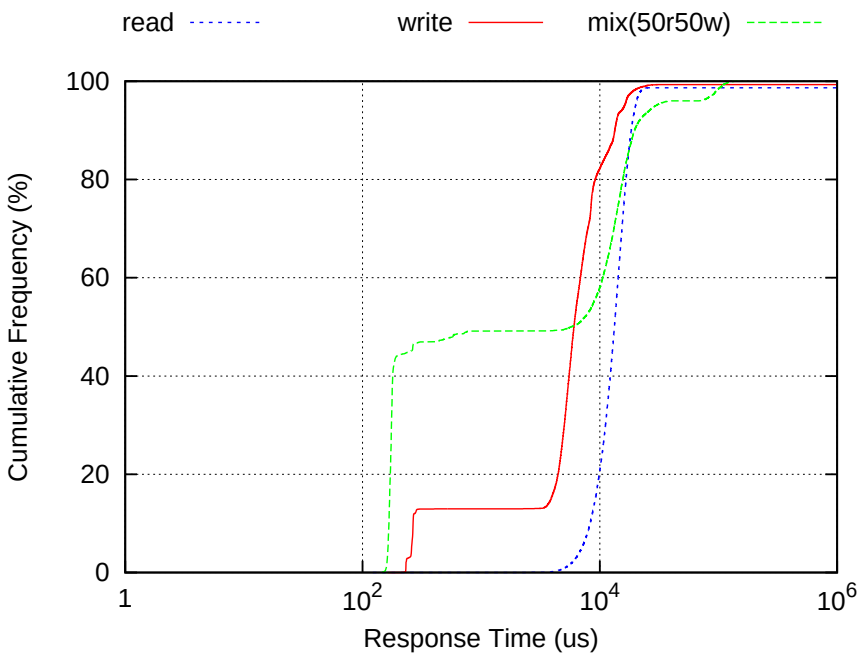


Figure 4.15: IO Response Time Distribution for Random Access (Single Thread): HGST

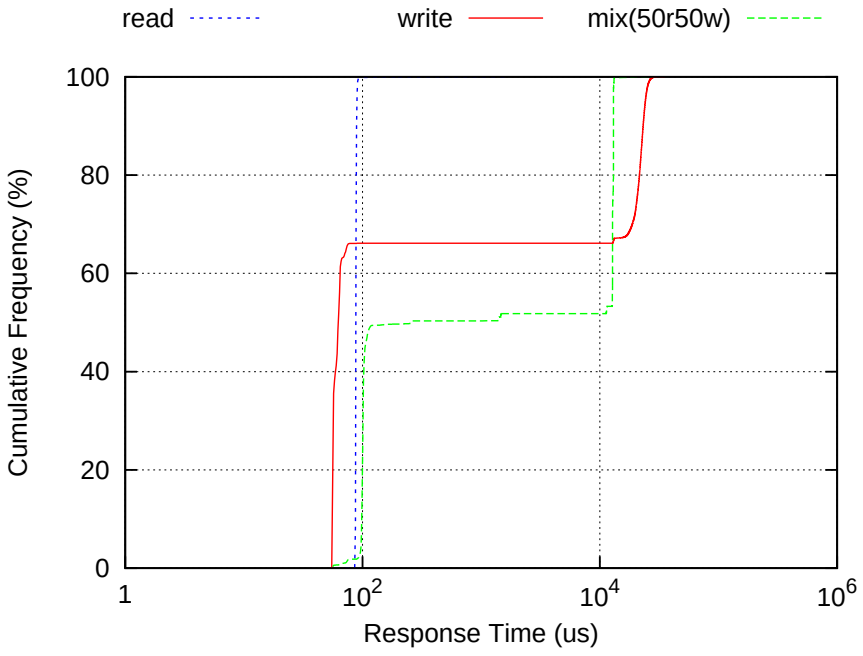


Figure 4.16: IO Response Time Distribution for Random Access (Single Thread): Mtron

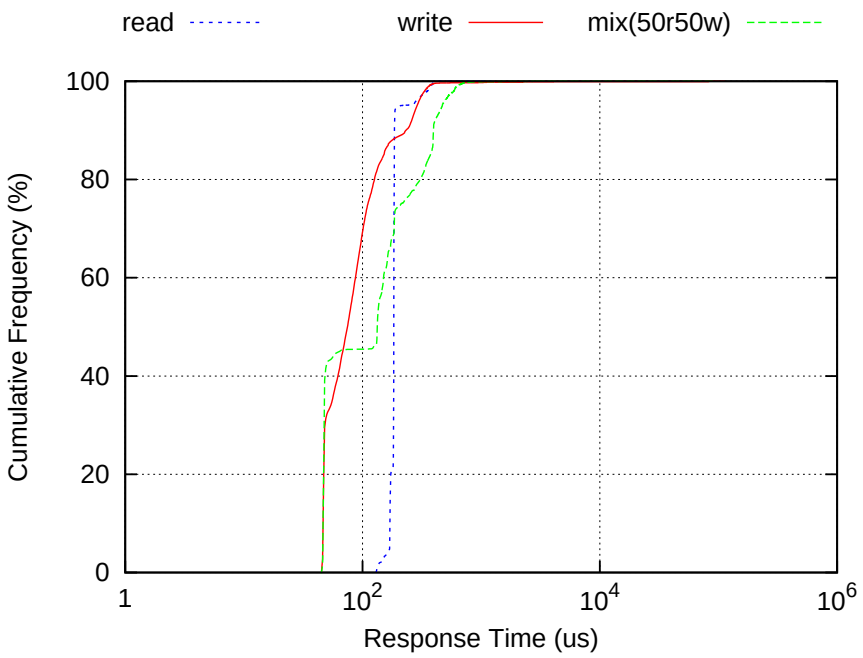


Figure 4.17: IO Response Time Distribution for Random Access (Single Thread): Intel

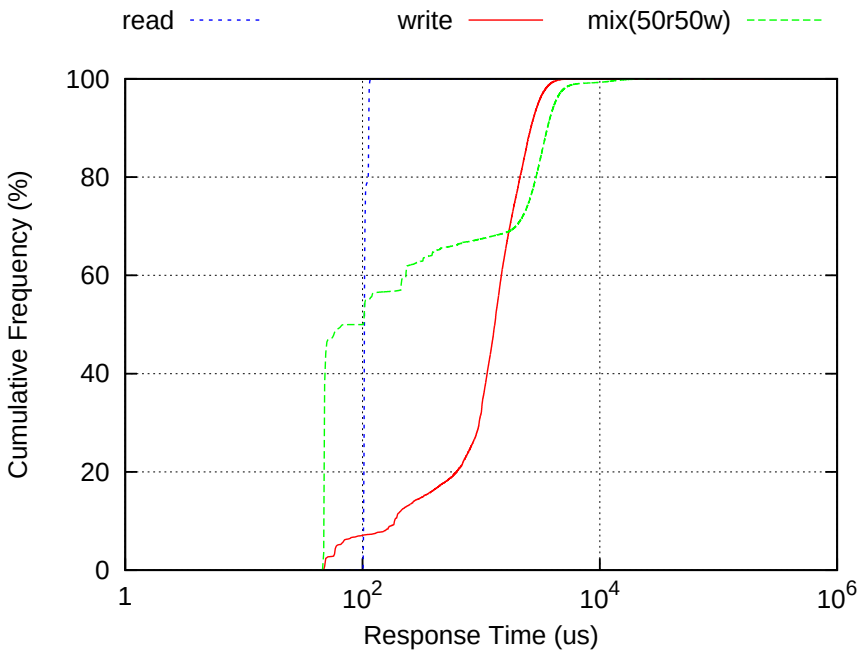


Figure 4.18: IO Response Time Distribution for Random Access (Single Thread): OCZ

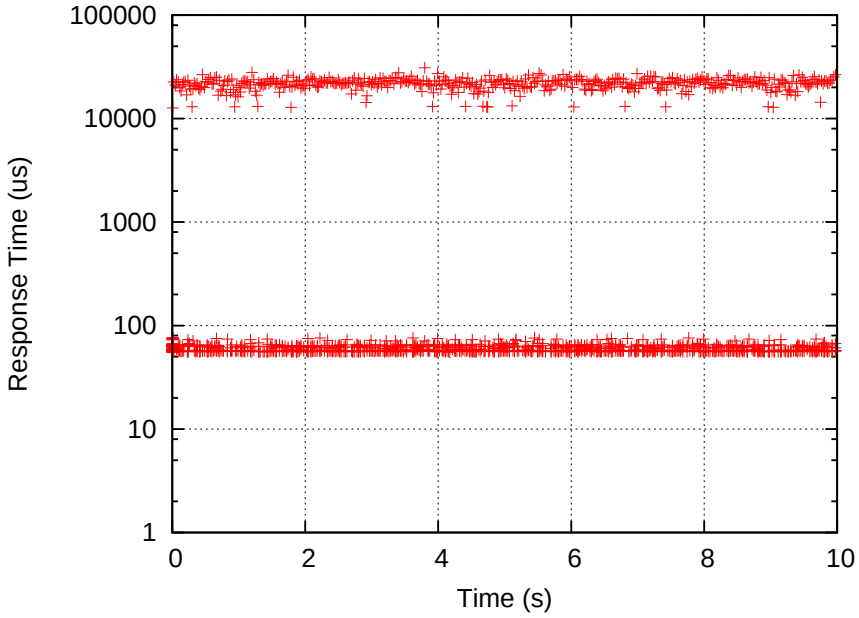


Figure 4.19: IO behavior of random write access on Mtron SSD(4KB Request Size)

4.3.3 Bathtub Effect

Bathtub effect[21] indicates the performance characteristics of mixed access patterns of reads and writes. In order to understand the performance characteristics of the mixed access patterns, I performed the experiments by varying the percentage of read requests. The percentage of read requests was varied from 0% to 100% with the 10% increase. The results are shown in Figure 4.20 to 4.31, whose x-axis shows the read percentage.

As shown in Figure 4.20 to 4.22, the overall IO throughput (read+write) is much better at either side of the pure sequential read and pure sequential write, and the throughput is the worse in the middle of the x-axis with 50% reads and 50% writes.

Figure 4.23 to 4.25 further provides the average response time in the mixed sequential access pattern. It shows that the read performance becomes much worse in the mixed access pattern compared to the pure read case (read percentage 100%). It also shows that the write performance was getting better slightly when increasing the percentage of reads.

Figure 4.26 to 4.28 shows the random IO throughput with mixed access pattern, the performance is better at either side except the Intel's SSD. Figure 4.29 to 4.31 further discloses the average response time of each access pattern. The

