2010 Fifth IEEE International Conference on Networking, Architecture, and Storage

Early Experience and Evaluation of File Systems on SSD with Database Applications

Yongkun Wang
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
yongkun@tkl.iis.u-tokyo.ac.jp

Kazuo Goda Miyuki Nakano Masaru Kitsuregawa
Institute of Industrial Science
The University of Tokyo
Tokyo, Japan
{*kgoda,miyuki,kitsure*}*@tkl.iis.u-tokyo.ac.jp*

***Abstract*—Flash SSDs are being incorporated in many enterprise storage platforms recently. However, the characteristics of the flash SSD are quite different from that of hard disk. The IO strategies in the existing systems should be carefully evaluated. This paper provides an evaluation on the flash-based transaction processing system. Two file systems, traditional in-place update-based file system and log-structured file system, are selected as the representative of two write strategies. Usually, the log-structured file system is believed to play better on flash SSDs. Our experiment shows that the performance results of two database applications are diverse with two file systems on different flash SSDs. We analyze the performance in different configurations. Based on the analysis, we provide our experience on building the flash-based database system to better utilize the performance benefits of flash SSDs.**

***Keywords*-Flash Memory; SSD; Database; Transaction Processing; File System**

I. Introduction

Flash SSDs are being incorporated into enterprise storage. Sun and Oracle demonstrated high possibilities of flash SSDs for enterprise database systems [1]. Many reports have clarified the superiority of flash SSDs over conventional hard disks in terms of access performance. Compared to the conventional hard disk, flash SSD has special IO characteristics. For example, there is no mechanical moving parts, hereby no seeking time. The flash memory cells must be erased before the data written in again, known as "erase-before-write", while the erase operation is a time-consuming process compared to the read and write operations. Therefore, the conventional IO optimizations and strategies which are mainly designed for disk-based systems, should be reconsidered for flash-based systems.

Several researchers have proposed flash-aware techniques for database systems [2][3]. Such elegant techniques look helpful for deriving the potential performance of flash SSDs in a experimental system. However, recent enterprise systems have many stacks through the IO path. Rewriting the code of database systems is often expensive, but unfortunately it may not help so much. It is very likely that flash SSDs will play a vital role in the storage systems in the future. But, all the current hard disks will not necessarily be replaced with flash SSDs. Rather, the storage system can become a mixture of performance-intensive flash SSDs and capacity-intensive hard disks. Upcoming new solid-state technologies such as PCRAM [4] may also be incorporated here. It may not be a good choice to rewrite database systems fully for flash SSDs.

If a particular optimization technique through the IO path can derive reasonably potential performance of flash SSDs, such a solution can be acceptable for a variety of systems. A naive approach is to employ the log-structured write strategy. As discussed in [5], the log-structured write strategy can be simply incorporated along the IO path for enterprise system to boost the performance of flash-based transaction processing system. The log-structured write strategy may be accepted as a solution to better utilized the flash SSD: the "erase-before-write" defect can be overcame to some extent.

In this paper, we choose an implementation of log-structured file system, compared with the traditional file system on flash SSDs. TPC-C benchmark results are obtained with two database management systems on three high-end flash SSDs. The log-structured file system is usually believed to have better IO performance on flash SSDs. We find that the experiment results are diverse: the transaction throughput on log-structured file system is not always better than that on traditional file system. We analyze performance difference in different configurations. Based on the analysis, we provide our experience on building the flash-based database system to better utilize the performance benefits of flash SSDs.

The rest of this paper will organize as follow: Section II will give a brief introduction to flash SSD. Section III will provide the basic performance study of several flash SSDs. Section IV will provide an evaluation of the high performance database system on flash SSDs. The related work will be summarized in Section V. Finally, our conclusion and the future work will be provided in Section VI.

II. 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 written in place. When updating the data, the entire erase-block containing the data must be erased before

978-0-7695-4134-1/10 $26.00 © 2010 IEEE
DOI 10.1109/NAS.2010.12

the updated data is written there. 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. The internal structure of the flash SSD can be shown by block diagram in Figure 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.

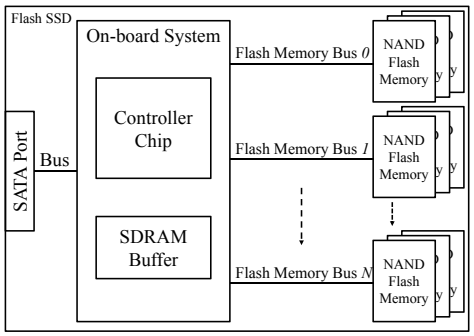


Figure 1. An example of internal structure of flash SSD

III. Basic Performance Study

A. System Setup

In this section, we present our basic performance study on three commercialized flash SSDs. Figure 2 illustrates our test system. We 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% random reads plus 50% random writes) to a target SSD. We let the benchmark tool bypass the file system and the operating system buffer in order to clarify the pure performance of the given SSDs. The effects of file systems and operating system buffers will be discussed in later sections. As target devices, we chose three typical high-end (relatively fast and reliable but having smaller capacity due to single-level voltage design) SSDs from the major product lines of Mtron, Intel and OCZ. These SSDs allow us to change their controller configurations, but we 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.

