DEIM Forum 2011 D5-5

IO Path Management with Application Hint for Database Systems on SSDs

Yongkun WANG†, Kazuo GODA†, Miyuki NAKANO†, and Masaru KITSUREGAWA†

† Institute of Industrial Science, the University of Tokyo
4–6–1 Komaba, Meguro–ku, Tokyo 153–8505 Japan
E-mail: †{yongkun,kgoda,miyuki,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract Flash SSD is being incorporated in many enterprise storage platforms recently. Traditional storage systems are mainly composed of hard disks, and the IOs are optimized based on the characteristics of hard disks. In the case of the flash SSD, the performance characteristics are quite different from that of hard disk. Therefore, it is necessary that the existing system should be examined and tuned to maximize the performance benefit of flash SSDs. In this paper, we employed a small piece of information from the application and then optimized the IOs along the IO path for flash SSDs. The information from the application is captured in our optimization, we called it "application hint". Our experimental evaluation showed that the IO time can be considerably reduced, hereby the performance of whole system can be improved greatly.

Key words

1. Introduction

Flash SSDs are drawing more and more attention in the storage world recently. With the characteristics of excelent read performance, good sequential write performance, and low power consumption, it is being incorporated into the enterprise storage platform. However, the access characteristics of flash SSD are quite different from that of the traditional hard disks. Simply incorporating into the existing system may not maximize the performance benefit, because the current system has been tuned for a long time for hard disks.

There are a lot of publications to optimize the application performance on SSDs. Generally, these optimizations can be categrized into two types; one is to essentially changing the existing applications to fit for the flash SSD, such as [7] [8] , another is to keep the existing system untouched and ignore the application, only considering the IOs at the block level [11].

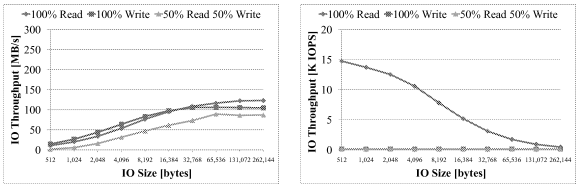
In this paper, we employed method of keeping the existing system untouched, but utilizing a small piece of information captured from the upper application, to optimize the IOs along the IO path. Our experimental evaluation shows that this method can improve the performance significantly.

The rest of this paper will organize as follow: section 2 will give a brief introduction to flash SSD. Section 3 will introduce the IO path management by application hint. Section 4 will provide a comprehensive experimental evaluation. Section 5 will give an extend discussion of our optimization method. The related work will be summarized in Section 6. Finally, our conclusion and the future work will be provided in Section 7.

2. Flash SSDs

NAND Flash memory is a kind of EEPROM (Electrically Erasable Programmable Read-Only 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. The data cannot be updated in place; when updating the data, the entire erase-block containing the data must be erased before the updated data is written in. This "erase-before-write" design leads to the relatively poor performance of random write.

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. Inside the flash SSD, the control system contains the mapping logic called Flash Translation Layer (FTL) which manages the address mapping, making the flash SSD appear to be a block device. The flash memory chips connect to the control system with multiple channels which can operate in parallel.



(a) Mtron Sequential (b) Mtron Random

Fig. 1 IO Performance of SSD

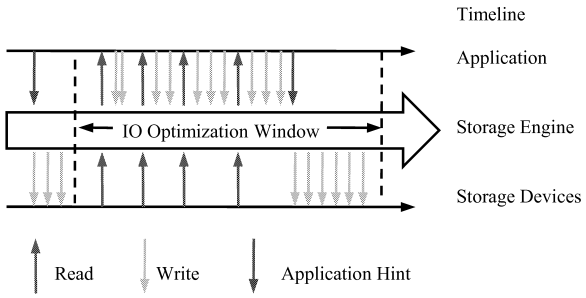


Fig. 2 IO Path Management with Application Hint

3. IO Path Management with Application Hint

3.1 IO Path Management

There has been a long history for the IO path optimization in the existing system, mainly due to the performance mismatch between the CPU and disk-based storage system. The system along the IO path is well tuned so that the IOs are well organized to maximize the performance of hard disk-based systems. The optimization techniques include the caching, merging, sorting and so on. The optimization can be implemented at the file system layer, block IO layer and IO scheduler layer. Some storage systems are much more complicated with multi-tier controllers, the IOs are further optimized in these controllers. These optimizations at different layers are usually separated from each other to avoid coupling and keep flexiblility.