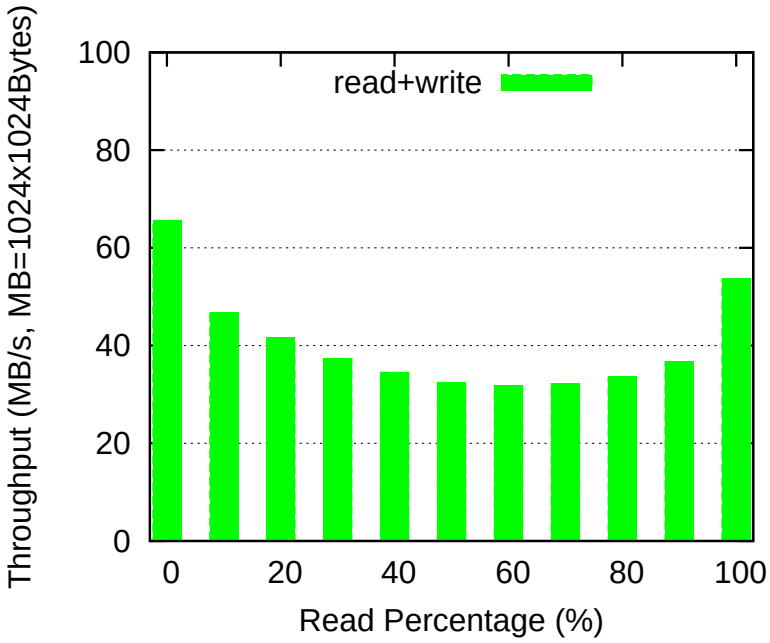


Figure 4.20: IO Throughput of Mixed Sequential Access Pattern: Mtron

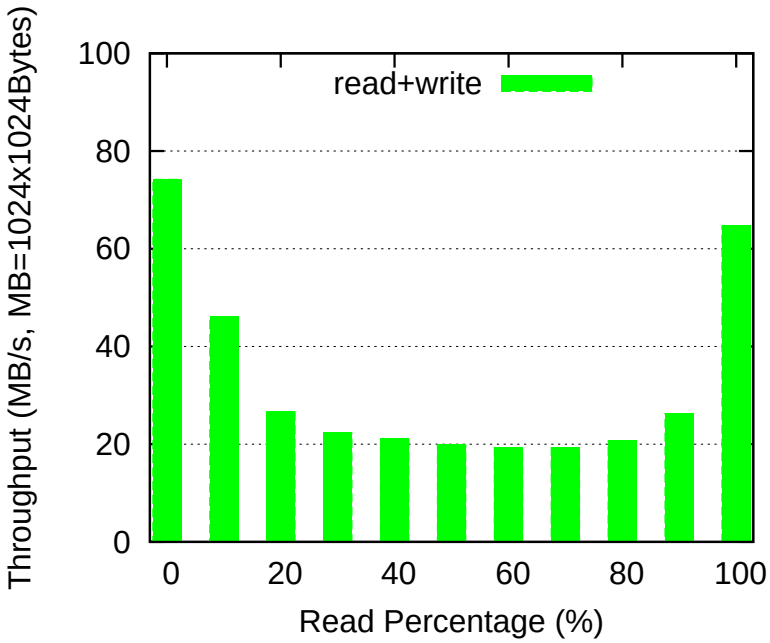


Figure 4.21: IO Throughput of Mixed Sequential Access Pattern: Intel

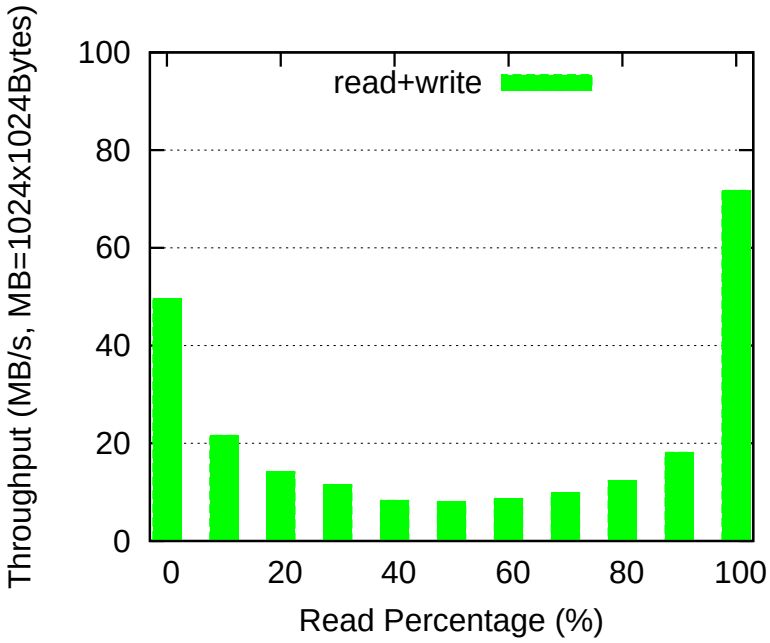


Figure 4.22: IO Throughput of Mixed Sequential Access Pattern: OCZ

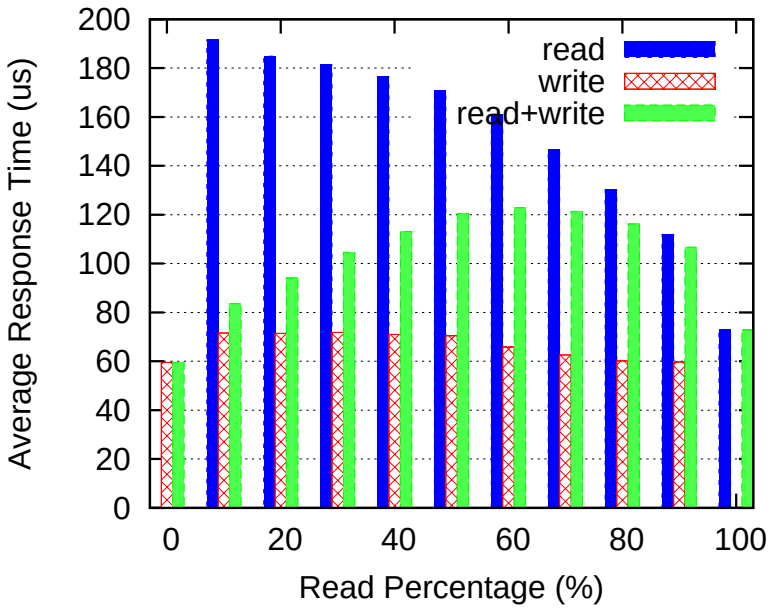


Figure 4.23: IO Response Time of Mixed Sequential Access Pattern: Mtron

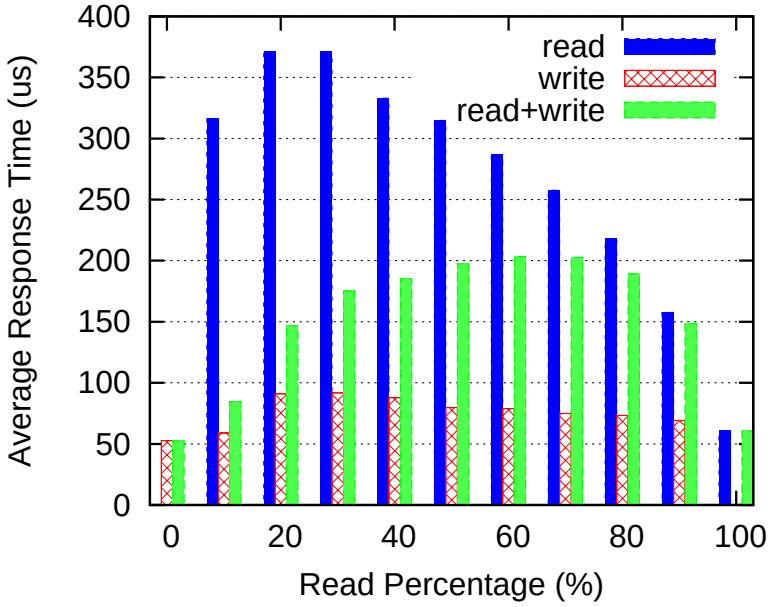


Figure 4.24: IO Response Time of Mixed Sequential Access Pattern: Intel

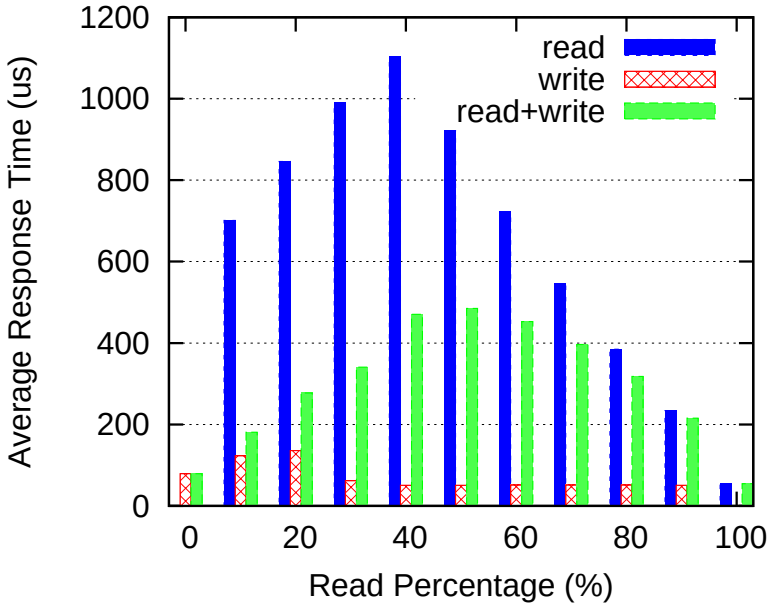


Figure 4.25: IO Response Time of Mixed Sequential Access Pattern: OCZ

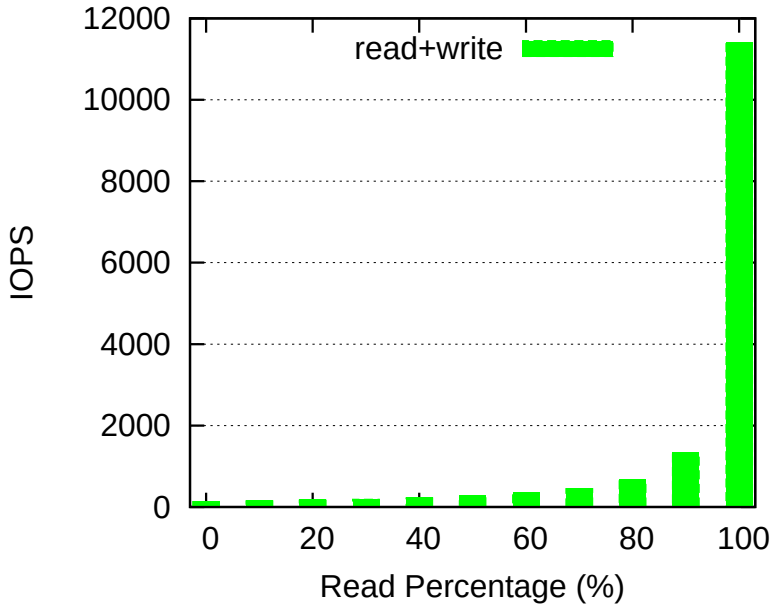


Figure 4.26: IO Throughput of Mixed Random Access Pattern: Mtron

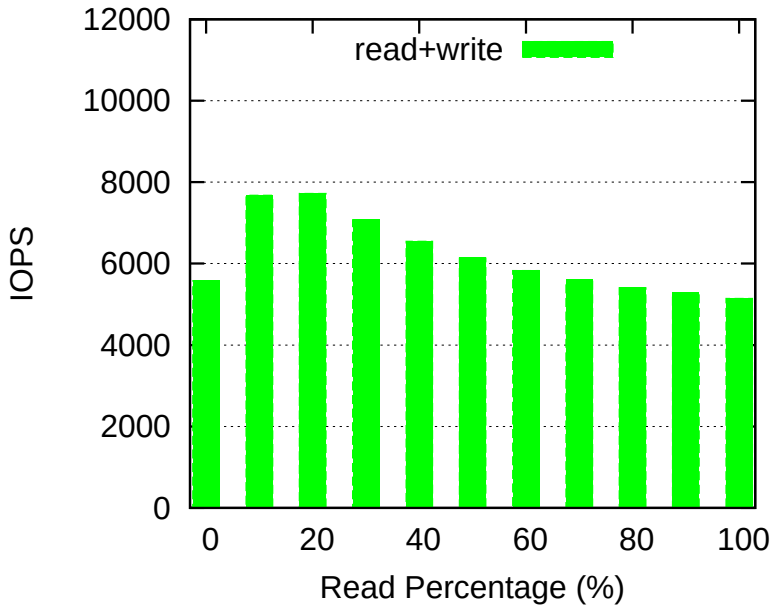


Figure 4.27: IO Throughput of Mixed Random Access Pattern: Intel

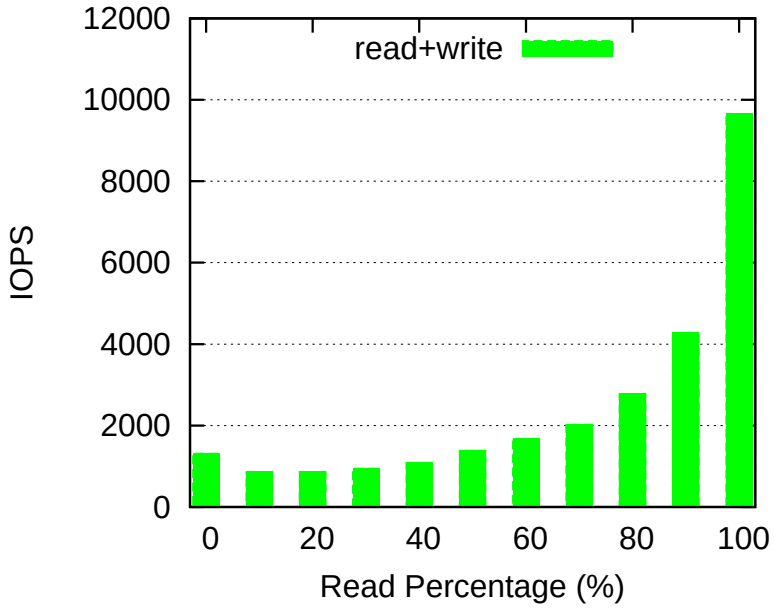


Figure 4.28: IO Throughput of Mixed Random Access Pattern: OCZ

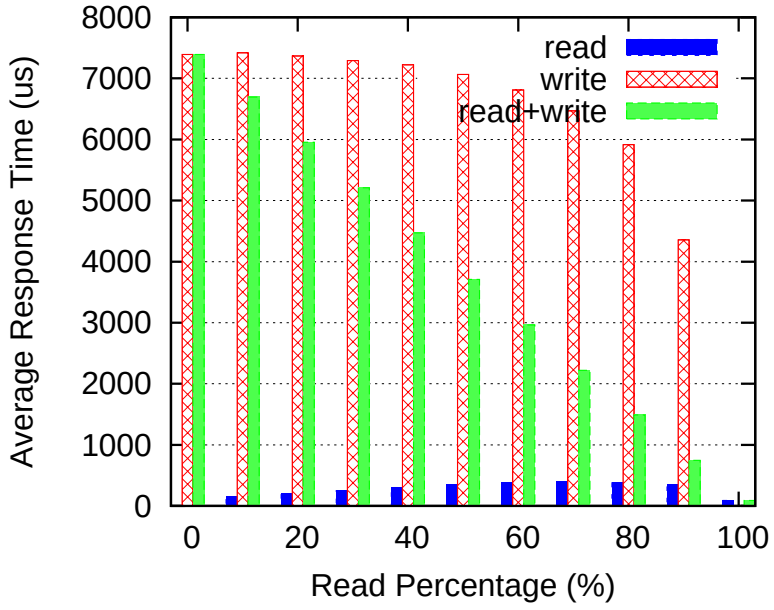


Figure 4.29: IO Response Time of Mixed Random Access Pattern: Mtron

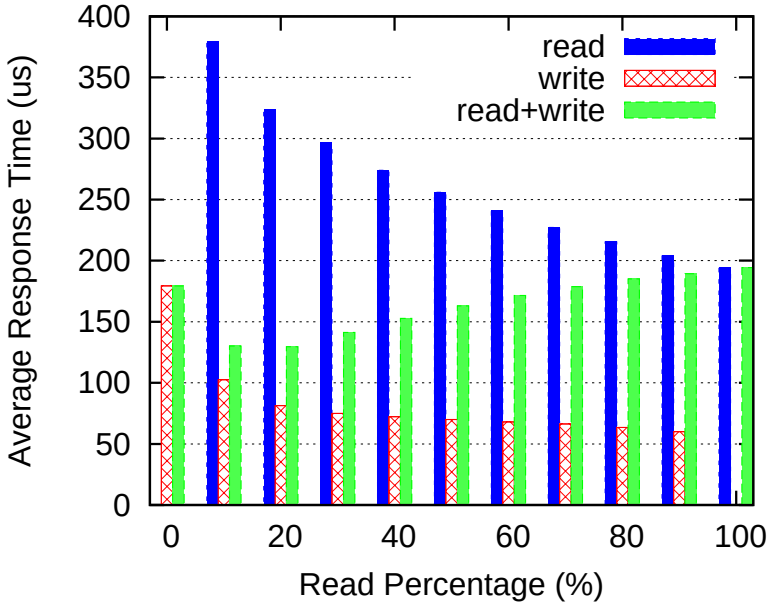


Figure 4.30: IO Response Time of Mixed Random Access Pattern: Intel

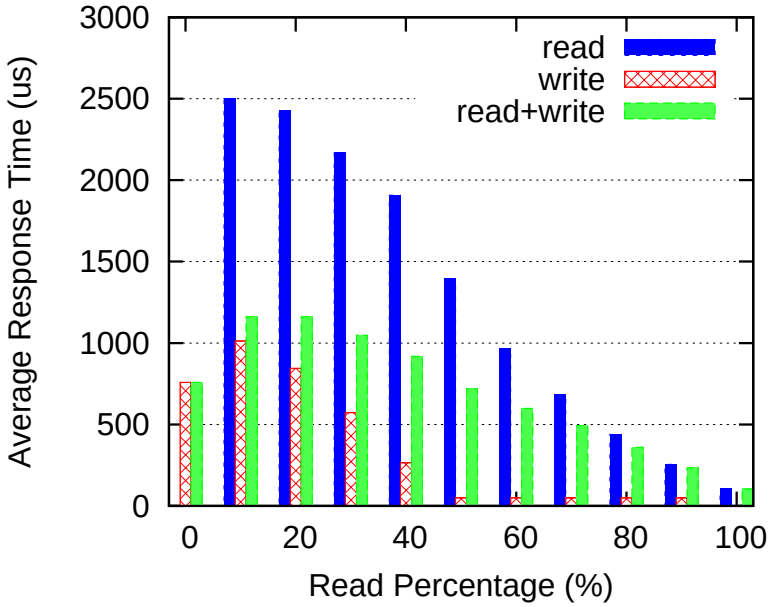


Figure 4.31: IO Response Time of Mixed Random Access Pattern: OCZ

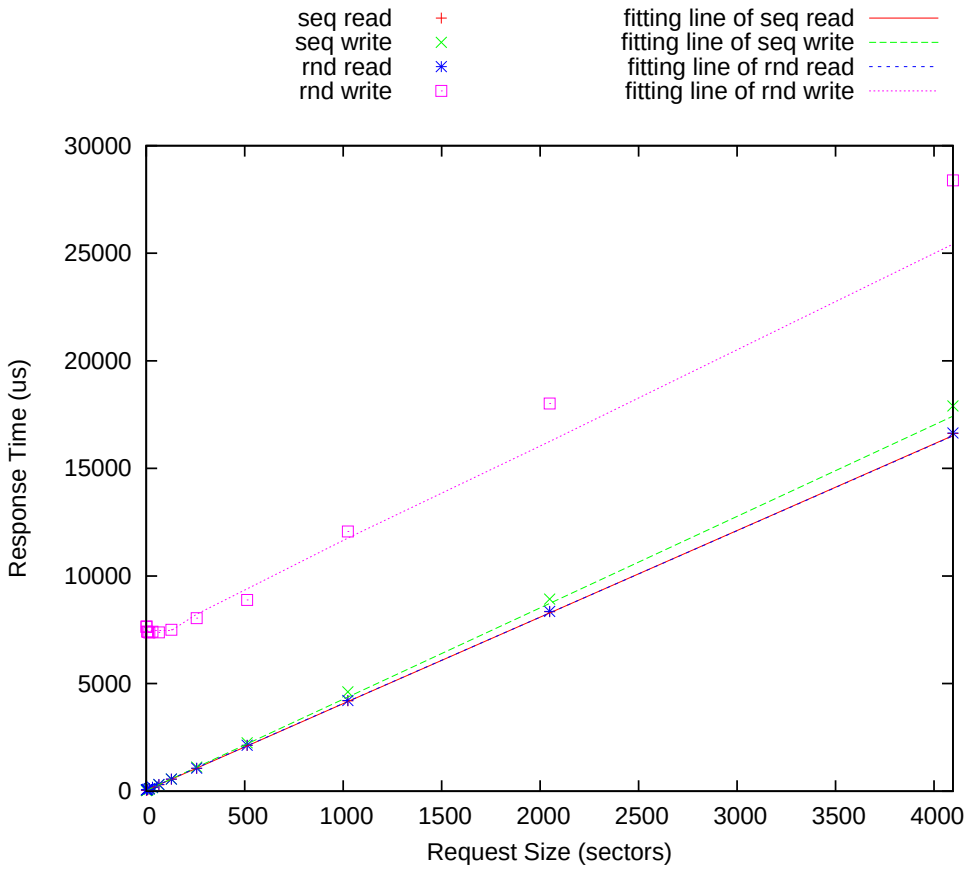


Figure 4.32: Micro benchmark results of Mtron SSD and the fitting lines

read performance also gets worse in the mixed access pattern, and the write performance gets better when increasing the read percentage.

4.3.4 Performance Equations

I will discuss the naive performance equations generated by micro benchmark results, similar work can be seen in [43]. There are several advanced flash SSD simulators, such as [38][3].

The request size and response time can be plotted into the x-y plane, as shown in Figure 4.32, which shows the response time is increasing proportionally with the increase of request size. I designed a naive fitting line equation, $y = ax + b$, to approximately represent the relation between the request size (x) and the response time (y). The coefficient a can be explained as some transmission overhead which is proportionally to the request size, and the constant b is explained as some innate overhead which is constant with the specific SSD. The fitting line equations were

calculated and given by the equation (4.1)(4.2)(4.3)(4.4), then the equations were plotted on Figure 4.32. As shown in Figure 4.32, it is clear that the fitting equation lines fit well for sequential read, sequential write, and random read. The random write is hard to fit. These equations were used for the estimation of writes flushing time required by checkpoint flushing overhead in the online scheduling discussed in chapter 6.

$$SequentialRead : y = 4.0247x + 41.3\mu s, x \geq 1sector \tag{4.1}$$

$$RandomRead : y = 4.0160x + 58.0\mu s, x \geq 1sector \tag{4.2}$$

$$SequentialWrite : y = 4.2495x + 26.4\mu s, x \geq 1sector \tag{4.3}$$

$$RandomWrite : y = \begin{cases} 7298.8\mu s, & 0 < x \leq 128sector \\ 4.6076x + 7043.9\mu s, & 128 < x \leq 1024sector \\ 4.4768x + 7083.6\mu s, & x > 1024sector \end{cases} \tag{4.4}$$

4.4 Summary

In order to have further optimization based on the characteristics of flash SSD, I developed a micro benchmark system. The performance is discussed by the results of my micro benchmark system. A naive performance model is generated based on the micro benchmark results, which will be used for the scheduling system in chapter 6.

Chapter 5

Performance Analysis of Flash SSDs Using TPC-C Benchmark

5.1 Introduction

The speed gap between the CPU and hard disk-based storage system is a bottleneck of the performance of the disk-based high performance database system. Using the new storage media such as flash SSD may be a good attempt to reduce the gap. However, direct deployment of flash SSD may not maximize the performance benefit, because the current system has been optimized based on hard disks for a long time. That is, the stack of the current storage systems have been well tuned for decades based on the characteristics of the hard disks. With the opposite characteristics such as no moving parts, "erase-before-write", flash SSDs may not be fully exploited in the existing storage systems by direct deployment.

To better utilize the flash SSD within the current system, we need to have a comprehensive understanding of the IO behaviors along the IO path for the current systems on flash SSD. The IO path has been designed chronically along the development of hard disks for decades, to hide the seek latency and utilize the sequential bandwidth. The softwares at different layers of the OS kernel, controllers and devices, separate the IO path into many passages. At each passage, the IO requests will be processed by the software layer in its own way. The whole system has been studied and adjusted by the researchers and developers for a long time to provide a good performance with a large number of hard disks. On the use of the flash SSD, the study of the IO activities at different locations is also very necessary to provide the optimizations by the characteristics of flash SSD, or balance some hard disk-based design.

In this chapter, I will try to evaluate the performance of flash SSDs in the database system with TPC-C benchmark, providing the IO analysis along the IO path.

5.2 Transaction Processing and TPC-C Benchmark

The high performance database systems are usually composed of the the simultaneous execution of multiple types of transactions that span a breadth of complexity. In such a database system, the disk input and output is significant with the contention on data access and update. The complex OLTP application environments can be viewed as the representative of such database systems. Multiple on-line terminal sessions of OLTP application will generate intensive and non-uniform data access. The response time is critical, which posing the challenging to the application execution time, in which the IO time of the underneath storage system is usually the main part.

The design of storage system is usually elaborate for high performance database system such as OLTP application. The high performance OLTP appli-

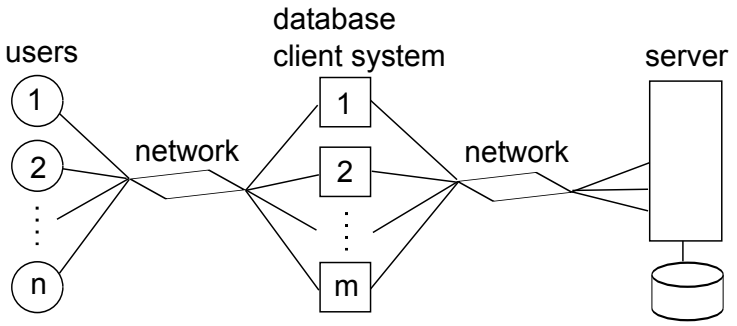


Figure 5.1: TPC-C System Architecture

cations can adopt the Direct Attached Storage (DAS) to shorten the IO path and utilize the high throughput of the bus between the server and storage systems. With the great improvement on the throughput of the network, the Network Attached Storage (NAS) and Storage Area Network (SAN) appear in the market. In either DAS or network-based NAS and SAN, the storage resources are often virtualized through the IO path and then presented to database systems. Various functions are incorporated into the IO path and they are managed far from database systems. This approach is widely accepted in the industry for mitigate the system complexity.

In order to simplify the system for analysis, I will simply separate the system into several layers, in a top-down manner: OLTP application, database application, operating system (OS), and storage devices.

I adopted the three-tier implementation of the TPC-C benchmark, as shown in Figure 5.1. The virtual users connect to the database clients via the network, and the database clients connect to the database server by the network. The storage system connects to the database server directly.

The virtual users will pick up one of the five transactions randomly and issue the selected transactions to the DBMS, as shown in Figure 5.2. Various transactions are issued to the DBMS in parallel, and these transactions are processed by DBMS concurrently with intensive data access to the storage system, and hereby having a big challenge to the performance of the storage system.

Figure 5.3 illustrates the software stacks of the TPC-C benchmark system. In Figure 5.3, I use the TPC-C [58] to represent the OLTP application. TPC Benchmark C(TPC-C) is an OLTP workload. Although TPC-C cannot reflect the entire range of OLTP requirements[28], the performance results of TPC-C can provide important reference for the design of database system, thus it is accepted by main hardware and software venders of the database systems. The workload of TPC-C is composed of read-only and update-intensive transactions that simulate the activities in OLTP application environments. Therefore, the disk input and output is

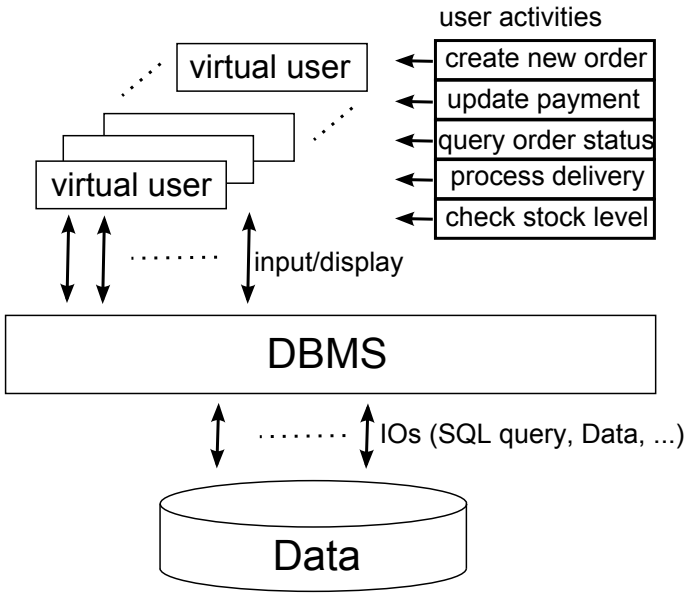


Figure 5.2: TPC-C Transaction Processing

very significant, and hereby the TPC-C can be used to exploit the potentials of the new storage media, such as the flash SSDs.

Beneath the TPC-C benchmark, specific database applications, such as MySQL and some commercial database application, are installed on host OS. In the OS kernel, special file system modules can be loaded, as well as the device driver. The transaction processing applications will issue the requests to the database systems, then the database system will process the requests and issue IOs to the storage devices (e.g. flash SSDs) via the device driver in the OS kernel.

5.3 Experimental Environment

I built a database server on the same system described in Figure 4.1 in Chapter 4. The software stacks can be illustrated in Figure 5.3.

In my TPC-C benchmark application, I started 30 threads to simulate 30 virtual users with 30 warehouses. The initial database size was 2.7GB. The *Key and Thinking* time was set to zero in order to measure the maximum performance. The mix of the transaction types is shown in the *normal* column of Table 5.1. Unless specially stated, I used this mix for the experiment. Besides the normal mix in Table 5.1, I also configured another two types of workloads: *read intensive* and *write intensive*, in which the *read-only* and *read-write* transactions are dominant respectively.

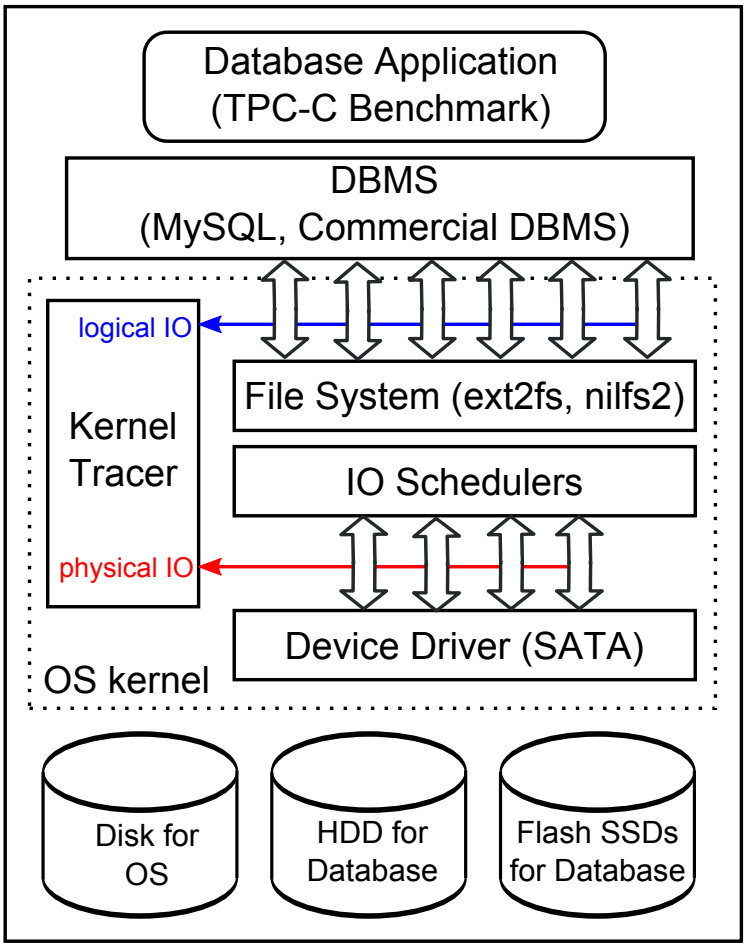


Figure 5.3: Stack of system configuration

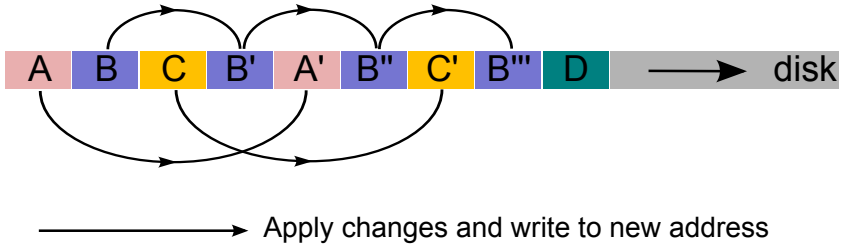


Figure 5.4: Non-In-Place Update techniques

DBMS serves the requests from TPC-C benchmark. In the experiment, I set up a commercial DBMS and an open-source DBMS MySQL. The detailed configuration of these DBMSs is shown in Table 5.2.