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

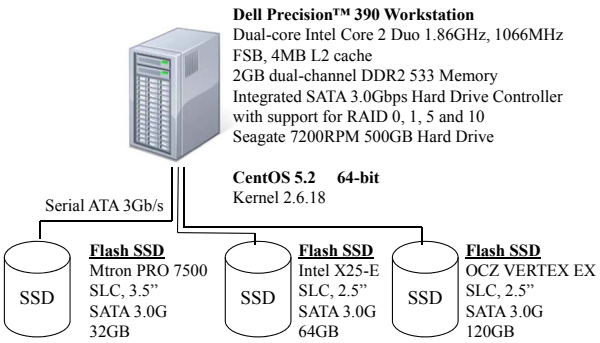


Figure 2. Experimental setup

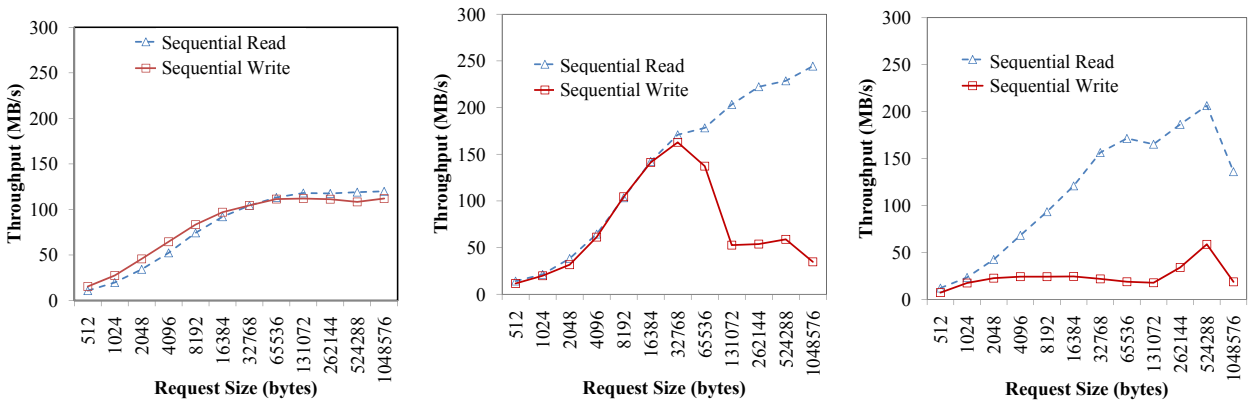
B. Basic IO Throughput

The basic IO throughput with different request size is shown in Figure 3 and Figure 4. Figure 3 shows the throughput when issuing the requests in a sequential way. In Figure 3(a), the Mtron SSD could present higher throughput for larger request sizes, but the throughput saturated around 120MB/s. The throughput between sequential read and sequential write is very close to each other. Figure 3(b) shows that the Intel SSD exhibits similar sequential throughput for read and write. However, when the request size is larger than 32768 (32K) bytes, the sequential write throughput starts to decrease and swing [2]. The max read throughput is much higher than that of Mtron SSD. In Figure 3(c), the disparity between sequential read and sequential write is clear. The sequential write throughput of OCZ SSD is much worse than that of the sequential read. High read throughput is also observed.

The random access performance is much more important for some applications, such as the transaction processing systems. Figure 4 shows the random throughput of each SSD. Figure 4(a) shows that the random throughput of Mtron SSD is different from the sequential throughput. The random write throughput decreases dramatically while the random read throughput keeps the same level as the sequential read throughput. We also tested with the mixed access pattern. The "Random %50Read & %50Write" shows the results with mixed requests composed of 50% read and 50% write: the mixed random throughput also decreases significantly. It is probably because the competition for on-board cache between reads and writes. As for the Intel and OCZ SSD, Figure 4(b) and Figure 4(c) confirm the similar performance gap between read and write respectively.

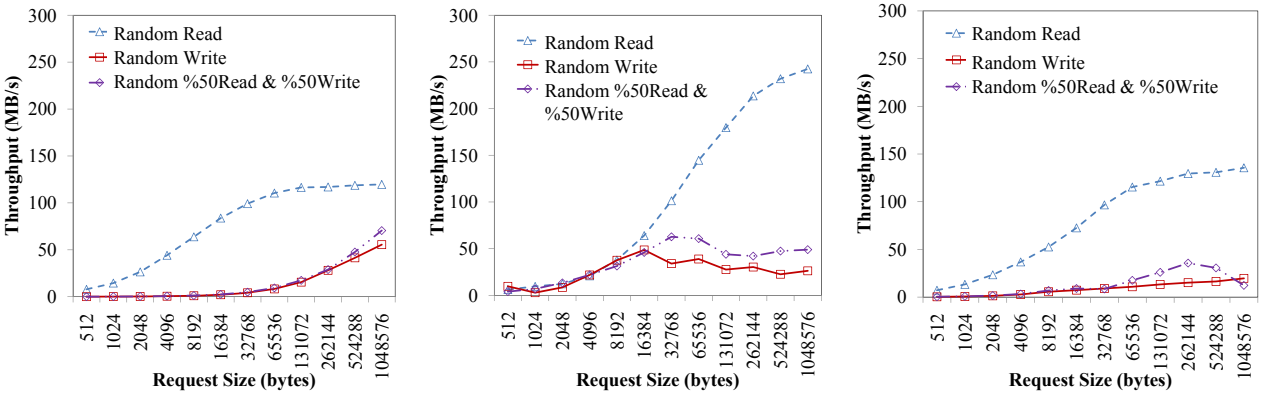
Combine Figure 3 and Figure 4, it is clear that the performance difference between sequential read and random read is not large. As a comparison, the performance difference between sequential write and random write is large. We can

[2]Similar behavior is observed when we disable the write-back cache. With write-through buffer policy, the decrease of throughput happens when the request is larger than 65536 (64K) bytes.