In our design, we try to capture the a small piece of information from the application, we call it "Application Hint", then use it for the IO optimization in the subsequent layers along the IO path. We will introduce the application hint in detail in next section.

3.2 Application Hint

We call a certain information from the application the "Application Hint", which can be used as a reference for the optimization in the subsequent layers. Regarding the IOs in the data intensive applications, syncing the data and marking a checkpoint are good examples as the application hint. In this paper, we will utilize the checkpointing as the application hint to optimize the IOs.

The eligibility of the checkpointing as the application hint can be seen as following:

Until the checkpointing, the dirty data can be deferred in the buffer as long as possible, because the log has been written into the disk by the Write-Ahead-Logging (WAL) design. Therefore, the safty of dirty data is ensured to some extent by the WAL.

The deferred writes can be processed according to the characteristics of the storage media, hereby the IO performance can be improved. Once the system crashes, the deferred dirty data in buffer can be restored to some extent by the WAL.

The WAL design is currently employed in some systems, such as the database systems and journal-enabled file systems. We will focus on the database systems in this paper.

3.3 System Design

As shown in Fig. 2, we capture a small piece of information from the application, the application hint (checkpointing information), and do the optimization in our optimization system, then destage the IOs to persistent storage as soon as possible. The existing application keeps untouched.

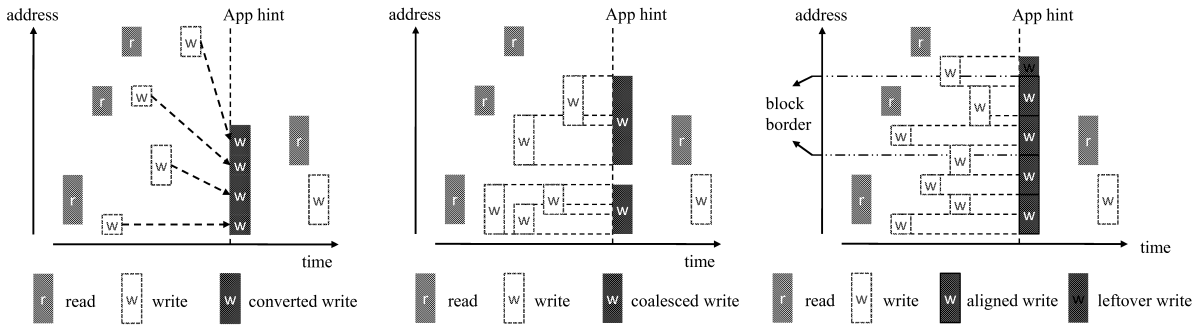
Only the IOs on data are optimized. The log IOs are keep untouched, the recovery and restore mechanism is still maintained by the database itself. The time for log writes is a small part of that of the total writes. In our experiments, the logging destination is different from that of data.

3.3.1 IO Optimization Techniques

- **Grouping** Group the reads and writes by deferring the writes until the application hint, as shown in Fig. 2. Therefore, the reads and writes are divided into groups separated by the application hint.
- **Converting** Convert the random IOs into sequential IOs. As shown in Fig. 3(a).
- **Coalescing** Merge the IOs which overlap or partial overlap with each other based on the application hint. As shown in Fig. 3(b).
- **Aligning** Combine and align IO requests into larger blocks, as shown in Fig. 3(c). The aligning is based on the Converting.

The slow erase operations are performed based on the unit of erase blocks. If a write request is across the border of two adjacent erase blocks, both of the erase blocks may possibly need to be perform the slow process of "copy data, apply changes, erase block, write back". The cost is not justified for a small write request. Therefore, we align the write requests into large and uniform request size. The cost of erase operations will be amortized. In section 4.3, we will see that another benefit of the aligning is that the device bandwidth is fully utilized.