Generally, there are two update strategies for the data, in-place update and non-in-place update (NIPU). For the in-place update strategy, the original data was searched firstly and replaced with the new data. As a comparison, the non-in-place update strategy, as shown in Figure 5.4, will write the new data into an new place, leaving the old data obsolete. The obsolete data will be cleaned by the background process called Garbage Collection or Segment Cleaning.

The NIPU techniques convert the logical in-place updates into physical non-in-place updates, using special address table to manage the translation between logical address and physical address. An additional process called garbage collection is required to claw back the obsolete data blocks. A good example of the NIPU technique is the log-structured file system described in [53]. Though the write performance is optimized by some detriment of scan performance [23], this feature is greatly helpful on flash memory to make up for the inefficient random write performance since the random read performance is about two orders of magnitude higher than that of erase operations. The overall write performance is hereby improved.

I selected two file systems as the representatives of two update stratiges for evaluation, the traditional EXT2 file system (ext2fs) and a recent implementation of log-structured file system, nilfs2 [40]. The block size was default to 4KB for both of them. The garbage collection (GC) was disabled by default in nilfs2 for the simplicity of analysis. I will also show the influence of GC in Section 5.4.1.

I also used several IO schedulers in this Linux server. By default, the Anticipatory was used in the experiment because it is the default one in my Linux distribution.

Table 5.1: Transaction types in TPC-C benchmark

Transaction Type	**IO Property**	**% of mix**		
		normal	**read intensive**	**write intensive**
New-Order	read-write	43.48	4.35	96.00
Payment	read-write	43.48	4.35	1.00
Delivery	read-write	4.35	4.35	1.00
Stock-Level	read-only	4.35	43.48	1.00
Order-Status	read-only	4.35	43.48	1.00

Table 5.2: Configuration of DBMS

	Commercial DBMS	**MySQL(InnoDB)**
Data buffer size	8MB	4MB
Log buffer size	5MB	2MB
Data block size	4KB	16KB
Data file	fixed, 5.5GB, database size is 2.7GB	
Synchronous IO	Yes	Yes
Log flushing method	flushing log at transaction commit	

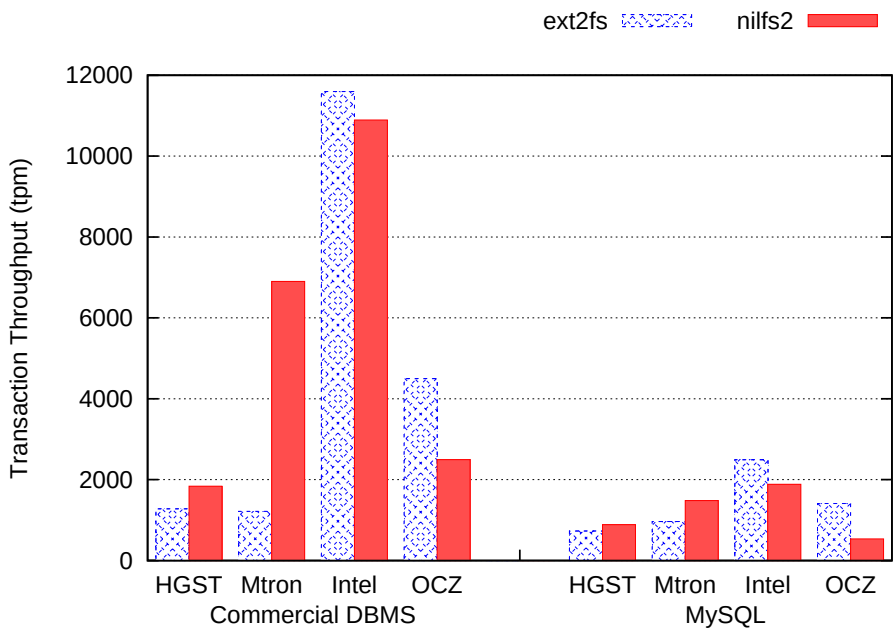


Figure 5.5: Transaction Throughput

5.4 Experimental Results

5.4.1 Transaction Throughput

Transaction Throughput

Figure 5.5 shows the transaction throughput in terms of transactions-per-minute (tpm).

The advantage of flash SSDs over hard disk is clear; the transaction throughput on the SSDs was higher than that on HGST in either DBMS.

For Mtron SSD, a noticeable improvement of nilfs2 over ext2fs on commercial DBMS was observed. This stemmed from the log-structured design that nilfs2 holds. That is, in nilfs2, every time a write is requested, the file system allocates a new space for that request. This helps to avoid the time-consuming erase operation on flash SSDs. Even if DBMS requests a sequence of random writes to the file system, it can give a converted sequence of virtually sequential writes. See again Section 4.3.1, where I confirmed that Mtron's SSD has higher throughput of sequential writes rather than random writes. Nilfs2 successfully exploited this characteristic to derive improved performance. However, contrary to expectation, the advantage of log-structured file system is not clear in Intel's and OCZ's SSDs. I gives analysis on this point in later sections.

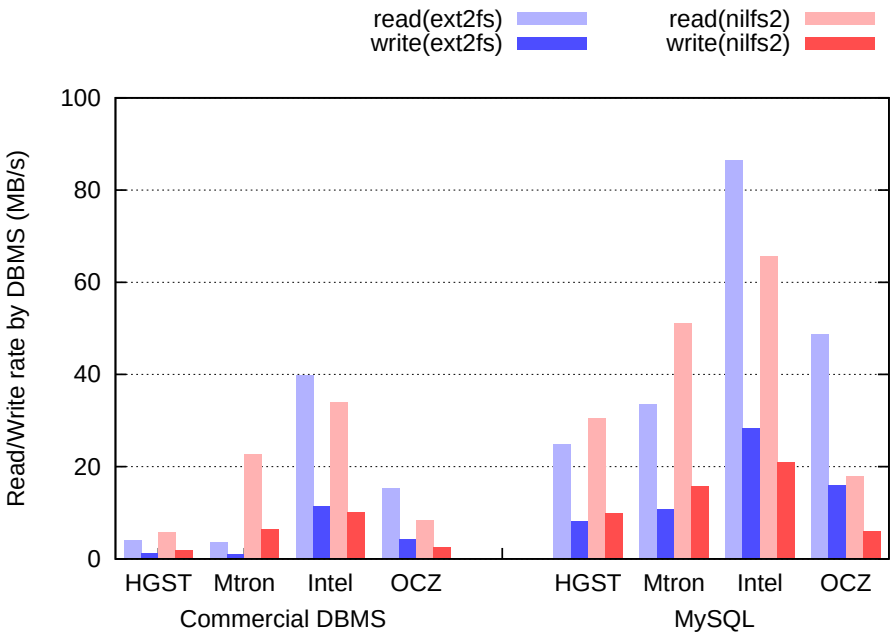


Figure 5.6: Logical IO Rate

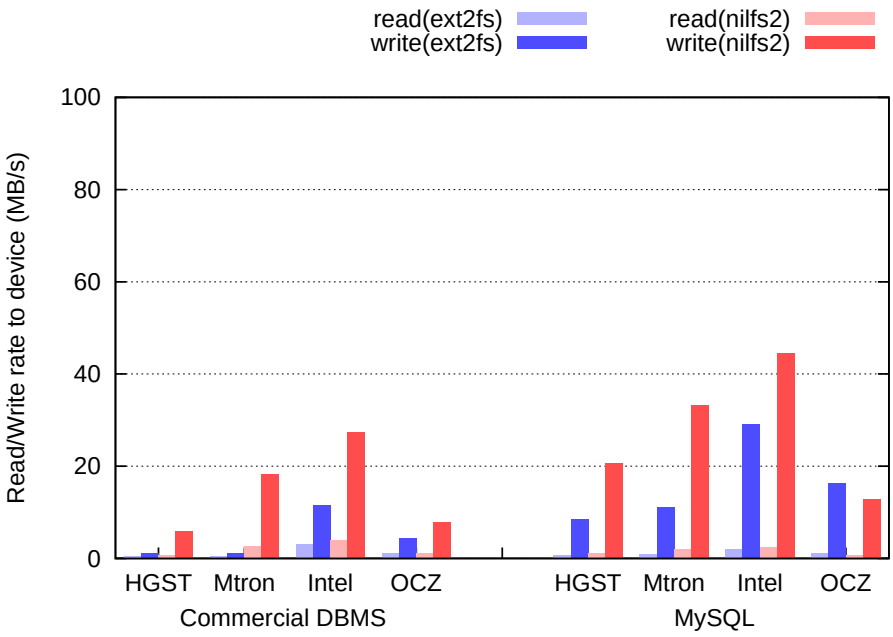


Figure 5.7: Physical IO Rate

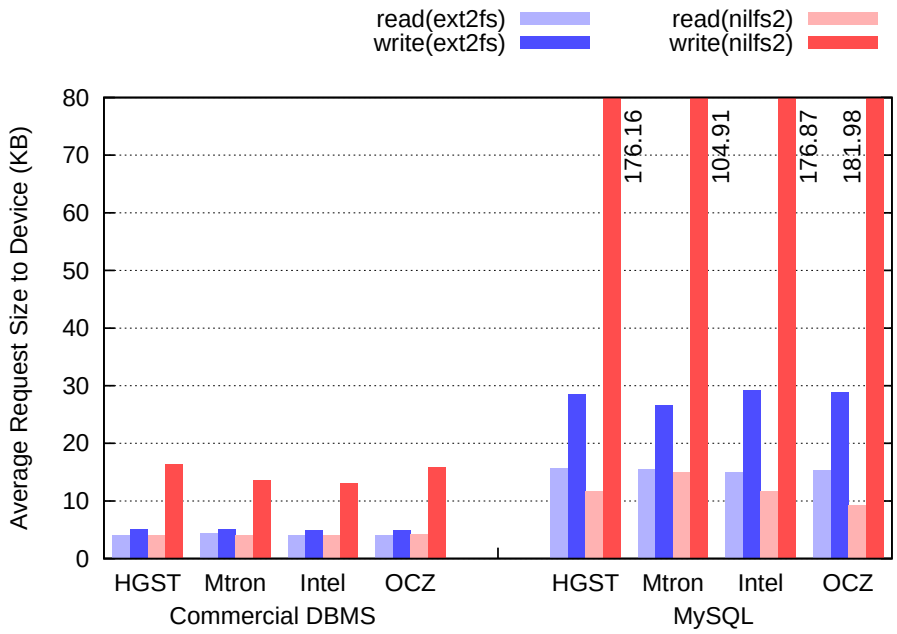


Figure 5.8: Average IO Size

IO Throughput

So as to understand the system behavior more, I traced in-kernel IO events by using SystemTap[56]. Figure 5.6 shows the throughput of file system access given by DBMS under the TPC-C execution. For reference, let us call these file system accesses *logical IOs*, which is also illustrated in Figure 5.3. The workload nature of TPC-C is IO intensive. The overall transaction throughput is mainly determined by the available IO power. Seeing Figure 5.5 and Figure 5.6, I could verified that the transaction throughputs actually followed the logical IO throughputs. Note that the logical IOs may not directly go to the storage device, but rather be split, merged or buffered by the file system. Further analysis is required on the IOs in the underlying layers.

In order to understand the IO path thoroughly, I also analyzed how these logical IOs are processed in the underlying layers. Figure 5.7 presents the throughputs of storage device accesses in the same execution. Let us call these accesses *physical IOs*, which is also illustrated in Figure 5.3. The physical IO rate is the consequence fueled by the file system capabilities and the device power. Read throughputs were always higher than write throughputs in Figure 5.6, whereas write throughputs were higher in Figure 5.7. This means that the file system absorbed many read requests in its buffer.

When ext2fs is used, write throughputs are almost the same between logical throughput and physical throughput. It is probably because that write requests are

temporarily stored in the file system buffer, but most are directly flushed out to the storage device.

In contrast, when nilfs2 is used, write requests are more eagerly optimized. As is mentioned before, nilfs2 has employed the log-structured design. Each time a write is requested, a new storage block is allocated and the write request is routed to the block. This helps avoiding slow erase operations in the flash SSDs. It is clear in Figure 5.8 that the write size of nilfs2 is larger than that of ext2fs because nilfs2 has coalesced the random writes into large sequential writes. Large sequential request is beneficial on hard disks and some SSDs. Actually, I could improve the transaction throughput by using nilfs2 in Mtron's SSD. However, my observation also suggests that I cannot ignore two possible drawbacks of this strategy. First, log-structured strategy has the possibilities of producing more writes. This was confirmed by Figure 5.6 and Figure 5.7. More writes were issued at the physical layer than the logical layer[1]. Even if nilfs2 can improve the IO throughput by converting random writes into sequential writes, additional writes may finally degrade the overall application performance. Second, too large IO sizes have the possibilities of degrading the throughput. In Intel's and OCZ's SSDs, sequential performance decreases when the request size is larger than 32KB, as discussed in Section 4.3.1. This explains why the physical write rate of nilfs2 on Intel and OCZ's SSD in Figure 5.7 is much better than that of ext2fs for the commercial DBMS because the average write size is smaller. For MySQL, since the request size is very large (100KB+), the physical write rate of nilfs2 on Intel's SSD is not so much better than that of ext2fs, on OCZ's SSD it is even worse than that of ext2fs. Note that although the write IO rate of nilfs2 is always higher than that of ext2fs for Intel's SSD, the transaction throughput of nilfs2 is lower than that of ext2fs.

Garbage Collection

The log-structured file system tries to allocate a new data block for writing a data, even if it overwrites the existing data. This strategy produces lots of invalid blocks when write-intensive workload runs for a long time. Garbage collection (GC), a.k.a. segment cleaning, is an essential function, which collects such invalid blocks and makes them reusable for future writes.

So far, I have done the experiments with garbage collection disabled. This was intended for me to measure the potential performance of the system. In real systems, peak workloads may not continue so long and disabling garbage collections can be an acceptable solution in such limited time. But garbage collection

[1] I performed an indirect analysis on the data written by nilfs2, which shows that the additional writes might be caused by the copy-on-write nature on the B-tree used by the nilfs2. This problem is described as "Wandering tree" in [6]. Further analysis on nilfs2 may be necessary.

is also an inevitable topic when I think about long-time operation. I also studied the influence of garbage collection. I got the transaction throughput with different cleaning interval as shown in Figure 5.9. Monotonic performance degradation was observed in all the experimental cases. As garbage collection occurred more frequently, the transaction throughput decreased more. The degradation ratios were varied around $0.77\% - 10.44\%$ with a moderate configuration (10 seconds) and $17.51\% - 38.83\%$ even with a severe configuration (1 second). Mtron's SSD for both DBMSs and OCZ's SSD for commercial DBMS were relatively sensitive to garbage collection, while Intel's SSD for commercial DBMS and MySQL and OCZ's SSD for MySQL were less sensitive.

5.4.2 Transaction Throughput by Various Configurations

Buffer Size

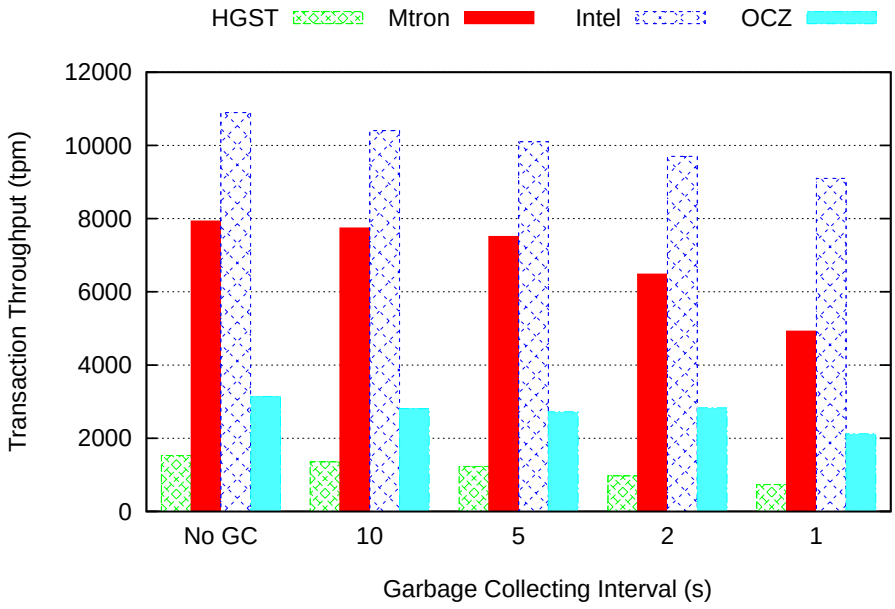
The database buffer plays a vital role to the cache hit rate, the write merging and re-ordering. The buffer size is influential to the performance. The complexity is that DBMS reserves some portion of the available main memory for the database buffer, but the remaining memory space is also used as the buffer cache for the file system, where some optimizations may be tried too. Figure 5.10 shows the transaction throughputs that I measured by varying the buffer size on Mtron's SSD. The absolute performance increased as I increased the buffer size of two DBMSs on both file systems. However, the performance speedup of nilfs2 to ext2fs decreased. With large database buffer size, a lot of random writes can be cached in the database buffer, the amount of random writes reaching the file system was greatly reduced. The advantage of log-structured file system is then reduced.

Workload Type

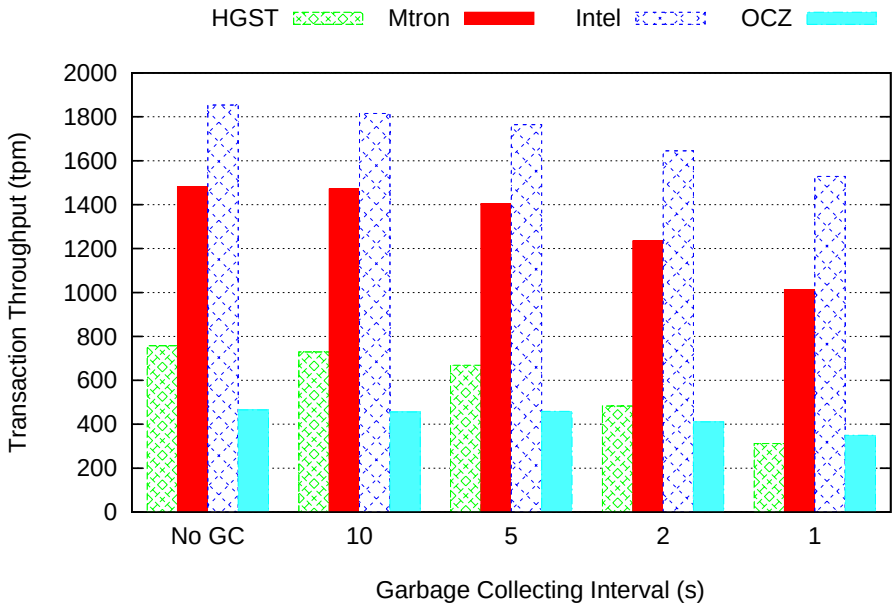
In the experiments presented so far, I have only employed a standard mix of transactions. Here I present another two types of workloads, read intensive and write intensive, as indicated in Table 5.1. As shown in Figure 5.11, absolute transaction throughputs were higher for read-intensive workloads. The speedups from ext2fs to nilfs2 were conversely higher for write-intensive workloads.

IO Scheduler

The IO strategies can also be implemented by different IO schedulers. Four IO schedulers are selected for comparison, as simply described below:



(a) Commercial DBMS



(b) MySQL

Figure 5.9: Transaction Throughput with Garbage Collection Enabled

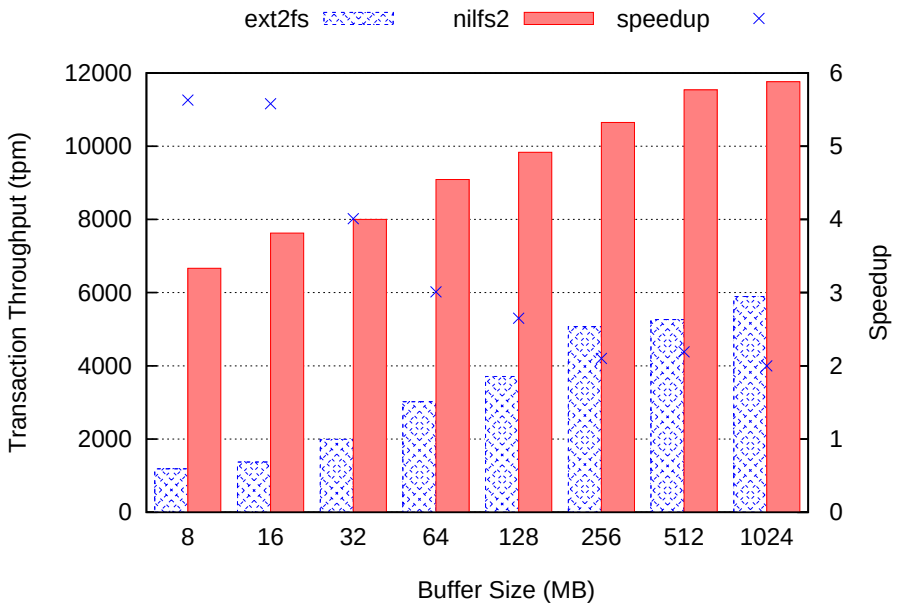
- Noop scheduler is the simplest one, which only merges the requests, and serves them in FIFO order.
- Anticipatory scheduler will do the requests merging and ordering, arrange the requests in the one-way elevator and delay some time to anticipate the next request, to reduce the movement of disk head.
- Deadline scheduler will impose the deadline for each request.
- CFQ (abbreviated for Completely Fair Queuing) scheduler will balance the service time of IOs among processes.