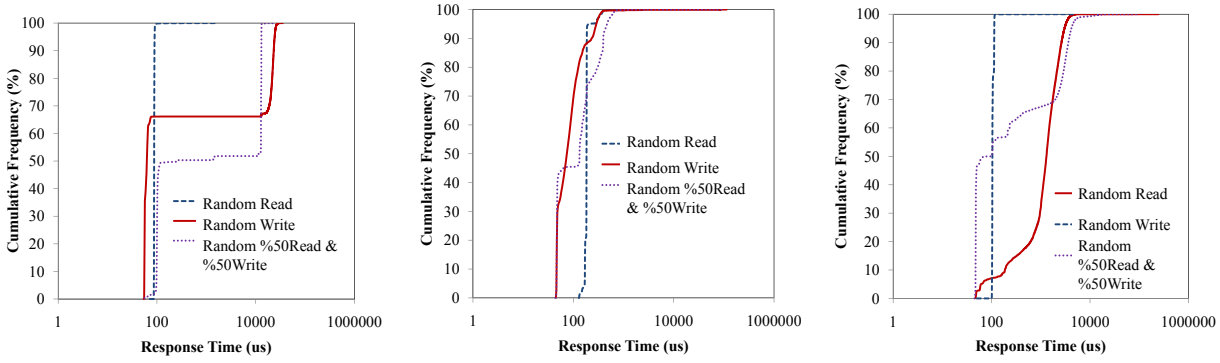
(a) Mtron PRO 7500 (b) Intel X25-E (c) OCZ VERTEX EX

Figure 3. Sequential access throughputs on three flash SSDs



(a) Mtron PRO 7500 (b) Intel X25-E (c) OCZ VERTEX EX

Figure 4. Random access throughputs on three flash SSDs



(a) Mtron PRO 7500 (b) Intel X25-E (c) OCZ VERTEX EX

Figure 5. Response time distribution of random accesses on three flash SSDs (4KB request size)

use the function

$$Gap(RequestSize)_{SSDname}^{Read/Write} = \frac{SequentialThroughput}{RandomThroughput}$$

to show the performance difference between sequential access and random access. The maximum value of Gap for each SSD can be calculated. For example, the $Gap(32768)_{Intel}^{Write} = 4.79$ is the maximum Gap value we can obtain from Figure 3 and Figure 4 for Intel SSD. The maximum Gap value of each SSD is different, it can help to find the optimal point for using the log-structured write strategy. The performance difference is also useful for the analysis of the transaction throughput on different IO strategies, as discussed in section IV-B.

C. IO Response Time

For deeper analysis, we traced response time for each IO in the measurement. Figure 5 shows the cumulative frequency distribution of the response time in case of random accesses of 4KB requests. We can have the following observations:

- "Mtron PRO 7500" clearly has the best random read performance: most of the "Random Read" response time is shorter than $100\mu s$ in Figure 5(a). As a comparison, most of the "Random Read" response time of "Intel X25-E" is longer than $100\mu s$ in Figure 5(b), while most of the "Random Read" response time of "OCZ VERTEX EX" is around $100\mu s$ in Figure 5(c).
- "Intel X25-E" has the best random write performance: most of the "Random Write" response time is shorter than $100\mu s$ in Figure 5(b). As a comparison, most of the "Random Write" response time of "OCZ VERTEX EX" is longer than $100\mu s$ in Figure 5(c), while about $1/3$ of the "Random Write" response time of "Mtron PRO 7500" is *much longer* than $100\mu s$ in Figure 5(a), and the rest $2/3$ is shorter than $100\mu s$.
- The values of "Random %50Read & %50Write" are close to that of "Random Write".

It is clearly that the "Mtron PRO 7500" is suitable for random read-intensive applications, "Intel X25-E" is suitable for random write-intensive applications, while the "OCZ VERTEX EX" has some trade-offs between the random performance of read and write.

In Figure 5, the response time distribution of random reads has a single cliff in all the three devices, while the distribution of random writes has multiple cliffs and the situations are different among the devices. In Figure 5(a), about $1/3$ of the random write response time is very distinct and much longer than the rest. In order to disclose the reason, we then examined the response time of each request in sequence. Figure 6 shows each response time of the 4KB random write request on Mtron SSD. At the bottom of the figure, the data points are very dense. Most of the values of the bottom data points are smaller than $100\mu s$. In the middle of the figure, many data points are scattered between $10000\mu s$ and $30000\mu s$, representing a very long response time. The long response time data points (between $10000\mu s$ and $30000\mu s$) are along with the short ones. Since the inside logic is not documented, our conjecture is that the long response time is caused by the flush operation by the "On-board System" described in Figure 1. When the "SDRAM 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.