In Fig. 3(c), the original requests are cut and assembled



(a) Converting (b) Coalescing (c) Aligning

Fig. 3 Optimization Techniques

into uniform large request size, which is dividable to the block size. There are two writes across the block borders, it has been cut into two pieces and aligned in different requests.

To maximize the benefits, the precedence of these optimization techniques, when using them together, is grouping firstly, then coalescing, next converting, and aligning at last.

The benefits for flash SSD is threefold:

（1） By address converting, random writes are converted into sequential writes, expensive erase operations are avoided or amortized. Obsolete data blocks will be cleaned by the garbage collection process at the background.

（2） Since the writes are deferred and flushed in batch, the reads are separated from writes, the "bathtub" effect is reduced. Therefore, the high pure read performance can be fully utilized.

（3） By aligning to large requests, the erase operations are avoided to some extent and amortized by large requests, and device bandwidth is fully utilized.

3. 3. 2 Implementation Options

There are various options to implement the optimization techniques with application hint described in section 3. 3. 1. The IO path along the kernel shows that we can have different optimizations at different layers, such as Virtual File System (VFS) layer, Block IO (BIO) layer and Device Driver layer. It is adaptive to the concrete DBMS. For instance, some DBMS supports raw device; the IO at the system call level is consistent with that at the device driver level, therefore, we can start the optimization from the system call level. We also can simply apply our optimization at the device driver level. In practice, the optimization can be assembled in a kernel module, then loaded into a running kernel, hereby the existing system keeps untouched.

4. Experimental Evaluation

In this secton, we firstly present the experiment setup, next provide the performance overview. Then we present our evaluation method, and the evaluation on each optimization techniques step by step, and finally the summary of the overall improvement and analysis.

Table 1 Configuration of DBMS

Data buffer size	8MB
Log buffer size	5MB
Data block size	4KB
Data file	5.5GB, database size is 2.7GB
Log flushing method	flushing log at transaction commit

4. 1 Experiment Setup

We built the TPC-C [9] benchmark system. We used a commercial(Com.) DBMS, the configuration is shown in Table 1. Log files are located to another dedicated flash SSD.

The commercial DBMS supports the raw device, so we set the data tablespace on the raw device, so that the OS buffer is bypassed, the database IO requests are consistent with the requests processed by the device. Therefore, we can do the optimization on the the DBMS requests and optimize it along the IO path.

4. 2 Performance Overview

We firstly present the transaction throughput, shown in Fig. 4. We can see that the transaction throughput is very close to each other among the cases of placing data on raw devices (raw dev), on device with ext2 configured with synchronous IO (ext2 sync), and on device with ext2 configured with asynchronous IO (ext2 asynch). Compared to the raw device and ext2 cases, the nilfs2 case (nilfs2 sync) can help to improve the transaction throughput at more than sixfold, which is consistent with the results reported by [12] [13].

4. 3 Evaluation Method

In order to clarify the influence of the IO response time in the total transaction processing time, we studied the response time of IO requests. Since the OS and file system buffer is exclude in the raw device case, the IOs are consistent between the system call layer and the device driver layer.

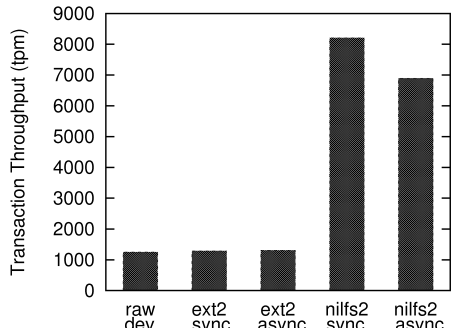


Fig. 4 Transaction Throughput of Comm. DBMS on Mtron SSD

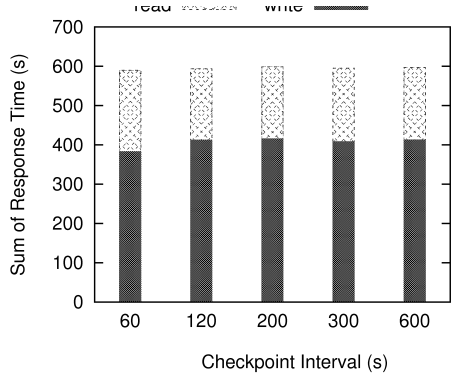


Fig. 5 Sum of the IO Response Time

Because the IOs can be queued deeply by multiple threads submitting the IO simultaneously, hereby the response time values of IO requests are overlapped each other. We then replay the IOs *one-by-one* (queue depth is one) on raw device with the same configuration, therefore, we obtained the individual response time of each requests. The summary of response time of each requests is the total time spent on the IO in the execution.