The Noop scheduler is believed to be the best choice for the flash SSDs since there is no mechanical moving parts. Figure 5.12 shows the transaction throughput with four schedulers. The Noop scheduler is not consistently better than other schedulers in all the cases. That is, IO scheduling by the schedulers does not affect the transaction throughput largely.

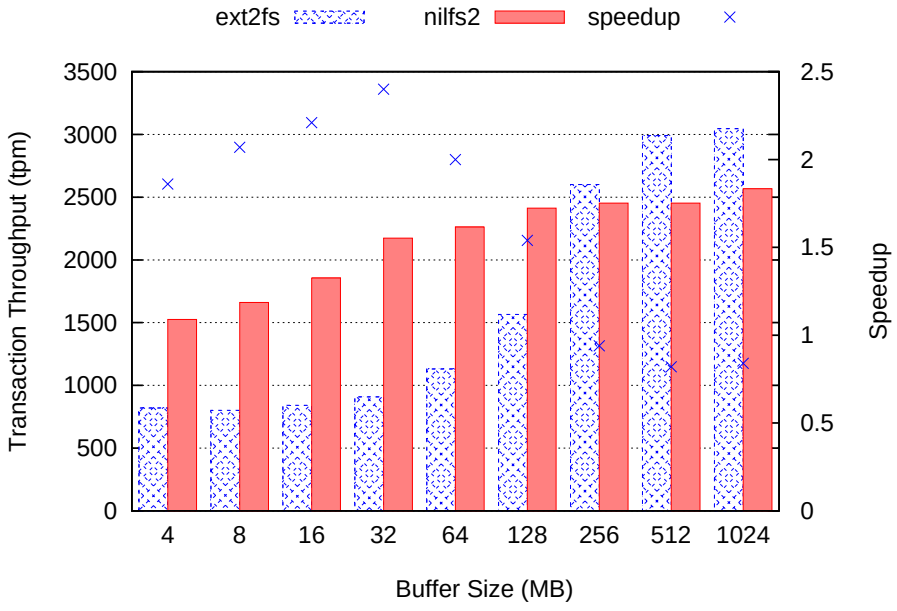
5.5 Discussion on SSD-Specific Features

One innate characteristic of flash chips is the limited programming cycles, which leads to limited lifespan of SSD. If a particular flash page is written in many times, that page will be worn out (i.e. coming to be unable to hold a written data safely) soon even though other pages are healthy. It would shorten the life time of flash SSDs. Balancing the write count among all the flash pages, often called wear-leveling, is an essential solution. One typical technique is to redistribute hot (frequently written) pages to other places [8][22]. If TPC-C is running on a conventional in-place-update file system such as ext2fs, writes are often skewed on particular pages. This technique seems essential to prolong the life span. When I use a log-structured file system such as nilfs2, the file system itself is self-balancing. That is, it can automatically distribute most of pages over the whole address space in a copy-on-write manner. Potentially wear-leveling could be mostly relieved. But wear-leveling is an internal function that is mainly implemented in SSD controller. Real algorithms are not disclosed by any vendors at present. I would like to study the effect of wear-leveling on the choice of file systems in the future.

Although the wear-leveling can prolong the lifespan of the whole disk, the overall write operation count is still limited. The limitation of write operation counts is directly related to the reliability of the SSD. I collected the reliability information of each SSD, as shown in Table 5.3. Given the information in Table 5.3, I try to roughly calculate the endurance of the SSDs in the transaction



(a) Commercial DBMS



(b) MySQL

Figure 5.10: Transaction throughput on Mtron SSD with different buffer size of database system

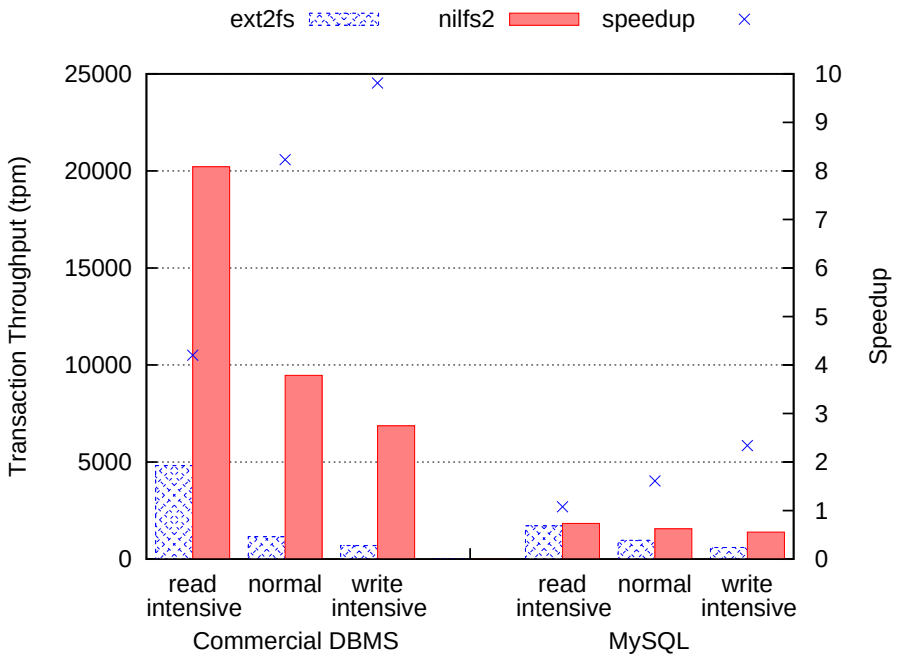


Figure 5.11: Transaction throughput of commercial database with different workload on Mtron SSD

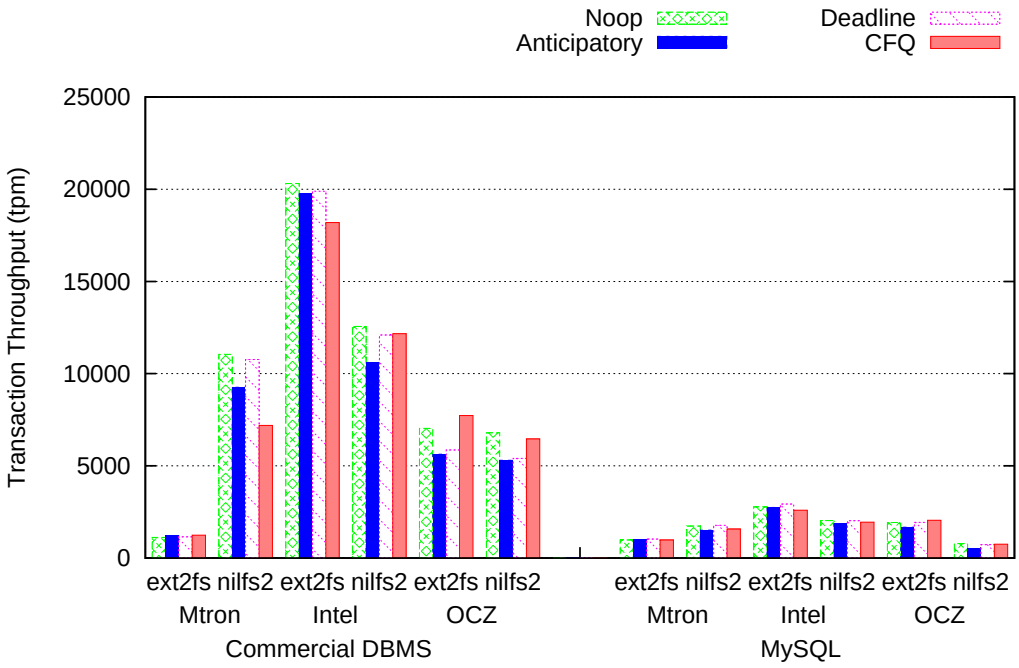


Figure 5.12: Transaction Throughput by different IO schedulers

Table 5.3: Reliability Information of SSDs

Manufacture & Model	Reliability Information
Mtron PRO 7500 [45]	MTBF[1]: 1,000,000 hours, write endurance is greater than140 years at 50GB write/day at 32GB SSD.[2]
Intel X25-E[30]	MTBF[1]: 2,000,000 hours, 64 GB drive supports 2 petabyte of lifetime random writes.
OCZ VERTEX EX[49]	MTBF[1]: 1,500,000 hours

[1] MTBF: Mean Time between Failures.

[2] The above calculation is based on the guaranteed 100,000 program and erase cycles of type of SLC type flash memory from vendors and the assumption that the write is performed in sequential manner.[45]

processing system by the IO throughput at the driver level in my experiment. Intel discloses in the specification that the SSD I used is guaranteed two petabytes of lifetime random writes. Random write produces the largest write counts in general. I obtained an expected minimum lifetime by dividing this guaranteed lifetime write amount (in bytes) by average throughput (MB/s) shown in Figure 5.7. Mtron also discloses its guaranteed lifetime write amount, but it is measured only for sequential writes. OCZ does not disclose any guaranteed lifetime write amount. I could not obtain an expected lifetime for Mtron's SSD or OCZ's SSD. The result is shown in Table 5.4. It shows that Intel's 64GB SSD could only last 1.43 years in the shortest case. Note that, in my experiments, TPC-C ran at top speed, namely without any keying or thinking time, in order to measure the potential performance of SSDs for transaction processing. In real systems, most of SSDs may be running at moderate workloads in most of time, and thus they possibly can survive much longer time. Further investigation is necessary on this point. Boboila *et al.*[8] had shown that the endurance of tested flash chips is far longer than the nominal values by manufactures. More solutions such as the redundancy of SSD or fat provision of flash chips could be also considered to improve the reliability.

Another feature specific to SSDs is the TRIM command [29]. When trying to delete a page in a file volume and/or a database, many file systems and/or database systems do not physically erase content of the concerned page, but merely drop a pointer to the page in the volume meta data (such as inode, directory or catalog). This logical deletion strategy is beneficial in terms of performance, but it would abandon a chance of SSDs to know which page has been deleted by the file systems or the database systems. TRIM, a new storage command, has been proposed as a solution to this. It can inform flash SSDs of which page has been

Table 5.4: SSDs endurance in years by the physical IO throughput shown in Figure 5.7

	Intel (years)
Commercial DBMS with ext2fs	5.47
MySQL with ext2fs	2.19
Commercial DBMS with nilfs2	2.32
MySQL with nilfs2	1.43

logically deleted, so that the notified SSDs can preemptively erase and release the concerned page. This often helps the performance improvement by hiding slow erase operations. Unfortunately this new command has not been supported in my experimental system, so I could not experiment this. I would like to investigate the effect in the future work.

5.6 Summary

I presented performance evaluation of three major flash SSDs with TPC-C benchmark. First, I have clarified the transaction throughput on three flash SSDs, two file systems and two DBMSs. Next, I measured and analyzed the in-kernel IO behavior. Finally, I studied the performance with a variety of configurations for TPC-C. These measurements have provided some practical experiences for building flash-based database systems. The performance benefits of log-structured design were confirmed only in limited cases. It was against my early expectation. The necessity of careful design was verified.

Chapter 6

IO Management Methods for Flash SSD

6.1 Introduction

In chapter 4, I have learned that the performance characteristics of flash SSDs are quite different from that of the traditional hard disk, such as the asymmetric read and write performance, asymmetric sequential write and random write performance, and the bathtub effect for the mixed access patterns.

In chapter 5, I showed the transaction throughput of the database systems by the TPC-C benchmark. A file system with non-in-place update strategy, the log-structured file system, is examined. The IO behaviors along the IO path is analyzed. Although the log-structured file system converted the random writes into sequential writes, I found that the physical IO throughput (Figure 5.7) was still much lower than the available bandwidth of the device (Figure 4.4 to 4.6), as studied in chapter 4.

In order to reach the potential performance, more comprehensive IO scheduling is necessary to process the IOs in a favorable way of SSDs. The IO path is the place for the IO scheduling and hereby the in-depth study of IO path is very necessary. The IO path is the way through which all the IOs will go to the device. If there is a proper and efficient scheduling on the IOs along IO path, the IOs can be well arranged before they reach the device. Therefore, the consideration of the IO management along IO path is important.

In this chapter, I will introduce the IO management methods of the storage subsystem for the database system, discuss the SSD-oriented IO management methods for such a system.

6.2 IO Path in Database Systems

IO path indicates the software or hardware layers through which the application data puts to or gets from the persistent storage devices. In the database systems, the IOs are served by the storage subsystem, so the IO path started from the system call by the database applications, and ended by the storage devices. As shown in the Figure 6.1, the IOs flow through many layers in the OS kernel[1], finally reach the devices. At each layer, there are special optimization techniques with some assumptions, to make sure that the IOs are well organized in the favorable way of the devices when the IOs reach the end of this path.

The IO path has been studied for the storage system for a long time. The IOs are mainly managed and scheduled by the characteristics of the hard disks which have been the main storage devices for several decades. Since the current IO management is specially designed for hard disks, especially to fully utilize

[1]Here I removed many layers for the brevity, more details about the OS kernel can be seen in [10].

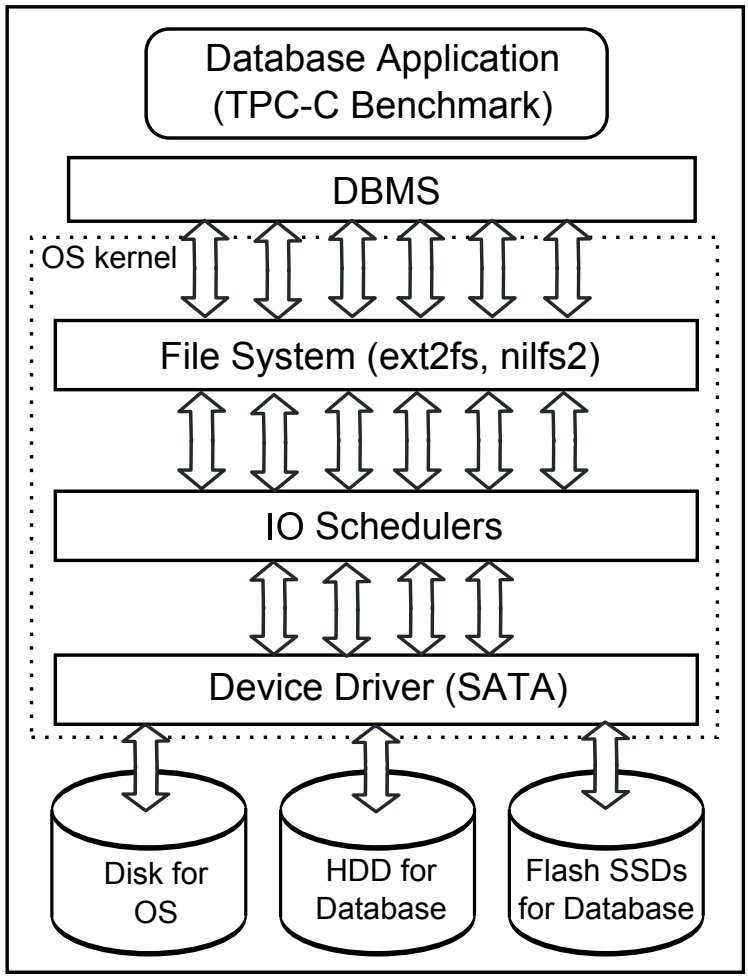


Figure 6.1: IO flow along the IO Path in database system

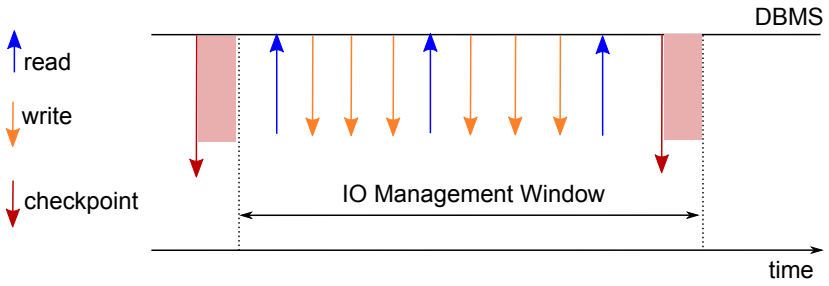


Figure 6.2: IO Management Window

the sequential access performance of hard disks, there will be a mismatch when this IO path is used for flash SSD. Therefore, the SSD-oriented IO management methods along the IO path are essential to reach the potential performance of flash SSDs.

By the knowledge of the basic performance study of flash SSD in chapter 4, the following points should be stressed for the design of the IO path management:

- Since there is no mechanical parts in SSD, the random reads are as fast as the sequential reads for flash SSD.
- The performance of random writes on flash SSDs is poor, and hereby the compensation of poor performance of random writes must be considered.

In the following sections, the SSD-oriented IO management methods will be discussed with above points.

6.3 SSD-oriented IO Management Methods

As described in the previous chapter, the TPC-C experiment showed that poor performance of random writes is dominant in flash-based database systems. Reducing large cost of random writes is a promising direction to try to achieve the potential performance of flash-based database systems. In current IT systems, IO path can be decoupled from database systems and storage devices. Implementing IO management method within IO path is a natural option. In this dissertation, I would like to focus on scheduling writes along IO path in order to improve SSD performance.

Many database systems allow write requests to secondary storage to be deferred and then flushed to the storage in a batch. Such write deferring has benefits of providing scheduling opportunities for reducing IO cost of the writes at run time. As writes are deferred longer, scheduling benefits could be larger, giving higher performance. However, database systems need to guarantee that writes

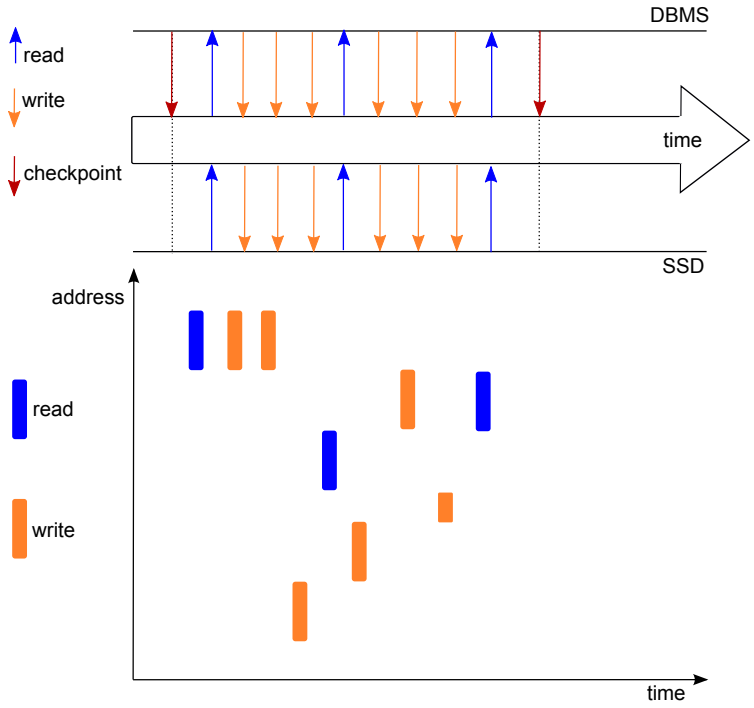


Figure 6.3: IO Management: Direct

older than a checkpoint are reflected onto the secondary storage. The available deferring is strictly limited by database checkpointing. Checkpoint is crucial information for write scheduling when we try to improve the IO performance along IO path.

In this dissertation, available write deferring window is called IO management window as illustrated in Figure 6.2. Each window starts when a checkpoint finishes and the window ends when next checkpoint starts. Writes within a window can be scheduled at run time and then reflected in a batch manner to improve the performance. Scheduling techniques are introduced in the next section.