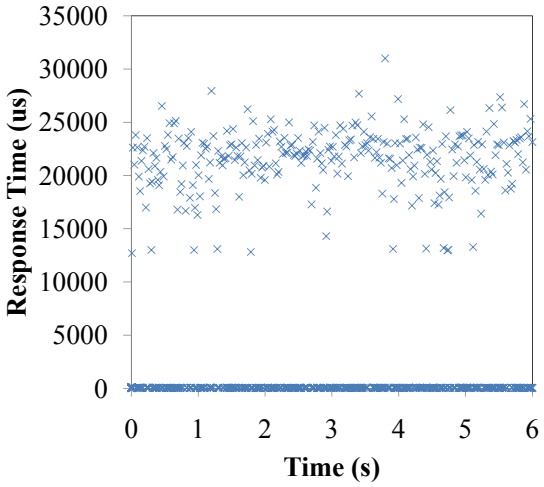


Figure 6. IO behavior of random write access on Mtron SSD(4KB Request Size)

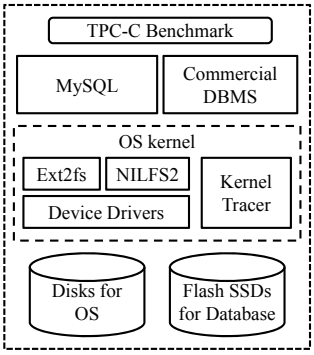


Figure 7. A transaction processing system on flash SSD

IV. Performance Evaluation of Different File Systems Using TPC-C Benchmark

A. Experimental setup

We built a database server on the system described in Figure 2. The configuration of the database system can be seen in Figure 7. We will describe it in a top-down manner.

We used the TPC-C [6] as the benchmark for the performance of transaction processing systems. TPC Benchmark

Table I
PERCENTAGE OF THE MIX OF TRANSACTION TYPE IN TPC-C BENCHMARK

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

C(TPC-C) is an OLTP workload. Although TPC-C cannot reflect the entire range of OLTP requirements, the performance results of TPC-C can provide important reference for the design of database system, thus it is accepted by many famous hardware and software vendors 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 very significant, hereby can be used to exploit the potentials of the new storage media, such as the flash SSDs. We started 30 threads to simulate 30 virtual users. 30 warehouses are created. The database size is 2.7GB. The "Key and Thinking" time was set to zero in order to get a hot workload. The mix of the transaction types followed the standards in [6], shown in the "normal" column of Table I, only the "New-Order" transaction is counted in the transaction throughput since it is the backbone of the workload. The transaction throughput is measured in *tpmC*, which means the number of "New-Order" transactions processed in one minute. Besides the "normal" mix of the transaction types shown in Table I, we also configure another two types of workload "read intensive" and "write intensive". The 86.96% of "read intensive" transactions consist of "Stock-Level" and "Order-Status" which are read-only transactions. On the other hand, the 96% of "write intensive" transactions are "New-Order" which has a lot of insertions and updates to the data tables.

As for the database applications, we chose a commercial DBMS and the popular open source DBMS MySQL. In the commercial database system, the default of buffer cache was set to 8MB, redo log buffer was 5MB, and the block size was 4KB. For logging, we set the behavior to immediately do flushing of the redo log buffer with wait when committing the transaction. The data file was fixed to a large size (5.5GB) in order to have a better sequential IO performance. All the IOs were set to be synchronous. For MySQL, we installed the *InnoDB* storage engine, the default of buffer cache was set to 4MB, log buffer was 2MB, and the block size was 16KB. The block size of MySQL was different from that of the commercial DBMS, because MySQL does not allow us to configure the block size, although 16KB might not be optimal. The data buffer pool size was 4MB, and log buffer was 2MB. We also fixed the size of data file instead of "autoextend". Synchronous IO behavior was chosen. For the flushing method of log, we set it to flush the log at transaction commit; for the flushing of data, we also used the synchronous IO.

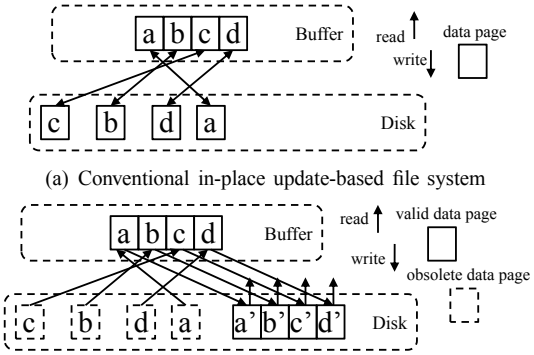


Figure 8. Two write strategies of modern file systems

We choose to use the file system to manage the data for database applications. The OS can incorporate different file systems to process the IOs from database application. Generally, there are two IO strategies of modern file systems, as illustrated in Figure 8. Figure 8(a) illustrates the concept of the conventional in-place update-based file system. The data pages may be scattered across the whole disk space. The seek operations are required when reading the pages into the OS buffer. When writing them back to the disk, the seek operations may also be required for each page, even though the pages may be continuous in the OS buffer. As a comparison, Figure 8(b) shows the mechanism of log-structured file system. The data pages are scanned and read into the OS buffer, applied changes, then written to the new address continuously. The obsolete data pages on disk will be recycled later by the garbage collection process. The seek operations are avoided for writes, hereby the overall performance can be improved for those write-intensive applications.