We captured the read and write requests on data in a period of 600 seconds in the raw device case with different checkpoint intervals, the replay these requests one-by-one with the same configuration. Fig. 5 shows the summary of the IO response time, which is very close to the total execution time (600s). This confirmed that the system is "IO Bound". Besides the data IO time, the rest time is used for the logging and other application behavior such as data processing or threads contention. It is clearly that if we reduce the IO time, the overall performance will be improved greatly. In the following sections, we will focus on using the application hint with IO optimization techniques to reduce the IO time, hereby improving the overall performance.

4.4 Evaluation of Optimization Techniques

In this section, we examined the performance benefit of each optimization technique step by step, then we provided the overall improvement with analysis.

4.4.1 Grouping

The purpose of the *Grouping* is (1) collecting the writes in batch and (2) separating the reads and writes to alleviate the "bathtub" effect. For (2), Fig. 6 shows the performance improvement on reads after grouping. The improvement is as expected, the speedup is 3.33-3.61x in all the cases.

The grouped writes will be further optimized as shown in the following sections.

4.4.2 Converting

We then studied the optimization of application hint with *Converting* discussed in section 3.3.1. The purpose of *Converting* is to convert the random writes into sequential writes, hereby (1) increase the throughput and (2) reduce the cost of erase operations. Fig. 7 shows the improvement is 46.08x to 53.07x.

4.4.3 Coalescing

The purpose of *Coalescing* is to reduce the amount of writes. Fig. 8 shows the performance improvement by applied IO optimization techniques, 1.52x to 2.02x. Coalescing reduced the duplicate write requests within the checkpoint. With the longer checkpoint interval, more writes can be coalesced, therefore, more improvement can be obtained.

4.4.4 Aligning

The purpose of the *Aligning* is to (1) reduce the cost of erase operations caused by the requests across the erase block border, and (2) fully utilize the bandwidth of device.

The aligning is performed on the basis of the converting optimization.

We choose 64KB alignment, the reason is that 64KB is size the SSD bandwidth beginning to get saturated, as shown in Fig. 1(a). So we can maximize the utilization of bandwidth, while keep the request size as smaller as possible.

With 64KB alignment, the further improvement compared to the coalescing is showing in Fig. 9, about 1.58x to 1.83x improvement can be observed.

4.4.5 Overall Improvement

We applied all the IO optimizations, that is, firstly grouping the IOs by application hint, then coalescing the IOs, next converting the random writes into sequential writes, finally aligning the IO requests into 64KB blocks.

As for the reads, the grouping has improved the read performance greatly. We find that a large amount of reads are repeated reads which can be served by the buffer (cache hit). Only the reads on distinct address reach the device (cache miss). We calculated the ratio of reads served by the buffer, as shown in Table 2, which is consistent with the results shown in [13].

Fig. 10 shows the total IO time after the full optimization, note that the y-axis is logarithmic. The total read speedup is 18.59x to 22.35x; speedup of writes is significant, from