6.3.1 IO Management Techniques

I will start the discussion on the IO management by a simple example in Figure 6.3 called “Direct” IO. There is no IO scheduling for SSDs in this figure, and the IOs are directly issued to the devices. In Figure 6.3, the upper half illustrates the reads and writes issued by the DBMS and these reads and writes are sent to device directly. DBMS applications are waiting for the IO completion. The below half of Figure 6.3 shows the address space accessed along timeline by the reads and writes issued by the DBMS. The orange arrows denote the write requests and the

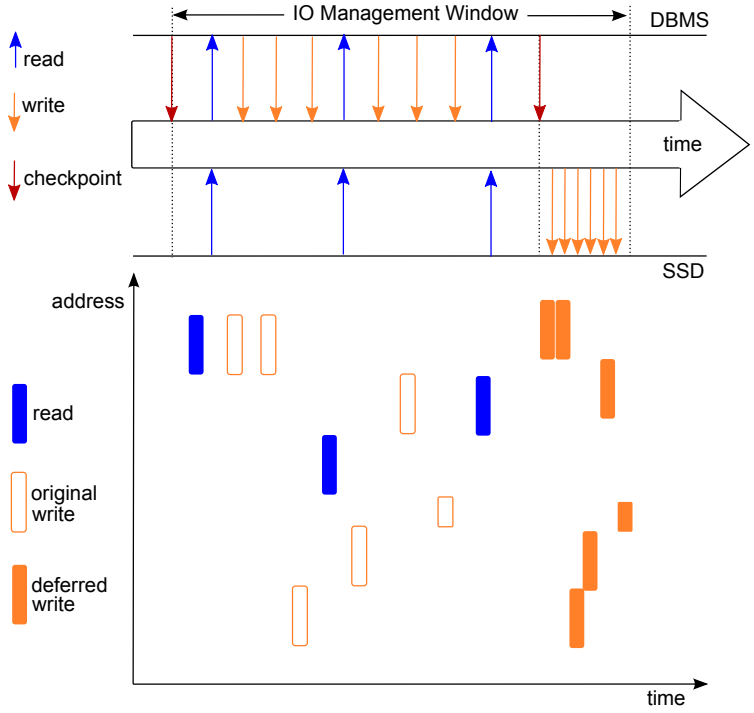


Figure 6.4: IO Management: Deferring

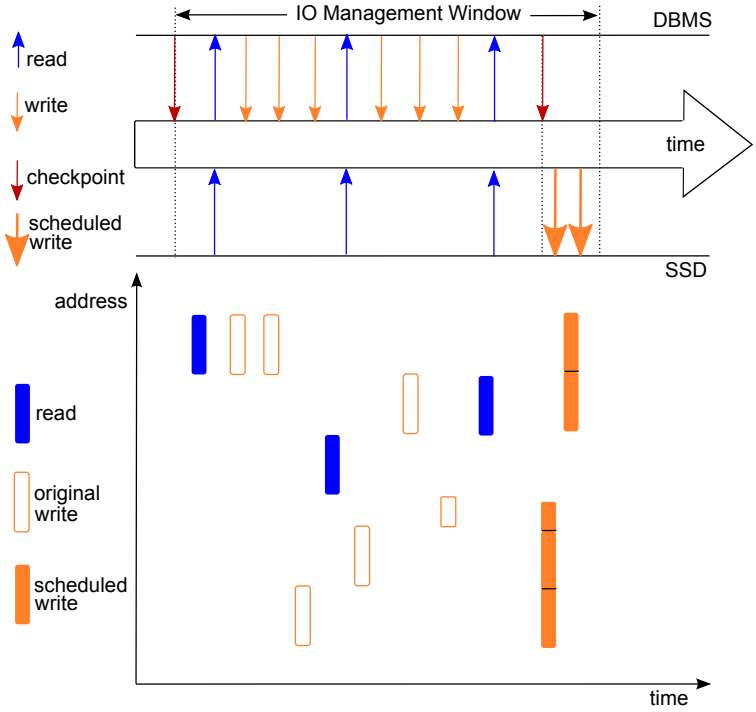


Figure 6.5: IO Management: Deferring + Coalescing

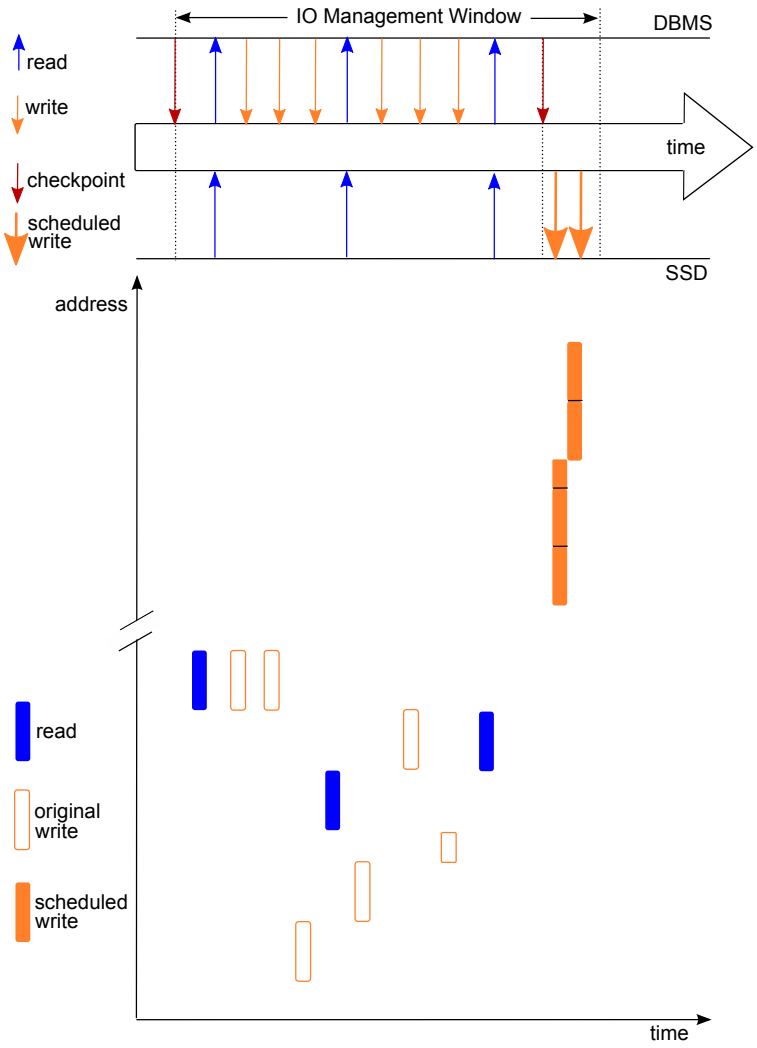


Figure 6.6: IO Management: Deferring + Coalescing + Converting

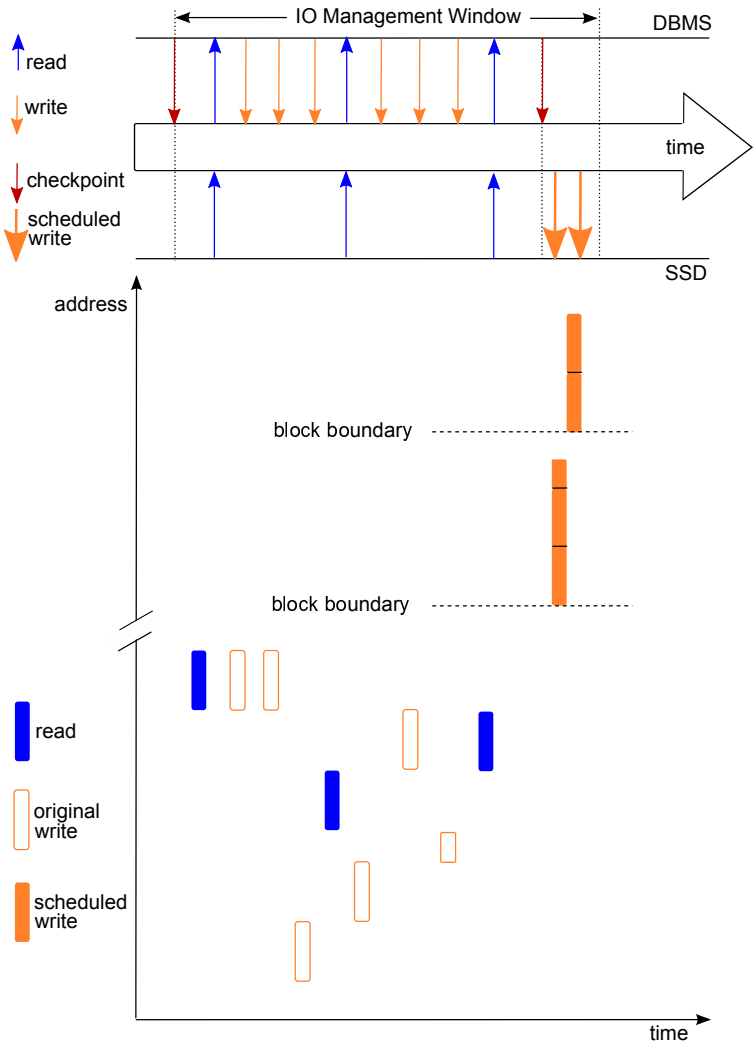


Figure 6.7: IO Management: Deferring + Coalescing + Converting + Aligning

orange rectangles denote the blocks in SSD written by the DBMS. Similarly, the blue arrows and blue rectangles denote the read requests and the read blocks in SSD.

Figure 6.4 illustrates the IO management by the checkpoint information captured from the application. The reads within the current IO path management window will be issued directly to the device since the read performance of flash SSD is very fast. The writes will be deferred, as shown in the upper half of Figure 6.4, the writes denoted by the orange arrows are not issued immediately to the device, but deferred to be flushed to device in batch until receiving the checkpoint information denoted by the red arrow. The write deferring technique is denoted as "Deferring" hereafter. By deferring, the reads and writes are separated automatically, and the scattered random writes are gathered together and can be scheduled with various techniques before being flushed to the device on receiving the checkpoint information. The scheduling techniques (include Deferring) are described as following:

- **Deferring** Defer the writes until the checkpoint notification, as shown in Figure 6.4. Deferring is the basis of the afterward scheduling techniques.

- **Coalescing** Merge the IOs as shown in Figure 6.5, the overlapped writes denoted by orange-color-filled rectangles are merged and concatenated within the IO management window, and hereby the total amount of writes in bytes and write operations are minimized.

- **Converting** Convert the address of IO blocks in a LFS manner. As shown in Figure 6.6, the IO blocks denoted by orange-color-filled rectangles are mapped to the new address space continuously. Therefore, the random writes are converted into sequential writes, and written to comparatively "clean" area which could minimize the IO time spent on the erase operations. The converting technique is more effective on SSD than hard disk. Because converting the random writes into sequenital writes may also converting the sequential reads into random reads. However, the random reads are as fast as sequential reads on SSD, therefore, this side-effect can be well mitigated. The converted random writes can fully enjoy the sequential write performance of SSD because the block erasing time is well mitigated. More details can be found in [60].

- **Aligning** Combine and align IO requests along the erase blocks, as shown in Figure 6.7, the scheduled writes denoted by orange-color-filled rectangles are aligned by the block boundary.

  The aligning is a SSD-oriented technique, because SSD has slow erase operations. The slow erase operations are performed based on the unit of

erase blocks. If a write request is across the boundary of two adjacent erase blocks, both of the erase blocks may possibly need to perform the slow process of “copy data, apply changes, erase block, and write back”. The cost is not justified for a small write request. By aligning, the cost of erase operations will be amortized or reduced. In section 7.3.2, it can be seen that another benefit of the aligning is that the device bandwidth is fully utilized with large request size in our implementation.

6.3.2 SSD-oriented Scheduling

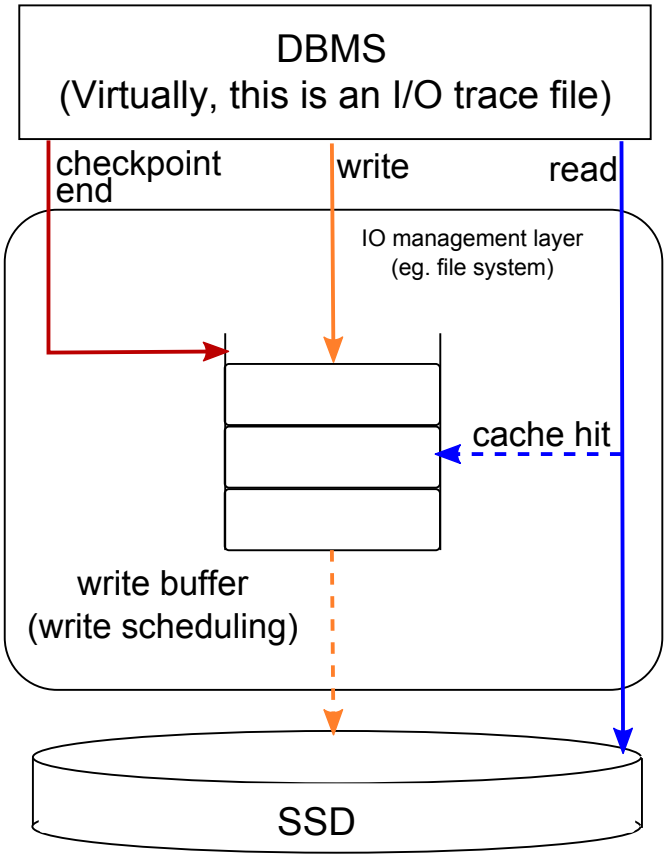


Figure 6.8: SSD-oriented Scheduling

I introduce the SSD-oriented scheduling, which is a scheduling system implemented with the IO management technques described in previous section 6.3.1. As shown in Figure 6.8, the write requests from DBMS will be captured and buffered (Deferring), while the reads will directly go to the device if there is no buffered data for that reads. The buffered data will be applied the scheduling

techniques, such as the Coalescing, Converting, and Aligning. Once the database issued the checkpoint ending information which is captured by the scheduling system, the scheduling system will flush the well managed writes to device in batch.

6.4 Summary

I described the IO path in the current database system, pointed out the design mismatch for flash SSD. Then I described the SSD-oriented IO management methods, introduced the scheduling techniques designed for the flash SSDs, and presented the SSD-oriented scheduling system.

Chapter 7

Performance Evaluation of IO Management Methods for Flash SSD

7.1 Introduction

In this chapter I evaluated the performance of the IO management methods described in chapter 6. First, I evaluated the scheduling with static ordering and merging of IO trace. Next, I evaluated the SSD-oriented scheduling without or with the IO waiting time respectively.

7.2 Experimental Environment

7.2.1 Experiment Configuration

The experimental system was built on the system described in Figure 4.1 in Chapter 4. The software stacks of the TPC-C system are shown in Figure 6.1. The transaction mix is shown in Table 5.1. 30 user threads are started for the TPC-C benchmark. The keying and thinking time is set to 0. I used the Mtron SSD and the commercial DBMS for the evaluation. The configuration of the commercial DBMS is shown in Table 7.1. The buffer size was set to 80MB, which may be a typical setting corresponding to the data size. In order to exclude the effect of the page cache by OS or file system, I chose to use the *raw device* to host the data.

Table 7.1: Configuration of Commercial DBMS

	Configuration of Commercial DBMS
Data buffer size	80MB
Log buffer size	5MB
Data block size	4KB
Data file	fixed, 5.5GB, database size is 2.7GB
Synchronous IO	Yes
Log flushing method	flushing log at transaction commit
Data table space is created on raw device, log files and system files are located in a separated device.	

7.2.2 IO Management Window in TPC-C benchmark

In my TPC-C benchmark system, I developed a script to trigger the full database checkpoint periodically. The interval can be configured as required. The interval of checkpoint determines the IO management window as described in chapter 6.

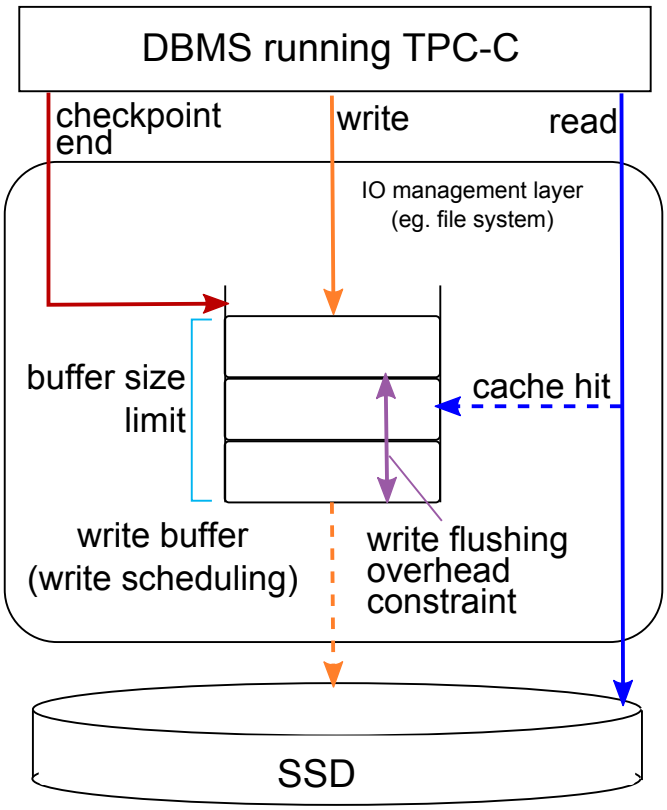


Figure 7.1: SSD-oriented scheduler for TPC-C

Within the checkpoint interval, the writes can be deferred in the buffer to be scheduled by the scheduling techniques described in previous chapter. The checkpoint interval varied from 30 seconds to 600 seconds in the evaluation.

At the end of the checkpoint, the database checkpoint process will write a mark to the data file, and this activity will be captured by my scheduling system, and at this time the defferred writes start to be flushed.

7.2.3 SSD-oriented IO Scheduling for TPC-C

Figure 7.1 illustrates the SSD-oriented scheduling system for my TPC-C benchmark system. In my experimental system, there will be some limitations required by this real TPC-C system, such as the buffer size limitation or the write flushing overhead constraint, as shown in Figure 7.1. So I defined another two constrains for the scheduling system; the buffer size threshold θ_n and the checkpoint overhead threshold θ_t. θ_n ensures the buffer size limitation is not exceeded, and θ_t ensures the write flushing overhead constraint, and hereby ensures the checkpoint flushing overhead is not too long.

7.2.4 Combination of the Scheduling Techniques

I configured several combinations of the scheduling techniques, as shown in Table 7.2. Their effectiveness will be examined one by one in the following sections.

Table 7.2: Combinations of Scheduling Techniques

Notation	**Deferring**	**Coalescing**	**Converting**	**Aligning**
drt				
dfr	√			
dfr_cls	√	√		
dfr_cnv	√		√	
dfr_cls_cnv	√	√	√	
dfr_cnv_aln	√		√	√
dfr_cls_cnv_aln	√	√	√	√

7.3 Evaluation

In this section, I firstly presented the baseline, the direct replay results as a reference for the subsequent measurement results, then the results with static ordering and merging of IO trace will be provided one by one. Next is the results of online scheduling without or with IO waiting.

7.3.1 Baseline

In order to clarify the influence of the IO response time in the total transaction processing time, I studied the response time of IO requests. Since the OS and file system buffer is excluded in the raw device case, the IOs are consistent between the system call layer and the device driver layer. The IOs can be queued deeply by multiple threads submitting the IOs simultaneously, and hereby the response time values of IO requests are overlapped one another. So the response time may not exactly reflect the IO time of individual request. If the IOs are replayed *one-by-one* (queue depth is one) on raw device with the same configuration, the exact response time of individual request can be obtained. The sum of response time of all requests is the total time spent on the IO in the execution. If this sum of response time is reduced, that is, the IO time is reduced, then the overall performance of the system (IO-bound) may be improved.