We evaluated two file systems, the conventional EXT2 file system and a log-structured file system, NILFS2 [7]. For EXT2, we set it with the default block size, that was 4KB blocks. For NILFS2, we set the block size to 4KB too, with 2KB blocks per segments. The segment size is influential to the garbage collection (GC, a.k.a segment cleaning) policy. By default we disabled it for the simplicity of analysis. The influence of the garbage collection have been discussed in [5].

The "Anticipatory" was chosen as the default IO scheduling algorithm. Three high-end flash SSDs are used to host the database and serve the requests via the block device driver.

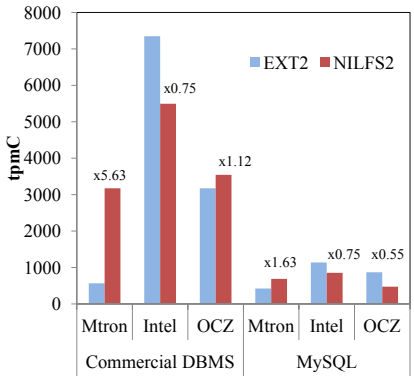


Figure 9. Overall transaction throughputs on different DBMSs, different file systems and different flash SSDs

B. Transaction Throughput

Transaction throughput is the basic measurement of the performance of the transaction processing system. Figure 9 shows the the transaction throughput of our test system. It is clear that the performance of Intel and OCZ SSD are much better in our test system. We think that there may be strong hardware support inside the Intel and OCZ SSD, such as the number of flash buses and size of on-board cache. With the help of the log-structured file system, the slow random write performance can be improved a lot on Mtron SSD. This is proved by the transaction throughput of Mtron SSD on Commercial Database with NILFS2: the transaction throughput is 5.63 times more than that of EXT2, and comparable to that of OCZ SSD.

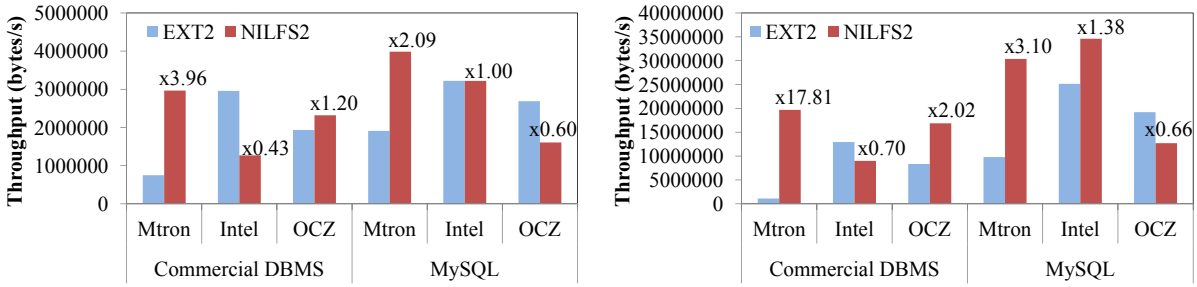
In order to analyze the performance difference, we also have a trace on the request size at the device driver level (device driver trace) where the requests are ready to be issued to the flash SSDs. The exact request size for each case will be provided along the analysis of the transaction throughput. The analysis of transaction throughput by three SSDs is as follow:

Mtron SSD The transaction throughput on log-structured file system (NILFS2) are much better than that on traditional file system (EXT2), about x5.63 (Commercial DBMS) and x1.63 (MySQL) can be observed in Figure 9. This is due to the different performance between sequential access and random access. The benefit of the log-structured file system comes from the great disparity between the sequential write and random write performance. Recall the performance of the basic IO throughput in section III-B, we can see that the sequential write performance of Mtron SSD in Figure 3(a) is much better than random write performance in Figure 4(b)when request size is smaller than 1MB. The device driver trace shows that the average write request size of Mtron SSD is about 14KB (14347 bytes, Commercial DBMS) and 102KB (104563 bytes, MySQL) respectively, where there is a big advantage of sequential write performance over random write performance. Therefore, the overall transaction throughput can be improved dramatically when converting the random writes into sequential writes by NILFS2. The average request size of MySQL (102KB) is much long than that of commercial database (14KB). We can clearly see from Figure 3(a) and Figure 4(a) that the Gap value between sequential write and random write decreases with the increasing of request size. That is the reason why speedup of the transaction throughput on MySQL is smaller than that of commercial database.

Intel SSD The transaction throughput of log-structured file system (NILFS2) is worse than that on traditional file system EXT2. Recall that the sequential write performance drops significantly after request size is larger than 32KB in Figure 3(b). The device driver trace shows that the average request size is about 21KB (21142 bytes, Commercial) and 176KB (180304 bytes, MySQL) respectively, where the sequential write performance (Figure 3(b)) is only about two times better than the random write performance (Figure 4(b)) ($Gap \approx 2$). Theoretically, the transaction throughputs of log-structured file system could still gain around twofold over the traditional file system. We will learn in section IV-G that this implementation of log-structured file system introduced more than twofold writes to provide additional function which cannot be disabled. Therefore, the performance gain is totally diminished.