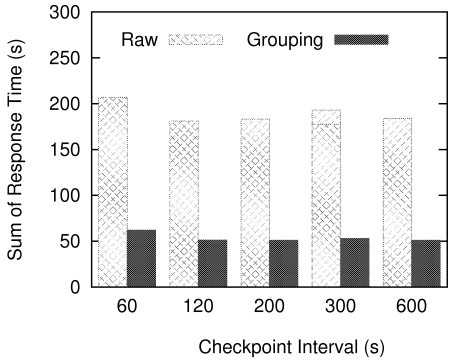


Fig. 6 Read Improvement by Grouping

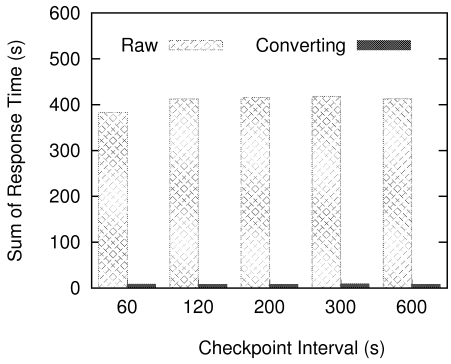


Fig. 7 Write Improvement by Converting

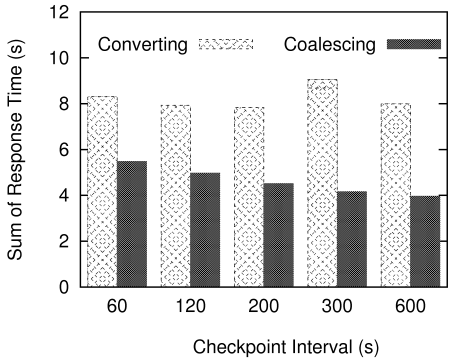


Fig. 8 Write Improvement by Coalescing

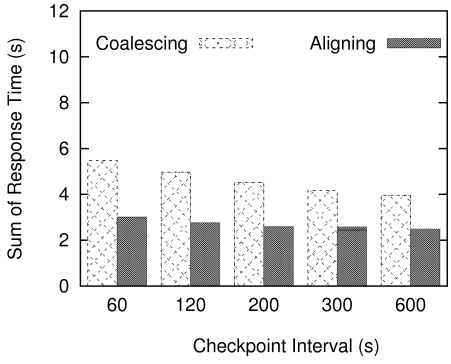


Fig. 9 Write Improvement by Aligning

Table 2 Cache hit ratio of reads ($\frac{R_c}{R_c+R_d}$, where R_c is read from cache, and R_d is read from device.)

	Checkpoint Interval (s)				
	60	120	200	300	600
Cache hit %	84.43	82.01	81.64	83.69	82.28

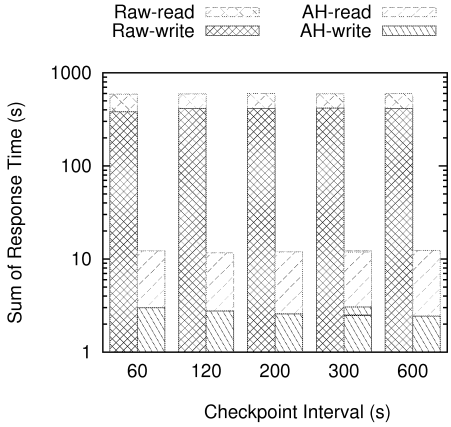


Fig. 10 Overall Improvement by Application Hint (AH), y-axis is logarithmic

128.03x to 169.24x; and the overall (read+write) speedup is considerable, from 48.15x to 51.41x.

5. Discussion

5.1 Logging and Recovery

The log are separated from data, and written to another dedicated flash SSD. The logging process is very fast since the write amount is small and writes are in a sequential manner. It has been confirmed in Fig. 5 that the logging time and IO time of other system files is a very samll part of the total IO time. The percentage is very small, so the infect to the overall speedup can be ignored.

About the recovery, once the system is crashed at some point, the database can still use the conventional recovery mechanism; reading the log data starting from the last checkpoint, and recovering those data blocks with log. With our optimization, the writes are deferred. Comparing with the traditional non-deferred writes, our solution may lost more dirty pages in buffer, hereby need to restore more data blocks and taking longer time to recover.

5.2 Buffer Usage

The buffer usage of deferred writes evaluated in section 4.4 is shown in Table 3. When the checkpoint interval changes from 60 seconds to 600 seconds, the maximum required buffer increased around 5.03x, however, the speedup only is improved 1.32x. Clearly the 60 seconds case is memory efficient.