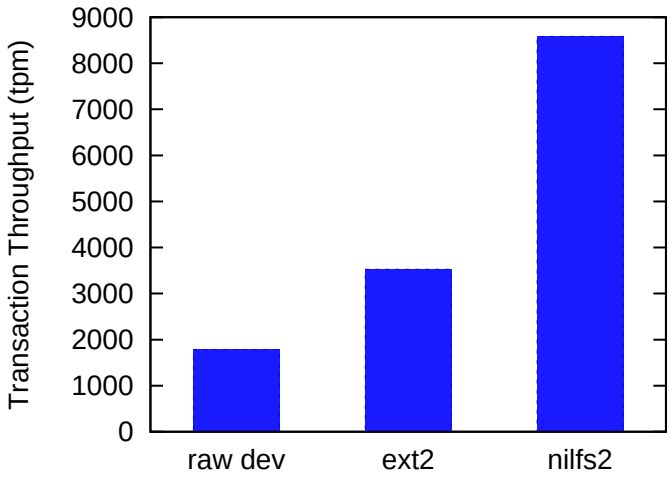


Figure 7.2: Transaction Throughput of Comm. DBMS on Mtron SSD with 80MB DBMS buffer

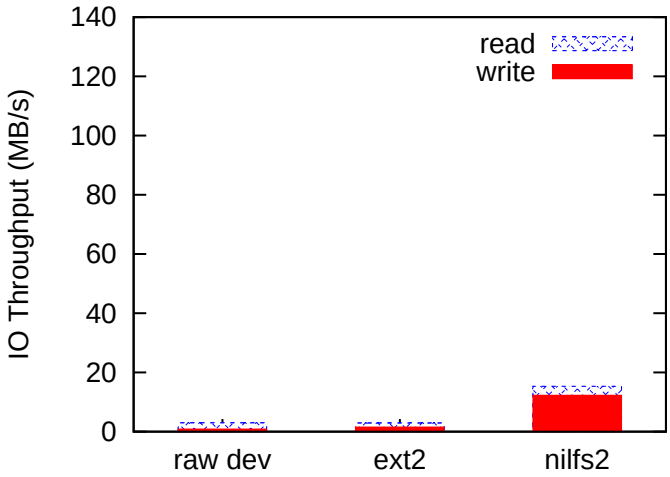


Figure 7.3: IO Throughput of Comm. DBMS on Mtron SSD with 80MB DBMS buffer

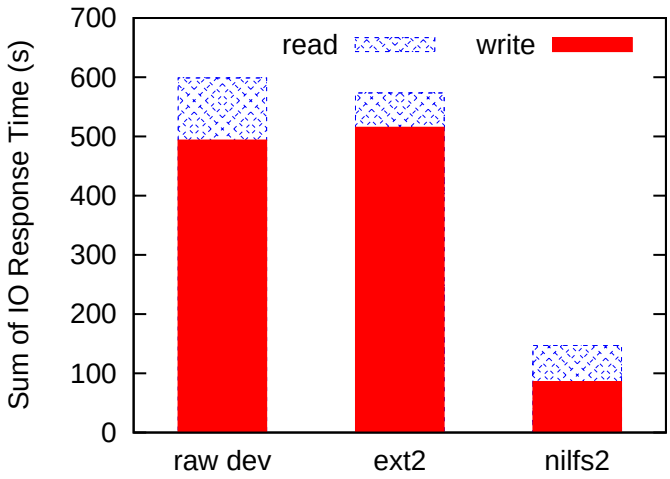


Figure 7.4: IO Replay of Comm. DBMS on Mtron SSD with 80MB DBMS buffer

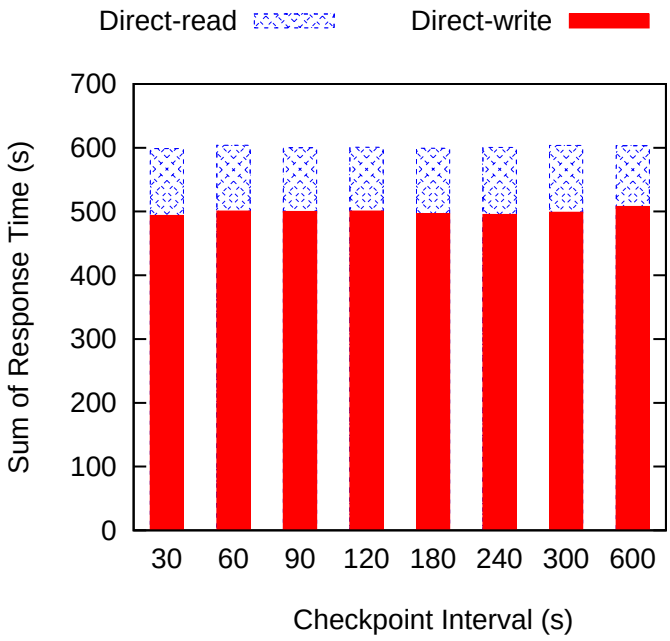


Figure 7.5: Sum of the IO response time of the raw device case

In order to get the total IO time, I did the TPC-C experiment with IO traces. First, I did the experiment with TPC-C benchmark with the configurations decribed in section 7.2.1. In addition to the case of hosting the data with raw device, I also added another two cases: hosting the data with device formated by ext2 file system and nilfs2 file system. I obtained the transaction throughput as shown in Figure 7.2. The IO throughput is shown in Figure 7.3. It is clear that the IO throughput of nilfs2 (sequential write throughput) is still far from that shown in the micro benchmark results in Figure 4.4. Next, in order to get the total IO time, I traced the IOs at the system call and device driver level for 600 seconds, then I replayed the IOs at the different layer. The system call trace is replayed in the raw dev case, while the traces at the device driver level are replayed in the rest file systems cases. The IO time of one-by-one raw IO replay is shown in Figure 7.4. It shows that the summary of the total IO time reflects the transaction throughput in Figure 7.2 to some extent. The sum of IO time in the raw dev case is very close to the overall execution (trace) time (600 seconds), which implies that this system was "IO bound". If the IO time is reduced greatly, especially the write time which is the major part as shown in Figure 7.4, the overall performance (transaction throughput) may also be improved accordingly. Therefore, I take the raw device case as a baseline.

In order to get the baseline cases, I started the TPC-C experiments again with the configuration in section 7.2, and captured the read and write requests on data in a period of 600 seconds in the raw device case with different checkpoint intervals. Afterward, I replayed these requests one-by-one with the same configuration. Figure 7.5 shows the summary of the IO response time with different checkpoint intervals (30 to 600 seconds), which is very close to the total execution time (600s). This confirms again that the experiment system is "IO Bound". In the following sections, I will investigate the potentiality of each IO scheduling technique with these baseline cases.

7.3.2 Potentiality of IO Scheduling

In this section, I examined the performance benefit of each scheduling technique step by step with static ordering and merging of IO trace, then I provided the overall improvement with analysis.

Deferring

The purpose of the *Deferring* is to postpone writes to be flushed later in batch. From Figure 7.6, it can be seen that the response time of deferred random writes (bars with the legend of "Deferring") increased compared to the write time in the baseline cases (bars with the legend of "Direct"). This is due to the bathtub

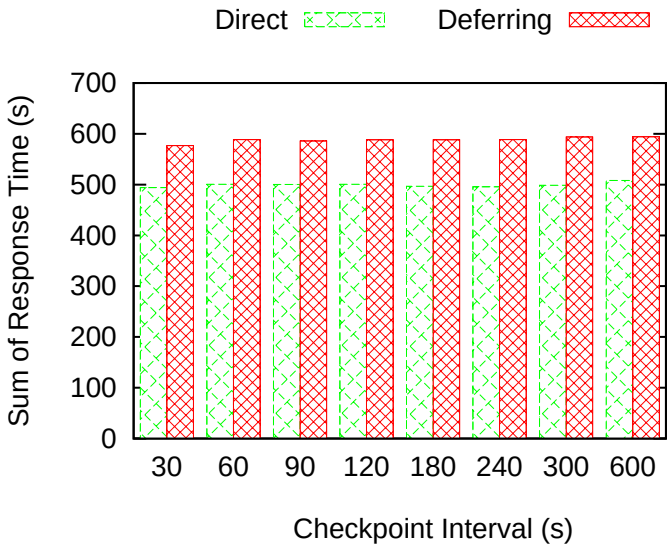


Figure 7.6: Write Time by Deferring

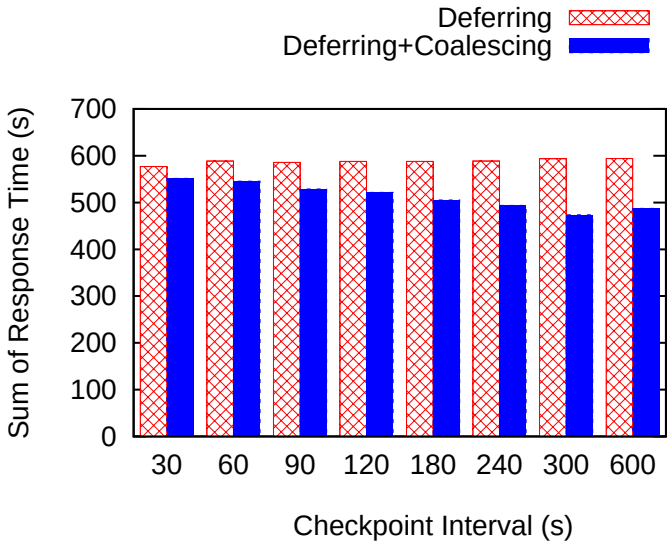


Figure 7.7: Write Time by Deferring and Coalescing

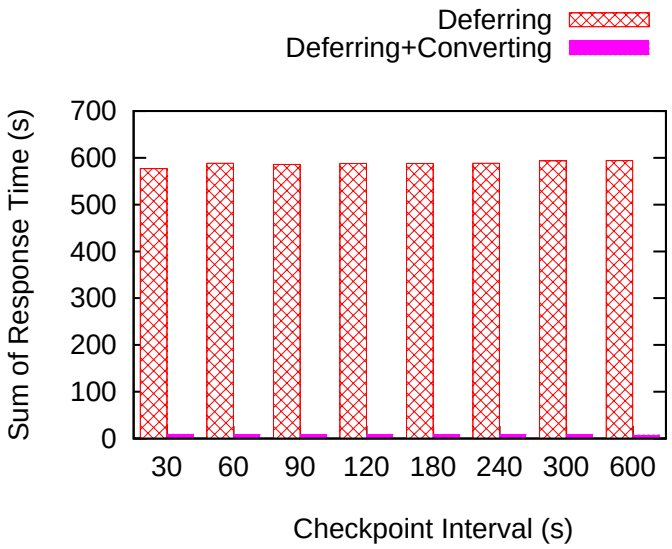


Figure 7.8: Write Time by Deferring and Converting

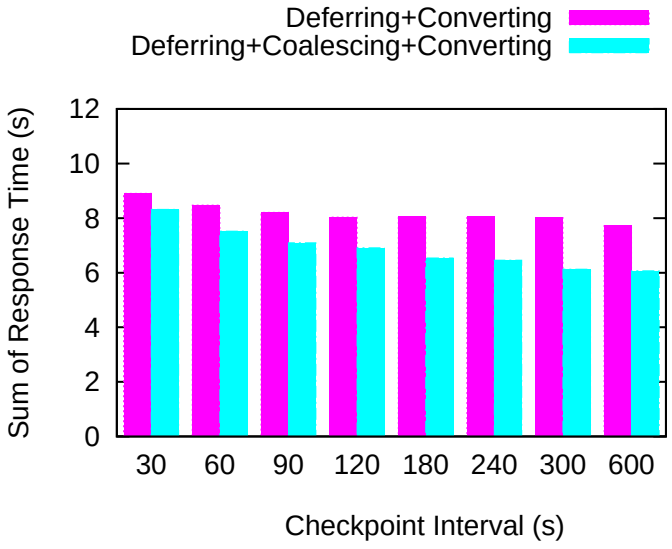


Figure 7.9: Write Time by Deferring, Coalescing and Converting

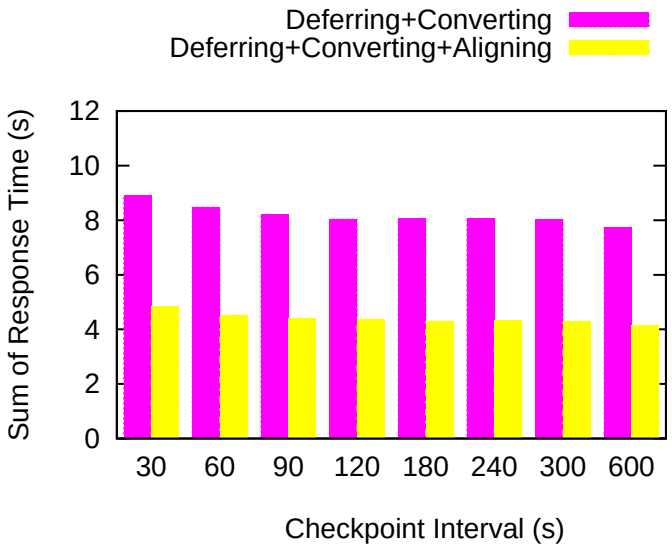


Figure 7.10: Write Time by Deferring, Converting and Aligning

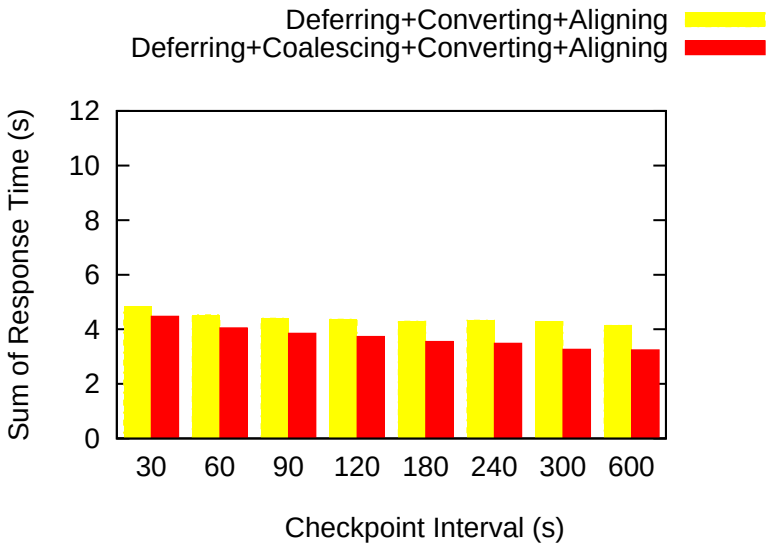


Figure 7.11: Write Time by Deferring, Coalescing, Converting and Aligning

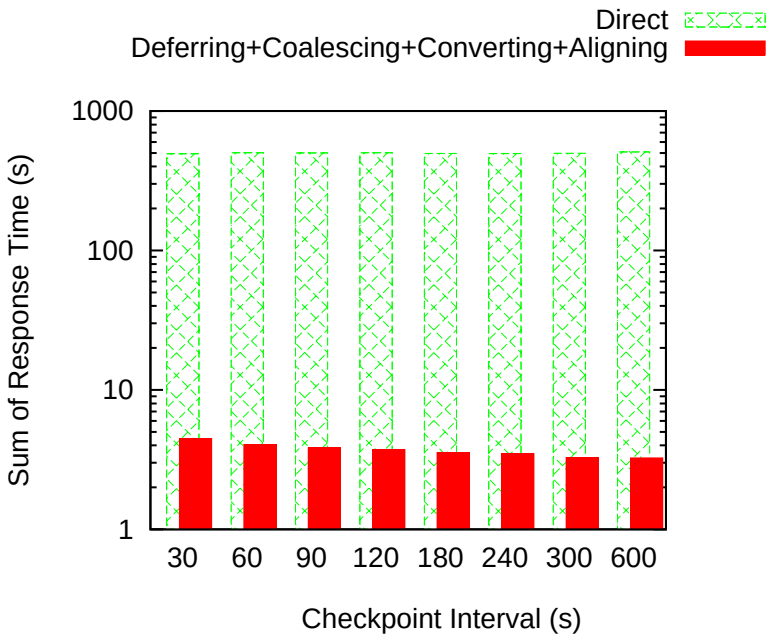


Figure 7.12: Total Write Time Improvement

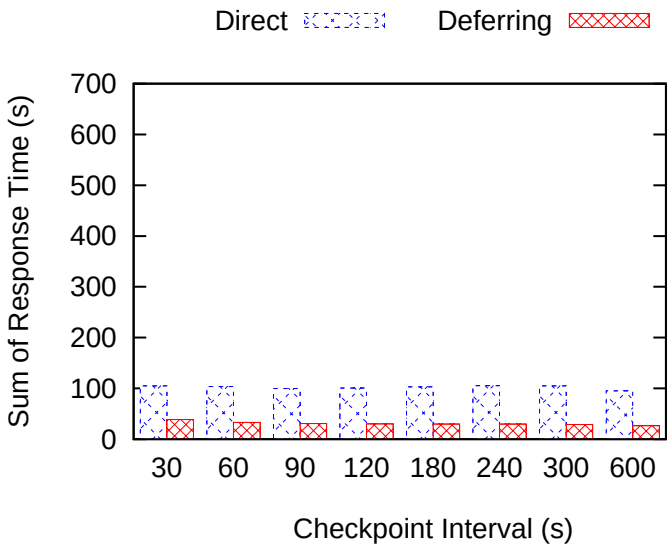


Figure 7.13: Read improvement by write deferring

effect, which is shown in chapter 4. The deferred writes will be managed by the scheduling methods and hereby the performance can be improved considerably, as shown in the following sections.

Deferring + Coalescing

The purpose of *Coalescing* is to reduce the amount of writes. Figure 7.7 shows the performance improvement by applying IO optimization techniques to the deferred writes. In Figure 7.7, compared to the "Deferring" case, the response time is reduced in the "Deferring+Coalescing" case, simply due to the reducing of the writes amount. The amount of merged writes by Coalescing methods is affected by the length of checkpoint interval. With the longer checkpoint interval, the more writes can be coalesced, and thus, more improvement can be obtained. So the improvement is increasing with the increase of checkpoint interval, as shown in Figure 7.7. In our observation, the speedup is from 1.05x to 1.25x.

Deferring + Converting

The purpose of *Converting* is to convert the random writes into sequential writes, like that in the LFS, reducing the cost of erase operations so as to improve the throughput. Figure 7.8 shows that the improvement of converting is significant, from 64.75x to 76.88x. The improvement origins from the asymmetric performance between sequential write and random write.

Deferring + Coalescing + Converting

Combining the Coalescing with the Converting, the IO time can be further reduced. The improvement is 1.07x to 1.27x, as shown in Figure 7.9. This improvement simply comes from the merged amount of writes.

Deferring + Converting + Aligning

The purpose of the *Aligning* is to reduce the cost of erase operations caused by the requests across the erase block boundary. The Aligning is performed on the basis of the Converting. I configured the DBMS with 4KB block size, so that most of the request size is 4KB. I implemented the Aligning by assemble the 4KB requests to 64KB. A typical erase block size, 256KB as shown in Table 2.1, can be dividable by 64KB. Another benefit is that 64KB is the size where SSD bandwidth is beginning to get saturated, as shown in micro benchmark results in Figure 4.4. So I can maximize the utilization of bandwidth, while keeping the request size as smaller as possible. With 64KB alignment, the further improvement on Converting is showing in Figure 7.10, in which the improvement

of “Deferring+Converting+Aligning” over “Deferring+Converting” is 1.84x to 1.87x, which mainly comes from the throughput difference of the 4KB request and 64KB request shown in the Figure 4.4.

Deferring + Coalescing + Converting + Aligning

Before applying the Converting and Aligning technique, the Coalescing technique can be applied on the deferred writes. Figure 7.11 shows the effectiveness with Coalescing. The improvement of “Deferring + Coalescing + Converting + Aligning” over “Deferring + Converting + Aligning” is 1.08x to 1.31x.

Total Improvement

With all the scheduling methods combined together, the overall improvement is shown in Figure 7.12, which shows the write response time of the baseline (“Direct”) and the replay with all the scheduling methods (“Deferring + Coalescing + Converting + Aligning”). The improvement is significant, from 110.30x to 156.39x. Note that the y axix is logarithmic.

An additional benefit by the deferring technique is that the read performance is also improved, due to the bathtub effect studied in chapter 4, as shown in Figure 7.13, in which the “Direct” and “Deferring” denote the read response time of baseline case and writes-deferred case respectively. The read response time is reduced due to the separation from the writes. The speedup is from 2.71x to 3.63x.

Findings

The findings of the experiment described above can be summarized as follow:

1. Performance dominants are confirmed: random writes are the main part of the IO as for the response time.

2. Existing LFSs (which can follow the case of “Deferring+Converting”) do not reach best performance. In the above experiments, the “Deferring + Converting” can gain more than 60x on the response time of writes. But the “Deferring+Coalescing+Converting+Aligning” gives more performance improvement. This confirms that there is a room of further performance improvement.

3. Effectiveness of “Deferring + Coalescing + Converting + Aligning” is confirmed. This combination contributes to significant improvement in the evaluation.

4. Write-optimized deferring technique can also improve read performance due to SSD's bathtub effects.

With above findings, I designed the SSD-oriented scheduler which can perform the online scheduling as described in next section.

7.3.3 SSD-oriented Scheduler

I implemented the SSD-oriented scheduler and showed the online scheduling results in this chapter. I studied the influence of the two configuration parameters, the checkpoint overhead limit θ_t and the buffer size limit θ_n.

Scheduling without IO Waiting

Firstly, the IOs were replayed one by one without consider the time interval between two consective IOs. I want to make sure that more IOs can be processed in a fixed time interval, and hereby the IO throughput and transaction throughput can be improved greatly.

Figure 7.14 shows the results with the checkpoint overhead limit θ_t. θ_t was changed with 1 second, 3 seconds, 10 seconds, and ∞(No checkpoint limit). The response time of "Deferring" and "Deferring + Coalescing" is increasing when enlarging the checkpoint limit, this may due to the bathtub effect since the writes in "Deferring" and "Deferring + Coalescing" are random writes. The results by the Converting method in Figure 7.14 is zoomed in and shown in Figure 7.15. As increasing the checkpoint interval, the Coalescing may work better by coalescing more writes, but this is not clear shown in Figure 7.15 with checkpoint limit changed from 1 seconds to 10 seconds.

An evaluation with varied buffer size limit θ_n is shown in Figure 7.16. For random writes, denoted by "Deferring" and "Deferring + Coalescing", the situation is quite similar to the evaluation with varied checkpoint limit in Figure 7.14. The sequential writes cases by the Converting methods in Figure 7.16 is zoomed in and shown in Figure 7.17, it can be seen that the response time of "Deferring + Coalescing + Converting + Aligning" is slightly decreasing when increasing the buffer limit, which means my scheduler can have further performance improvement with more buffer.

Scheduling with IO Waiting

In previous experiments, the keying and thinking time of end users was ignored. In the real system, the end users may require the time to typing the data and thinking the next step. So I inserted the IO waiting time to evaluate the scheduling methods.