OCZ SSD The transaction throughput of log-structured file system (NILFS2) is a little bit better (x1.12) than that of traditional file system (EXT2) on commercial database system, but much worse (x0.55) on MySQL. The device driver trace shows that the average request size of OCZ SSD is about 13KB (13190 bytes, Commercial DBMS) and 173KB (177404 bytes, MySQL) respectively. Figure 3(c) shows that the sequential write performance, though very low compared to read, is at the peak between 8192 (8KB) and 16384 (16KB). It can be safely estimated that with 13KB request, the sequential performance is more than two times better than that of random write performance. Therefore, the transaction throughput of this log-structured file system could be a little bit better than that of traditional file system though around twofold additional IOs (discussed in section IV-G) are introduced. Figure 3(c) also shows that there might be the lowest point for sequential write performance between 131072 (128KB) and 262144 (256KB). Therefore, for the 173KB request size, the performance gain between the sequential write performance (Figure 3(c)) and random write performance (Figure 4(c)) is less than two. Therefore, with more than two times additional IOs introduced by NILFS2, it is easy to understand that the transaction throughput on NILFS2 is worse than that of EXT2.

To sum up, we find that the request size is kind of important for the speedup of the log-structured file system to traditional file system. Long write request size may not be always superior on the SSDs. This should be considered



(a) Read Throughput

(b) Write Throughput

Figure 10. IO throughputs on different DBMSs, different file systems and different flash SSDs

for the design of the system with the log-structured write strategy. For example, the *Gap* value calculated in section III-B shows that it may be better to tune the system to use 32768 bytes (32KB) requests for Intel SSD to utilized the sequential write performance on the basis of the full consideration of the overall transaction throughput. Although there is influence from reads because the reads may compete the cache as discussed by the "Random %50Read & %50Write" in section III-B, the reads are usually not dominant in the transaction processing system and are mostly absorbed by the system buffer as we observed in section IV-G. Therefore, the *Gap* value may be helpful to confirm the benefit of log-structured file system on flash SSDs. With the existence of the performance asymmetry between sequential write to random write, we think the log-structured write strategy could be more beneficial for flash SSD. We will show the superior points by more experiments and analysis in the following sections.

C. IO Throughputs

Figure 10 shows the IO throughput for each case in Figure 9. The IO throughput confirms the transaction throughput and validates the analysis in section IV-B.

D. IO Behavior

A device driver level trace in Figure 12 shows the difference of the address distribution between the traditional file system EXT2 and the log-structured file system NILFS2. In Figure 12(a), writes are in series from the lower addresses to the upper addresses, and repeating with the reads between each series of writes. The serialized writes shows that the writes of Commercial DBMS on EXT2 are organized to approximately sequential writes with some help of "Anticipatory" scheduling algorithm. As comparison, Figure 12(b) shows the access pattern of Commercial DBMS on NILFS2: the addresses of writes are in sequence.

The address distribution in Figure 12 shows that the writes in the traditional file system are ordered with some help from the IO scheduler, so that it appears to be approximately sequential in each series. However, the writes are limited in the address space of the file, so that it will frequently go back and update the data from the starting address of the file, causing the erase operations for each series. As a comparison, writes of log-structured file system are in a "copy-on-write" manner to the whole disk space, thereby the frequency of erase operations may be lower than that of traditional file system. Therefore, the address distribution shows that the flash SSD would be quite favorable on the log-structured file system.

E. Workload Property Effects

We tested with another two types of workload, "read intensive" (86.96% of transactions are read-only) and "write intensive" (98% of transactions are read-write) shown in Table I. Since the log-structured file system favors the write-intensive workload, the performance speedup of write-intensive workload should be much better than that of the "normal" and "read intensive" workload. Figure 11 validates our expectation on both Commercial DBMS and MySQL: speedup increases in all the "write intensive" cases, and decreases in all the "read intensive" cases.

F. Database Buffer Effects

The database buffer size will have an essential influence on the performance. We configure 5.5GB datafile (2.7GB data), while the default buffer size in our test is small: 8MB (Commercial DBMS) and 4MB (MySQL). Figure 13 shows that the transaction throughput increases with large database buffer, and is saturated at about 512MB (about 10% of the datafile size). The performance speedup of log-structure file system to traditional file system decreases when increasing the database buffer size. Since more writes are cached in the database buffer, the advantage of the log-structured file system cannot be fully utilized.

G. IO Workloads Analysis

In order to see the difference of IO processing between this two file system, we traced the number of requests to file system issued by the DBMS (default buffer settings) and the number of requests processed by the flash SSD at the device driver level. Theoretically, the IOs per transaction between EXT2 and NILFS2 should be close to each other. Figure

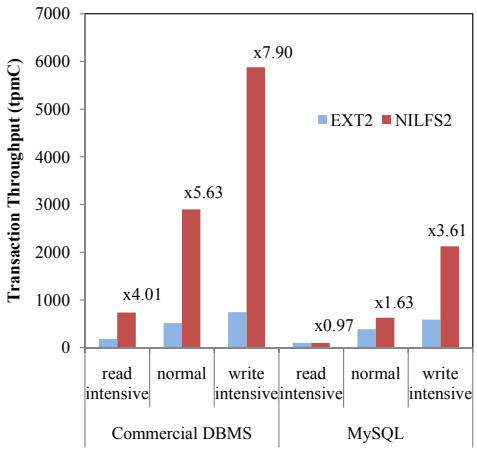


Figure 11. Workload property effects on transaction throughput on commercial database , Mtron SSD