5.3 SSD-specific Features

The wear-leveling is a special feature of the flash SSD. In

Table 3 Buffer Usage vs. Speedup

	Checkpoint Interval (s)				
	60	120	200	300	600
Minimum Required Buffer(MB)	31.33	60.58	95.43	135.89	196.84
Maximum Required Buffer(MB)	39.10	71.43	101.36	141.61	196.84
Speedup	128.03	149.14	161.60	160.00	169.24

our optimization, we do not intentionally consider the wear-leveling on the writes. Currently the SSD are well sealed by the manufacture with their own firmware (FTL) to do the address mapping and wear-leveling. It is hard for us to control each flash cell exactly. We believe that the "Converting" in our optimization can do some balancing on the writes to the flash cells, because the "Converting" will view the whole disk as a circular log and distribute the writes across the whole address space.

6. Related Work

6.1 Flash-based Technologies

By a systematical "Bottom-Up" view, the research on flash memory can be categorized as follow:

6.1.1 Hardware Interface

This is a layer to bridge the operating system and flash memory, usually called FTL (Flash Translation Layer). 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 using a simple but efficient page-to-page mapping [3] with a log-structured architecture [6]. However, it requires a lot of space to store the mapping table. In order to reduce the space for mapping table, the block mapping scheme is proposed, using the block mapping table with page offset to map the logical pages to flash pages [1]. 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 [4].

6.1.2 File System

Most of the file system for flash memory exploit the design of Log-structured file system [6] to overcome the write latency caused by the erasures. JFFS2 [2] is a journaling file system for flash with wear-leveling. YAFFS [10] is a flash file system for embedded devices.

6.1.3 Database System

Early design for database system on flash memory mainly focused on the embedded systems. FlashDB [7] is a self-tuning database system optimized for sensor networks, with two modes; disk mode for infrequent write and log mode for frequent write. LGeDBMS [5], is a relational database system for mobile phone. For enterprise database design on flash memory, In-Page Logging [8] is proposed. The key idea is to co-locate a data page and its log records in the same physical location. A block (page) level optimization is shown in [11]. Some evaluation works can be seen in [12] [13].

7. Conclusion and Future Work

With a small piece of information from the application, the application hint, we shows that the IOs can be optimized eagerly in the subsequent layers, hereby the performance can be improved significantly.

As for the future work, we plan to study the online optimization with application hint.

References

[1] Ban, A.: Flash file system. US Patent No. 5404485 (April 1995)

[2] JFFS2: The Journalling Flash File System, Red Hat Corporation, `http://sources.redhat.com/jffs2/jffs2.pdf`. (2001)

[3] Kawaguchi, A., Nishioka, S., Motoda, H.: A Flash-Memory Based File System. In: USENIX Winter. (1995) 155-164

[4] Kim, J., Kim, J.M., Noh, S.H., Min, S.L., Cho, Y.: A space-efficient flash translation layer for CompactFlash systems. IEEE_J_CE 48(2) (May 2002) 366-375

[5] Kim, G.J., Baek, S.C., Lee, H.S., Lee, H.D., Joe, M.J.: LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In: VLDB. (2006) 1255-1258

[6] Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1) (1992) 26-52

[7] S. Nath, A. Kansal: FlashDB: dynamic self-tuning database for NAND flash. IPSN 2007:410-41

[8] S.W. Lee, B. Moon,: Design of flash-based DBMS: an in-page logging approach. In: SIGMOD Conference. (2007) 55-66

[9] TPC: Transaction Processing Performance Council: TPC BENCHMARK C, Standard Specification,Revision 5.10. (April 2008)

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

[11] Y. Kim, K. Whang, I. Song: Page-differential logging: an efficient and DBMS-independent approach for storing data into flash memory. SIGMOD 2010:363-374

[12] Y. Wang, K. Goda, M. Kitsuregawa,: Evaluating Non-In-Place Update Techniques for Flash-based Transaction Processing Systems. In: DEXA (2009), 777-791

[13] Y. Wang, K. Goda, M. Nakano, M. Kitsuregawa: Early Experience and Evaluation of File Systems on SSD with Database Applications. IEEE NAS 2010:467-476