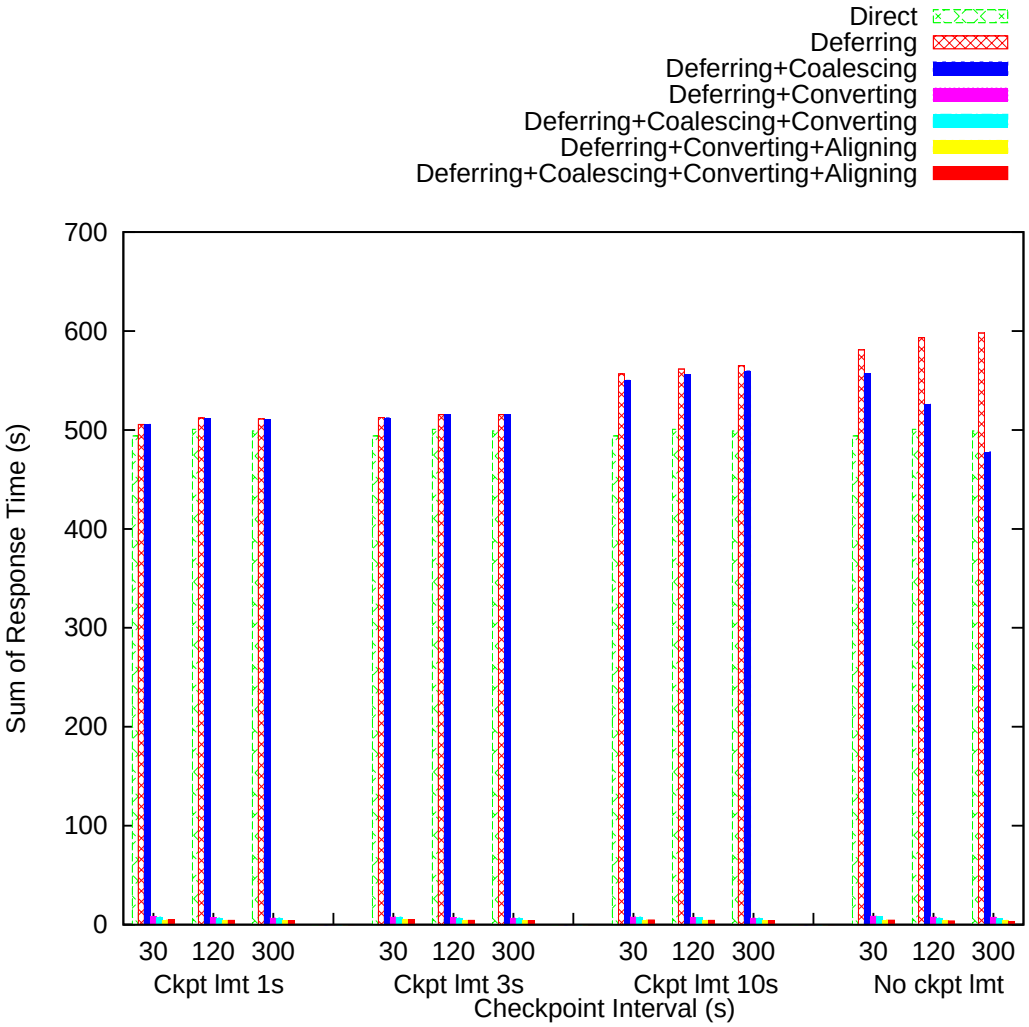


Figure 7.14: Online Scheduling with varied checkpoint limits

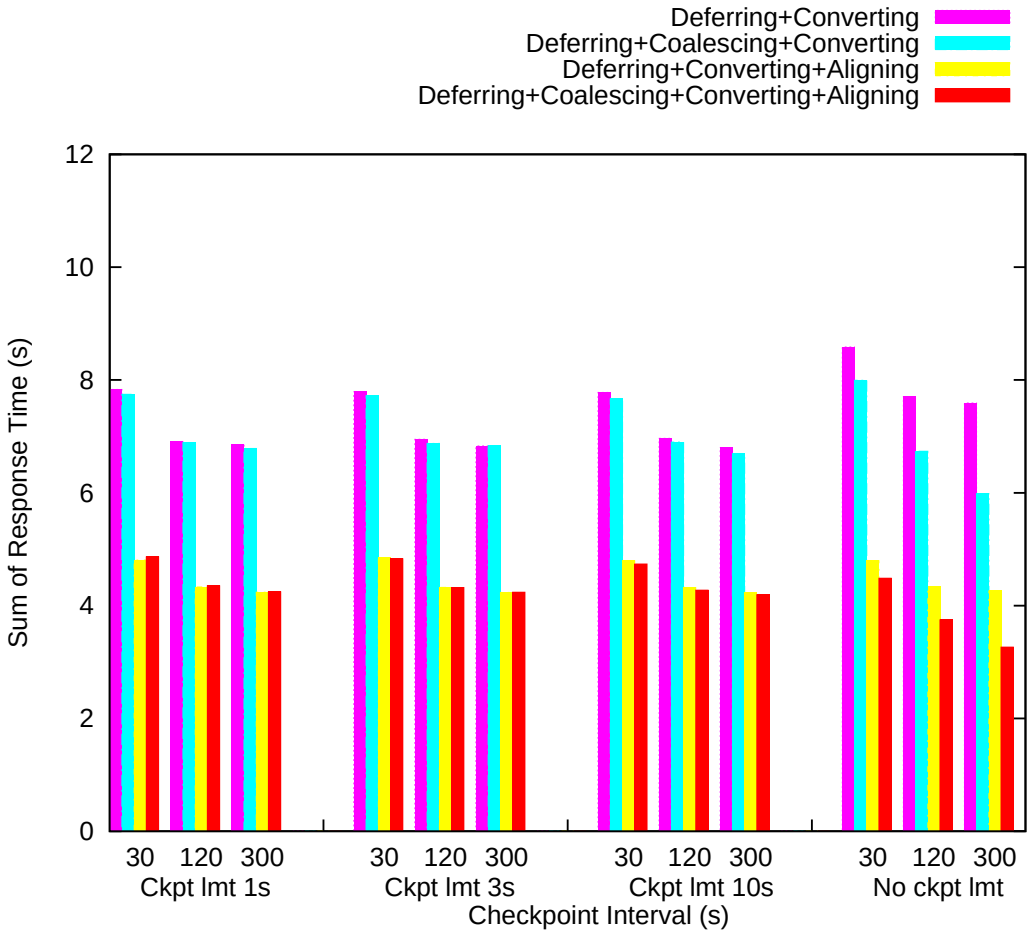


Figure 7.15: Online Scheduling with varied checkpoint limits (Converting Cases)

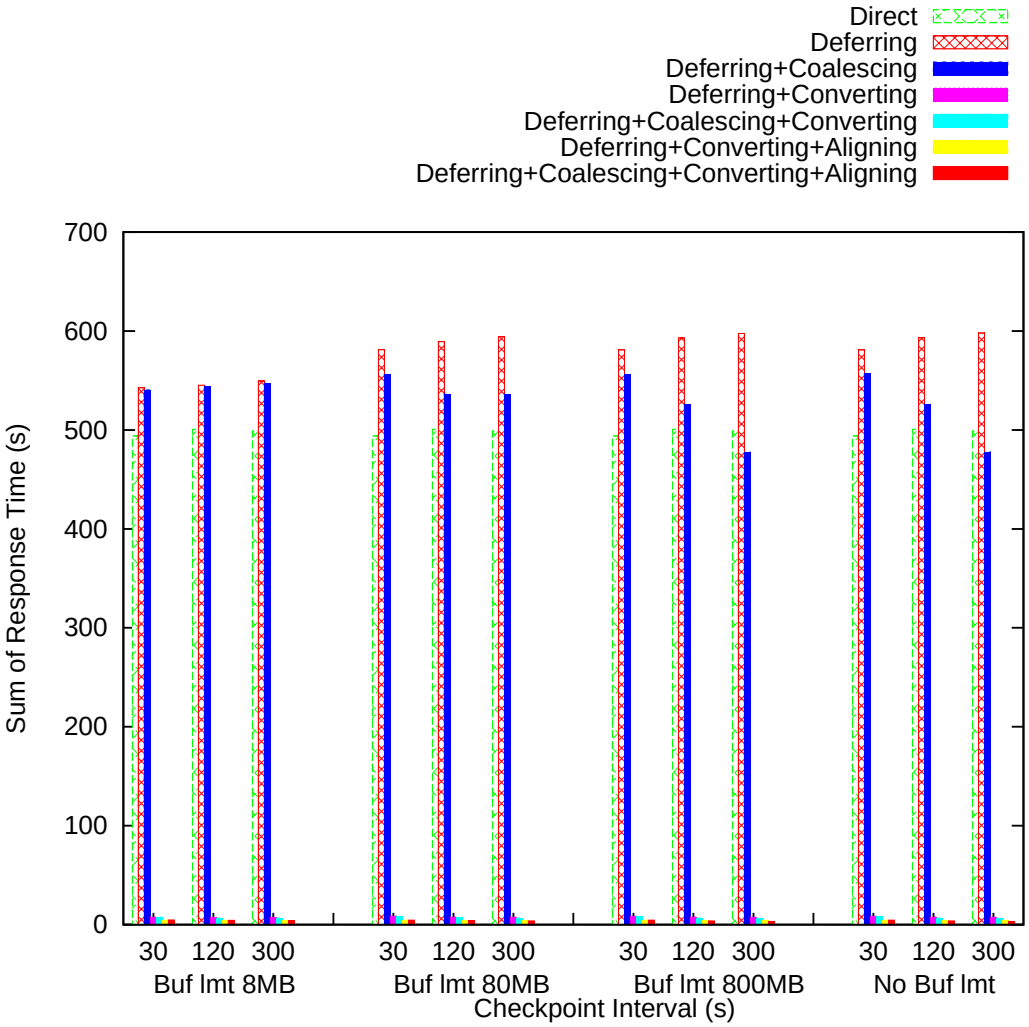


Figure 7.16: Online Scheduling with varied buffer size limits

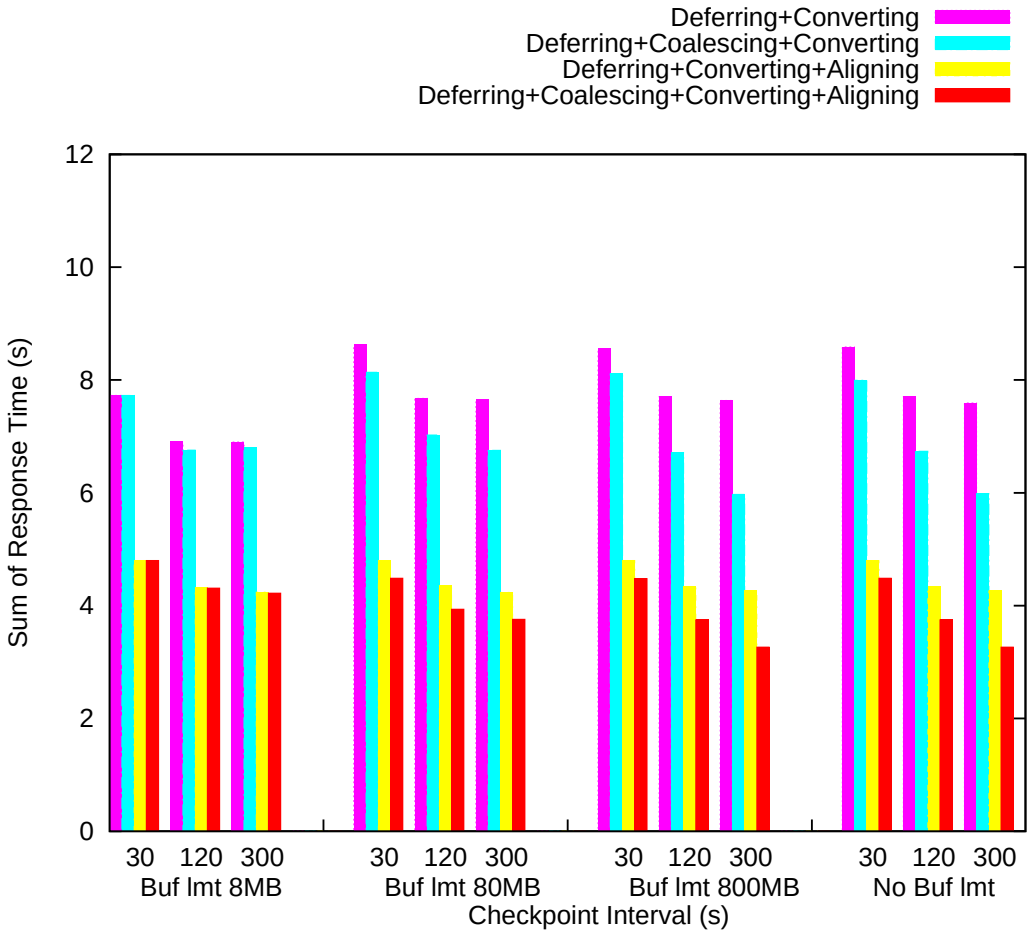


Figure 7.17: Online Scheduling with varied buffer size limits (Converting Cases)

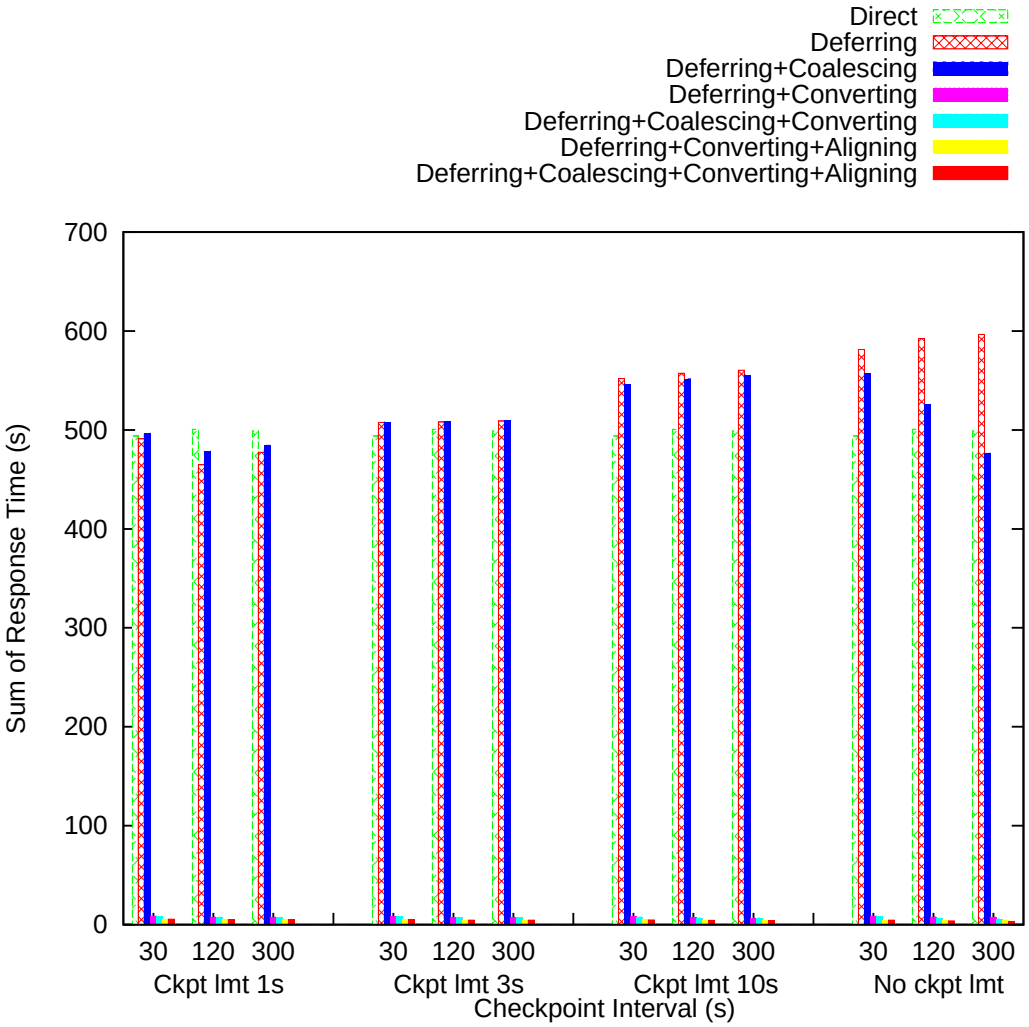


Figure 7.18: Online Scheduling with IO waiting and varied checkpoint limits

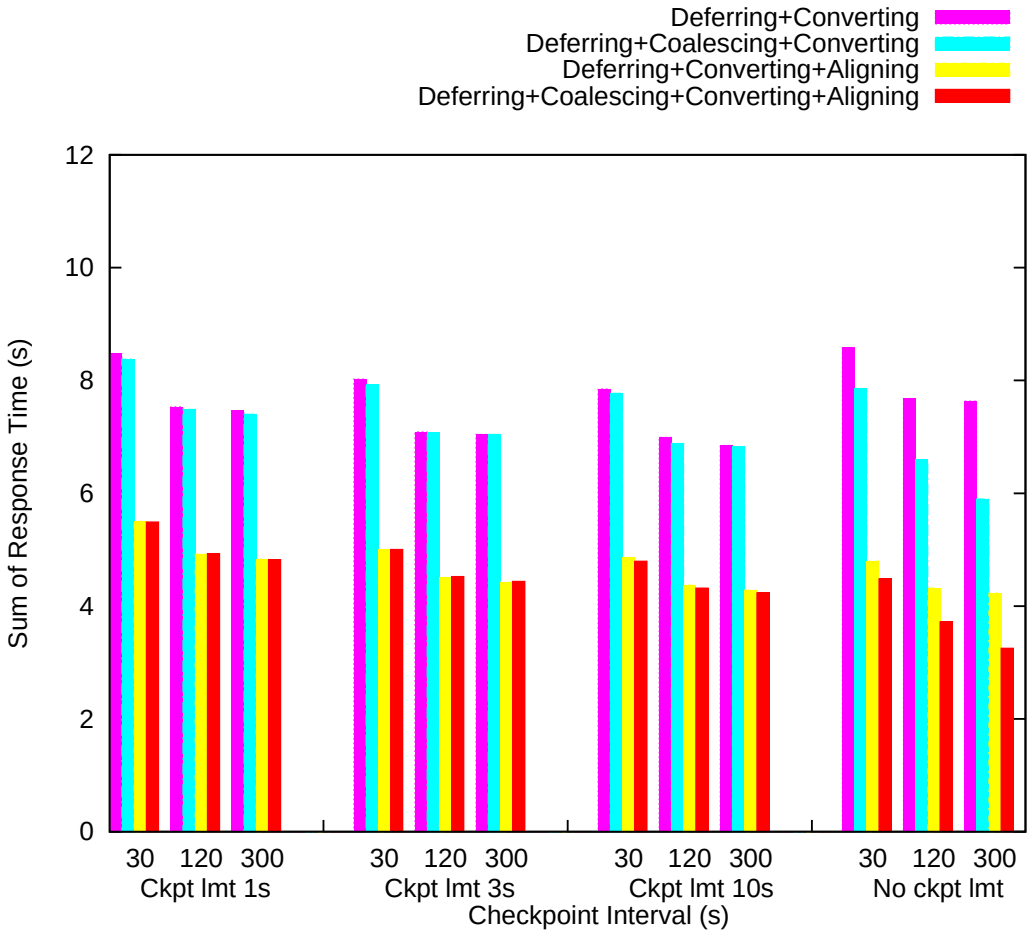


Figure 7.19: Online Scheduling with IO waiting and varied checkpoint limits (Converting Cases)