14 shows the captured IOs per transaction on traditional file system. We can see in Figure 14(a) that the read requests of "Issued by DBMS" are greatly reduced compared to that of "Processed by flash SSD", which manifests that a large amount of the read requests are served by the cache. As a comparison, the amount of writes in Figure 14(b) are almost equal between "Issued by DBMS" and "Processed by flash SSD". This is because we use the synchronous IO for the database. Similar results on read requests of log-structured file system can be observed in Figure 15(a), this is as expected. As for writes in Figure 15(b), we find that amount of writes of "Processed by flash SSD" is more than twofold as many as that of "Issued by DBMS". We checked the newly created NILFS2 blocks on disk after 30,000 transactions, finding that around 50% of the new blocks are not used for data. These additional blocks store the B-Tree nodes and metadata. A large amount of additional blocks hinder the performance speedup. We believe that a log-structured file system aiming at the performance can gain much more on flash SSDs for those applications in which the small and random writes are dominant.

V. Related Work

A. Log-structured File System

The Log-structured file system (LFS) [8] is designed to exploit fast sequential write performance of hard disk, by converting the random writes into sequential writes. However, the side effect is that the sequential reads may also be scattered into random reads. Overall, the performance can be improved to write-intensive applications. The LFS is also expected to improve the random write performance of flash memory, since the fast read performance of flash memory well mitigates the side effect. For the garbage collection of LFS, an adaptive method based on usage patterns is proposed in [9].

B. Flash-based Technologies

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

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 [10] with a log-structured architecture [8]. 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 [11]. 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 [12].

2) File System: Most of the file system for flash memory exploit the design of Log-structured file system [8] to overcome the write latency caused by the erasures. JFFS2 [13] is a journaling file system for flash with wear-leveling. YAFFS [14] is a flash file system for embedded devices.

3) Database System: Early design for database system on flash memory mainly focused on the embedded systems. FlashDB [3] 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 [15], is a relational database system for mobile phone. For enterprise database design on flash memory, In-Page Logging [2] is proposed. The key idea is to co-locate a data page and its log records in the same physical location.

VI. Conclusion and Future Work

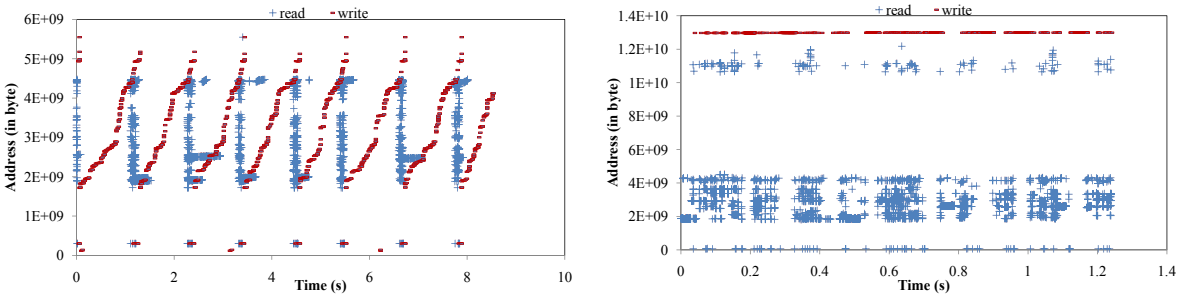
In order to better utilize the flash SSD in enterprise storage system, we provide an evaluation on the transaction processing system between two file systems as the representative of two IO strategies: in-place update and log-structured write strategy. Several enterprise-class flash SSDs are selected to be evaluated. The overall transaction throughput and IO behaviors are analyzed from several aspects. On the basis of analysis, we provide our experience on building the flash-based database system to better utilize the performance benefits of flash SSDs.

As for the future work, we plan to design the experiment system to maximize the performance gain by log-structured write strategy.

References

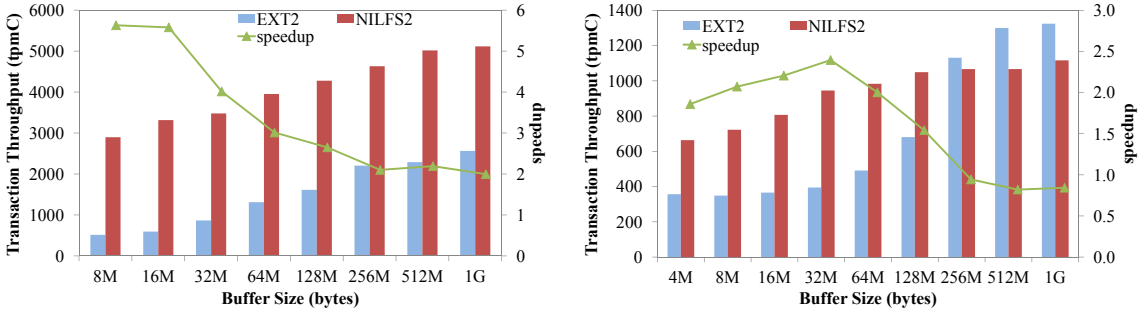
[1] TPC-C Report, *Sun SPARC Enterprise T5440 Cluster with Oracle Database 11g with Real Application Clusters and Partitioning*, January 2010. [Online]. Available: www.tpc.org

[2] S.-W. Lee and B. Moon, "Design of flash-based DBMS: an in-page logging approach," in *SIGMOD Conference*, 2007, pp. 55–66.