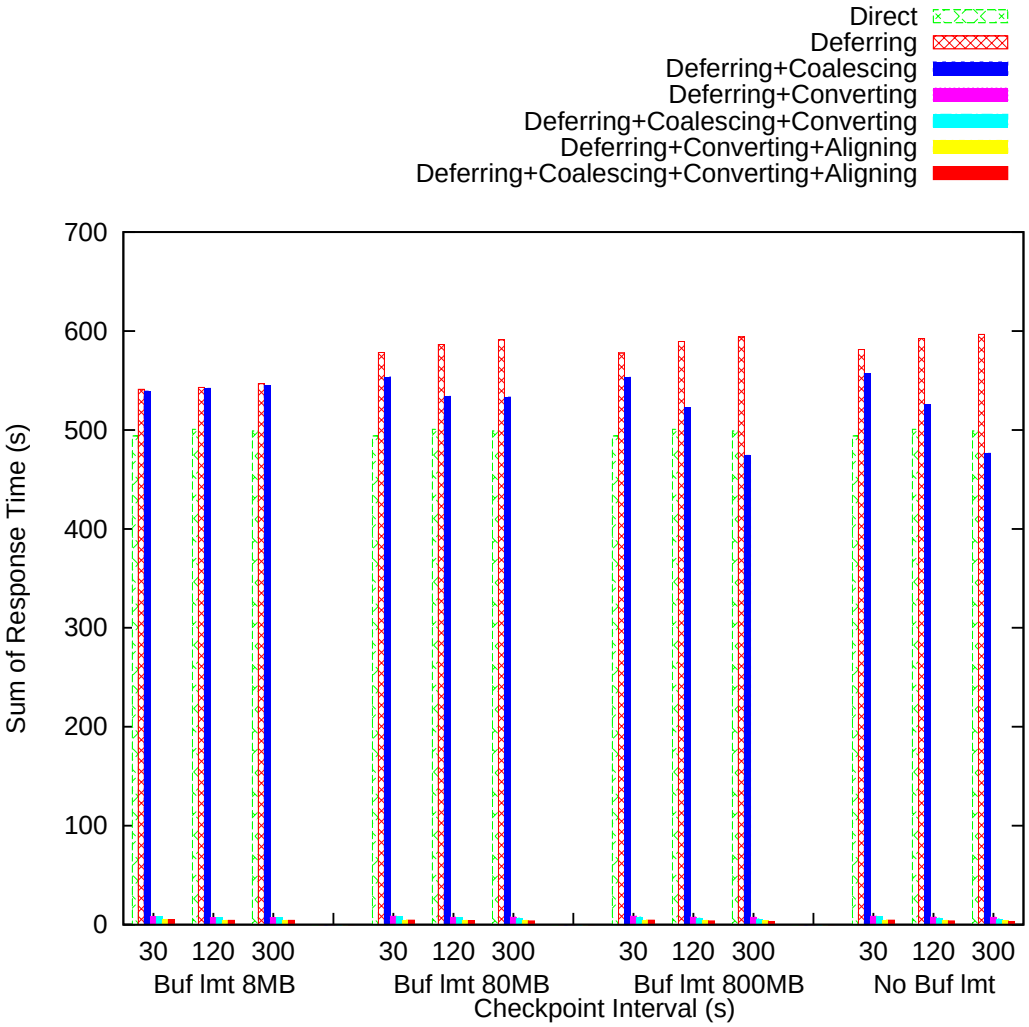


Figure 7.20: Online Scheduling with IO waiting and varied buffer size limits

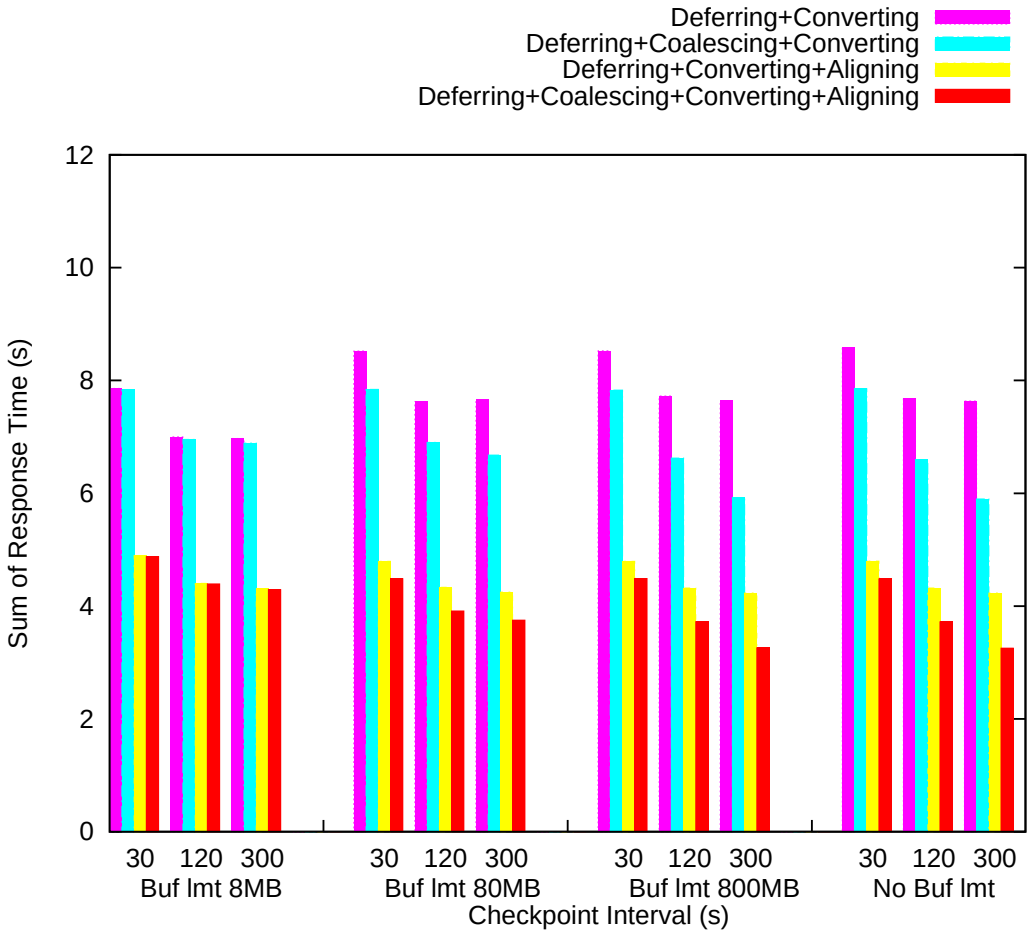


Figure 7.21: Online Scheduling with IO waiting and varied buffer size limits (Converting Cases)

The IO waiting interval is calculated by the timestamps in the original IO trace. The results is shown in Figure 7.18, 7.19, 7.20, and 7.21. It is similar to the results without IO waiting. Therefore, the online scheduling system is confirmed to be effective in this case.

Findings

The findings with the online scheduling methods can be summarized as follow:

1. LFS (Converting) can reduce IO time significantly with small resources (little buffer requirement and little computing requirement).

2. SSD-aware alignment can further reduce IO time with small resources (buffer size requirement is small)

3. Coalescing can further reduce IO time, but its effect is not so large even with large resources (increasing the buffer size limit). The Coalescing needs large buffer to calculate and merge the requests.

4. Performance is not so sensitive to checkpoint interval. The varying of checkpoint overhead limit is not influential to the overall performance.

Discussion

Let me discuss performance improvement that we confirmed in an actual LFS implementation, NILFS2 and potential performance improvement that I verified in my IO replaying experiment. For TPC-C IO sequence, my IO replaying experiment showed more than x60 performance improvement for writes as depicted in Figure 7.8, in contrast, NILFS2 could present only about x5 improvement as depicted in Figure 7.4. Further analysis is necessary for this, but I am considering a possible explanation. The proposed IO scheduling mechanism eagerly defers write requests by using database checkpointing information to flush writes in a batch manner. This can be effective for successfully generating almost pure sequential write accesses, and thus present significant performance improvement. In contrast, NILFS2 needs to flush every write since it has no information of available IO scheduling window. Random writes were actually converted in a LFS manner, but those writes were not issued separately from reads. Consequent IO sequence could be a mixture of reads and writes and the file system may fail to achieve the potential performance of the SSD. This result encourages me to build a new IO scheduling mechanism.

7.4 Summary

In this chapter, I evaluated the SSD-oriented scheduling methods. Significant performance improvement is confirmed. Some valuable findings are listed, which is helpful to design the SSD-oriented scheduler. As for the SSD-oriented scheduler, I summarized the following points for the positioning of the scheduler:

1. The use of checkpoint information enables to bring many opportunities of write scheduling.
    - Checkpoint conscious is a must.
2. The scheduler should be SSD-oriented, hereby it can maximize the performance benefit. Techniques to be considered:
    - Deferring (benefit from bathtub effect)
    - Converting (benefit from the sequential performance)
    - Aligning (benefit from the amortization of erase operations and the bandwidth)
3. The scheduler should be light-weight, because
    - The DBMS usually uses large resources, therefore, the scheduler should be light-weight so as not to compete for the resource with DBMS and process the requests efficiently for DBMS.
    - Simple design and easy implementation is important to apply it into the current database systems.

Chapter 8

Conclusion

8.1 Conclusions

In this dissertation, I had a research on high performance database management systems with flash SSDs. Firstly I studied the basic performance of flash SSDs, which validated the different characteristics of flash SSDs. On the basis of the basic performance study, I continued the evaluation of high performance database systems built on the SSDs, and analyzed the IO behaviors along the IO path. With the knowledge of the basic performance of the SSDs and the analysis of the IO path of the database systems, I designed the SSD-oriented IO scheduling system, which can obtain the application information such as the checkpoint, and then schedule the IOs with SSD-oriented techniques. Four techniques were adopted in this scheduling system according to the characteristics of the flash SSD. I evaluated the potentiality of SSD performance by static scheduling of IO trace. The SSD-oriented scheduling with TPC-C was also implemented and evaluated.

With the research results shown in this dissertation, I draw a conclusion with the following points:

1. The random writes are dominating the IOs due to the innate characteristics of flash SSDs. To improve the performance of database systems on SSDs, random write performance should be considered carefully.

2. Log-structured IO optimization, such as LFS, is effective to solve the problem of slow random writes with little resource. However, it is clearly shown in my scheduling system that the existing LFS failed to reach the optimal performance.

3. The effectiveness of deferring and aligning the writes is confirmed. The combination of “Deferring + Coalescing + Converting + Aligning” almost reaches the optimal performance in the evaluation.

4. Write-optimized deferring technique can also improve read performance due to SSD’s bathtub effect.

With the evaluation of the SSD-oriented scheduling system, I believe that a light-weight, checkpoint-conscious, and SSD-oriented scheduler can maximize the performance benefit of SSDs for the high performance database systems.

8.2 Future Work

The research in this dissertation can be taken as a reference to design and implement a SSD-oriented scheduler for the high performance database system. Specially, as I have shown in the online SSD-oriented scheduling, the buffer size limit

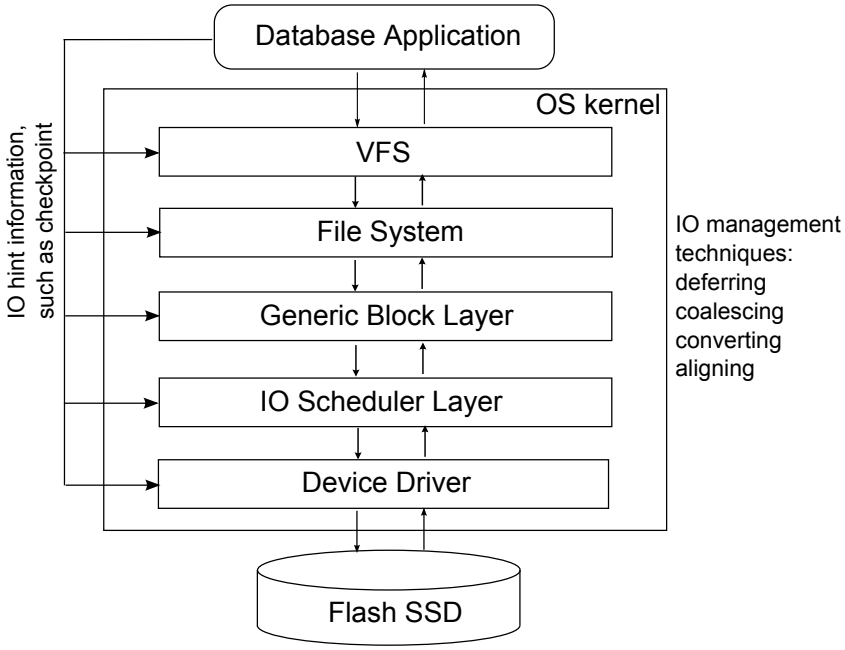


Figure 8.1: Implementation Options

and the checkpoint overhead limit can be a flexible tuning knob for the scheduler. Therefore, making such a scheduler can be a promising work in the future and applicable to many database systems.

Here I simply describe some implementation options. There are various options to implement the SSD-oriented scheduling techniques described in section 6.3.1. The IO path along the kernel shows that there are different optimizations for IOs at different layers, such as Virtual File System (VFS) layer, Block IO (BIO) layer and Device Driver layer[10], as shown in Figure 8.1. When considered for the SSD-oriented scheduler, it should be powerful and flexible to support different DBMSs' configurations. For instance, some DBMS supports raw device; the IO at the system call level is consistent with that at the device driver level, therefore, the SSD-oriented IO schedulering can be started from the system call level. Some DBMS requires support of file system, the scheduling can be enabled either in the file system layer when it is possible, or at the below device driver level in order to keep the file system layer untouched. In practice, the scheduler can be assembled in a kernel module, then loaded into a running kernel, and hereby the existing system keeps untouched.

Publication List

International Journal Articles

1. **Yongkun Wang**, Kazuo Goda, Miyuki Nakano and Masaru Kitsuregawa, *Performance Evaluation of Flash SSDs in a Transaction Processing System*, IEICE TRANSACTIONS on Information and Systems, Special Section on Data Engineering, March 2011 (To appear)

International Conference Publications

1. **Yongkun Wang**, Kazuo Goda, Miyuki Nakano and Masaru Kitsuregawa. *Early Experience and Evaluation of File Systems on SSD with Database Applications*. Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage (**IEEE NAS 2010**), pp.467-476 (2010.07).

2. **Yongkun Wang**, Kazuo Goda and Masaru Kitsuregawa. *Evaluating Non-In-Place Update Techniques for Flash-based Transaction Processing Systems*. Proceedings of the 20th International Conference on Database and Expert Systems Applications (**DEXA 2009**), pp.777-791 (2009.09).

Symposium and Workshop Publications

1. **Yongkun Wang**, Kazuo Goda, Miyuki Nakano and Masaru Kitsuregawa: *IO Path Management with Application Hint for Database Systems on SSDs*. The 3rd Forum on Data Engineering and Information Management (DEIM 2011), 2011.02.28, (To appear)

2. **Yongkun Wang**, Kazuo Goda, Miyuki Nakano and Masaru Kitsuregawa: *An Experimental Study on Basic Performance of Flash SSDs with Micro Benchmarks and Real Access Traces*. The 72nd National Convention of Information Processing Society of Japan (IPSJ), 6R-5, 2010.03

3. **Yongkun Wang**, Kazuo Goda, Miyuki Nakano and Masaru Kitsuregawa: *An Experimental Study on IO Optimization Techniques for Flash-based Transaction Processing Systems*. The Second Forum on Data Engineering and Information Management (DEIM 2010), E8-2, 2010.02

4. **Yongkun Wang**, Kazuo Goda and Masaru Kitsuregawa: *A Performance Study of Non-In-Place Update Based Transaction Processing on NAND Flash SSD*. The First Forum on Data Engineering and Information Management (DEIM 2009), E7-5, 2009.03

Bibliography

[1] Review: Intel X25-E 32GB SSD. http://www.bit-tech.net/hardware/storage/2008/12/17/intel-x25-e-32gb-ssd-review/1.

[2] AGRAWAL, D., GANESAN, D., SITARAMAN, R. K., DIAO, Y., AND SINGH, S. Lazy-adaptive tree: An optimized index structure for flash devices. *PVLDB 2*, 1 (2009), 361–372.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX ATC* (2008).

[4] ATHANASSOULIS, M., AILAMAKI, A., CHEN, S., GIBBONS, P. B., AND STOICA, R. Flash in a dbms: Where and how? *IEEE Data Eng. Bull. 33*, 4 (2010), 28–34.

[5] BAN, A. Flash file system. US Patent No. 5404485, April 1995.

[6] BITYUTSKIY, A. B. *JFFS3 design issues, Version 0.32 (draft)*, November 2005.

[7] BJØRLING, M., FOLGOC, L. L., MSEDDI, A., BONNET, P., BOUGANIM, L., AND JÓNSSON, B. T. Performing sound flash device measurements: some lessons from uflip. In *SIGMOD Conference* (2010), pp. 1219–1222.

[8] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *FAST* (2010), pp. 115–128.

[9] BOUGANIM, L., JÓNSSON, B. T., AND BONNET, P. uFLIP: Understanding Flash IO Patterns. In *CIDR* (2009).

[10] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel, Third Edition*. O' REILLY, 2005.

[11] CANIM, M., BHATTACHARJEE, B., MIHAILA, G. A., LANG, C. A., AND ROSS, K. A. An object placement advisor for db2 using solid state storage. *PVLDB 2*, 2 (2009), 1318–1329.

[12] CANIM, M., MIHAILA, G. A., BHATTACHARJEE, B., ROSS, K. A., AND LANG, C. A. Ssd bufferpool extensions for database systems. *PVLDB 3*, 2 (2010), 1435–1446.

[13] CAULFIELD, A. M., GRUPP, L. M., AND SWANSON, S. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS* (2009), pp. 217–228.

[14] CHEN, S. Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD* (2009), pp. 73–86.

[15] CHOI, H.-J., LIM, S. H., AND PARK, K. H. JFTL: A flash translation layer based on a journal remapping for flash memory. *TOS 4*, 4 (2009).

[16] CORPORATION, O. An Oracle White Paper: Exadata Smart Flash Cache and the Sun Oracle Database Machine.

[17] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[18] EMC. *White Paper: Leveraging EMC CLARiiON CX4 with Enterprise Flash Drives for Oracle Database Deployments Applied Technology*, December 2008.

[19] EMC. *White Paper: Leveraging EMC CLARiiON CX4 with Enterprise Flash Drives for Oracle Database Deployments Applied Technology*, October 2009.

[20] FOONG, A., VEAL, B., AND HADY, F. Towards SSD-Ready Enterprise Platforms. *First International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)* (September 2010).

[21] FREITAS, R., AND CHIU, L. Solid-state storage: Technology, design and applications. FAST Tutorial, `http://www.usenix.org/events/fast10/tutorials/T2.pdf`, 2010.

[22] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37*, 2 (2005), 138–163.

[23] GRAEFE, G. Write-Optimized B-Trees. In *VLDB* (2004), pp. 672–683.

[24] HDPARM. http://hdparm.sourceforge.net/.

[25] HEY, T., TANSLEY, S., AND IN TOLLE, K. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. 2009.

[26] HITACHI GLOBAL STORAGE TECHNOLOGIES. Deskstar 7K1000 Specifications. http://www.hitachigst.com/deskstar-7k1000.

[27] HOLLOWAY, A. L. *Adapting Database Storage for New Hardware. U.W. Madison PhD Thesis*, 2009.

[28] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal 40*, 3 (2001), 781–802.

[29] INTEL. Intel SSD Optimizer, White Paper. http://download.intel.com/design/flash/nand/mainstream/Intel_SSD_Optimizer_White_Paper.pdf.

[30] INTEL. Intel X25-E SATA Solid State Drive, Product Manual. http://download.intel.com/design/flash/nand/extreme/319984.pdf.

[31] INTEL. IntelR X25-E Extreme SATA Solid-State Drive, Technical specifications. http://www.intel.com/design/flash/nand/extreme/index.htm.

[32] JFFS2. *The Journalling Flash File System, Red Hat Corporation, http://sources.redhat.com/jffs2/*, 2001.

[33] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. Dfs: A file system for virtualized flash storage. In *Proc. of FAST* (2010), pp. 85–100.

[34] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A Flash-Memory Based File System. In *USENIX Winter* (1995), pp. 155–164.

[35] KIM, G.-J., BAEK, S.-C., LEE, H.-S., LEE, H.-D., AND JOE, M. J. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *VLDB* (2006), pp. 1255–1258.

[36] KIM, H., AND AHN, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *FAST* (2008), pp. 239–252.

[37] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (May 2002), 366–375.

[38] KIM, Y., TAURAS, B., GUPTA, A., NISTOR, D. M., AND URGAONKAR, B. FlashSim: A Simulator for NAND Flash-based Solid-State Drives. *Technical Report CSE-09-008, The Pennsylvania State University* (May 2009).

[39] KIM, Y.-R., WHANG, K.-Y., AND SONG, I.-Y. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *Proc. of SIGMOD* (2010), pp. 363–374.

[40] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The Linux implementation of a log-structured file system. *Operating Systems Review 40*, 3 (2006), 102–107.

[41] LEE, S.-W., AND MOON, B. Design of flash-based dbms: an in-page logging approach. In *Proc. of SIGMOD* (2007), pp. 55–66.

[42] LEE, S.-W., MOON, B., AND PARK, C. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 863–870.

[43] MAGHRAOUI, K. E., KANDIRAJU, G. B., JANN, J., AND PATTNAIK, P. Modeling and simulating flash based solid-state disks for operating systems. In *WOSP/SIPEW* (2010), pp. 15–26.

[44] MASTER, N. M., ANDREWS, M., HICK, J., CANON, S., AND WRIGHT, N. J. Performance Analysis of Commodity and Enterprise Class Flash Devices. *5th Petascale Data Storage Workshop(PSDW)* (November 2010).

[45] MTRON. Solid State Drive MSP-SATA7535 Product Specification, revision 0.3. `http://rocketdisk.com/Local/Files/Product-PdfDataSheet-35_MSP-SATA7535.pdf`, 2008.

[46] MYERS, D. *On the Use of NAND Flash Memory in High-Performance Relational Databases. Master's thesis, MIT*, 2007.

[47] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. I. T. Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys* (2009), pp. 145–158.

[48] NATH, S., AND KANSAL, A. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN* (2007), pp. 410–419.

[49] OCZ. OCZ Vertex EX Series SATA II $2.5''$ SSD Specifications. `http://www.ocztechnology.com/products/solid_state_drives/ocz_vertex_ex_series_sata_ii_2_5-ssd`.

[50] ORACLE. ORACLE EXADATA V2. http://www.oracle.com/us/products/database/exadata/index.htm.

[51] PARK, C., CHEON, W., KANG, J.-U., ROH, K., CHO, W., AND KIM, J.-S. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Trans. Embedded Comput. Syst. 7*, 4 (2008).

[52] PETROV, I., ALMEIDA, G., AND BUCHMANN, A. Building Large Storage Based On Flash Disks. *First International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)* (September 2010).

[53] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst. 10*, 1 (1992), 26–52.

[54] SAMSUNG CORPORATION. *K9XXG08XXM Flash Memory Specification*, 2007.

[55] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending ssd lifetimes with disk-based write caches. In *FAST* (2010), pp. 101–114.

[56] SYSTEMTAP. http://sourceware.org/systemtap/.

[57] TPC-C REPORT. Sun SPARC Enterprise T5440 Cluster with Oracle Database 11g with Real Application Clusters and Partitioning. http://www.tpc.org, January 2010.

[58] TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC). *TPC BENCHMARK C, Standard Specification,Revision 5.10*, April 2008.

[59] TSIROGIANNIS, D., HARIZOPOULOS, S., SHAH, M. A., WIENER, J. L., AND GRAEFE, G. Query processing techniques for solid state drives. In *Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 59–72.

[60] WANG, Y., GODA, K., AND KITSUREGAWA, M. Evaluating non-in-place update techniques for flash-based transaction processing systems. In *DEXA* (2009), pp. 777–791.

[61] WANG, Y., GODA, K., NAKANO, M., AND KITSUREGAWA, M. Early Experience and Evaluation of File Systems on SSD with Database Applications. In *IEEE NAS* (July 2010), pp. 467–476.

[62] WU, C.-H., KUO, T.-W., AND CHANG, L.-P. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embedded Comput. Syst. 6*, 3 (2007).

[63] YAFFS. *Yet Another Flash File System*, `http://www.yaffs.net`.