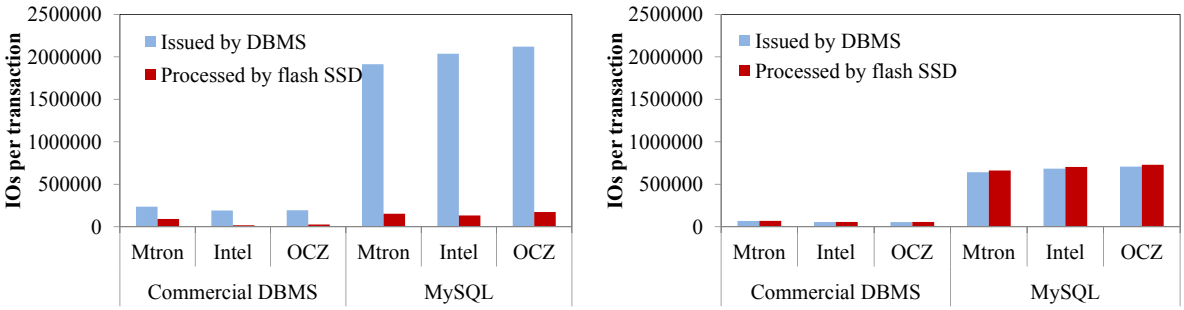
(a) Commercial DBMS on EXT2 (b) Commercial DBMS on NILFS2

Figure 12. The address of IOs transfered to Intel SSD by Commercial DBMS at device driver level



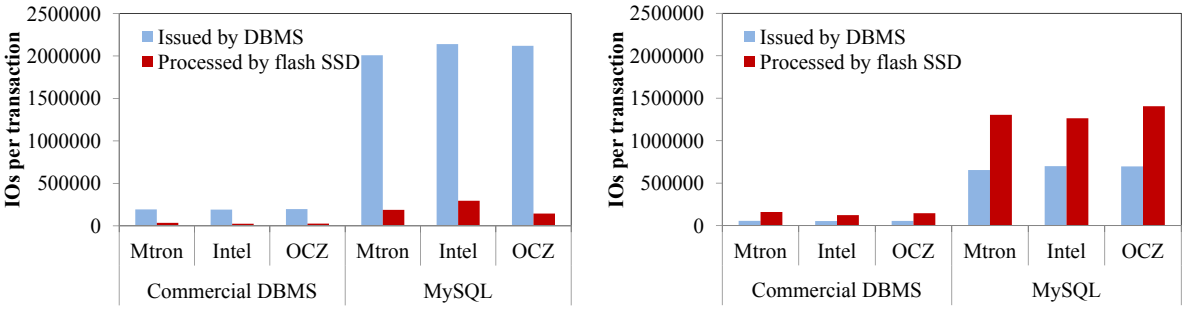
(a) Commercial DBMS (b) MySQL

Figure 13. Database buffer effects on transaction throughput on Mtron SSD



(a) Read (b) Write

Figure 14. IO workloads analysis on EXT2 and different flash SSDs



(a) Read (b) Write

Figure 15. IO workloads analysis on NILFS2 and different flash SSDs

[3] S. Nath and A. Kansal, "FlashDB: dynamic self-tuning database for NAND flash," in *IPSN*, 2007, pp. 410–419.

[4] A. Pirovano, A. Redaelli, F. Pellizzer, F. Ottogalli, M. Tosi, D. Ielmini, A. L. Lacaita, and R. Bez, "Reliability study of phase-change nonvolatile memories," *IEEE_J_DMR*, vol. 4, no. 3, pp. 422–427, Sept. 2004.

[5] Y. Wang, K. Goda, and M. Kitsuregawa, "Evaluating non-in-place update techniques for flash-based transaction processing systems," in *DEXA*, 2009, pp. 777–791.

[6] Transaction Processing Performance Council (TPC), *TPC BENCHMARK C, Standard Specification,Revision 5.10*, April 2008. [Online]. Available: www.tpc.org

[7] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The Linux implementation of a log-structured file system," *Operating Systems Review*, vol. 40, no. 3, pp. 102–107, 2006.

[8] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[9] J. M. Neefe, D. S. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the Performance of Log-Structured File Systems with Adaptive Methods," in *SOSP*, 1997, pp. 238–251.

[10] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in *USENIX Winter*, 1995, pp. 155–164.

[11] A. Ban, "Flash file system," US Patent No. 5404485, April 1995.

[12] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE_J_CE*, vol. 48, no. 2, pp. 366–375, May 2002.

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

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

[15] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe, "LGeDBMS: A Small DBMS for Embedded System with Flash Memory," in *VLDB*, 2006, pp. 1255–1258.

[16] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX ATC*, 2008.