IEICE TRANS. INF. & SYST., VOL.E93–D, NO.6 JUNE 2010

LETTER

A Buffer Management Issue in Designing SSDs for LFSs*

Jaegeuk KIM[†a)], ***Member***, **Jinho SEOL**[†], ***and*** **Seungryoul MAENG**[†], ***Nonmembers***

SUMMARY This letter introduces a buffer management issue in designing SSDs for log-structured file systems (LFSs). We implemented a novel trace-driven SSD simulator in SystemC language, and simulated several SSD architectures with the NILFS2 trace. From the results, we give two major considerations related to the buffer management as follows. *(1)* The write buffer is used as a buffer not a cache, since all write requests are sequential in NILFS2. *(2)* For better performance, the main architectural factor is the bus bandwidth, but 332 MHz is enough. Instead, the read buffer makes a key role in performance improvement while caching data. To enhance SSDs, accordingly, it is an effective way to make efficient read buffer management policies, and one of the examples is tracking the valid data zone in NILFS2, which can increase the data hit ratio in read buffers significantly.

key words: *NAND flash memory, solid state disk, log-structured file system, storage device*

1. Introduction

A Solid State Disk (SSD) has been one of the most attractive storage devices. With recent technology development of capacities as well as lower prices, consumers are now starting to consider the possibility of using SSDs as their primary storage devices in their computers. This is also becoming a tempting option in many laptops where size, weight, and power consumption are all the concerns.

SSDs generally provide better performance than traditional magnetic disks by eliminating mechanical overheads such as large seek time, spin-up delay, and rotational delay in magnetic disks. On a detailed performance point of view, however, while read speeds of SSDs are much faster than traditional disks, write speeds, especially random write speeds, may not guarantee such a huge performance improvement. Because SSDs employ multiple flash chips, which have several operational characteristics that need to be taken into account. For example, a flash chip does not allow in-place updates, which means that the previous data cannot be overwritten at the same location without being erased first. Also, the erase unit is relatively larger than a flash page that is the unit of read and write operations. These differences limit the blind adoption of the existing file systems.

Manuscript received November 9, 2009.
Manuscript revised December 21, 2009.

[†]The authors are with the Computer Science Department, KAIST, Korea.

[*]This work was supported by the IT R&D Program of MKE/KEIT. [2010-KI002090, Development of Technology Base for Trustworthy Computing]

a) E-mail: jgkim@camars.kaist.ac.kr

DOI: 10.1587/transinf.E93.D.1644

Many file systems have been designed to hide the disadvantages of SSDs [1]. The most common approach is implementing a log-structured file system (LFS) [2]. Initially proposed LFS treats storage as a big circular log and write all updated data to the head of the log continuously. Since SSDs are particularly poor at random writes, this approach have been able to make write operations almost be sequential.

Recently, NILFS2 [3] has been designed and implemented as a log-structured file system, especially on Linux kernel basis. NILFS2 enhances the write mechanism of original log-structured file systems by applying modern file system technology, a B-tree structure. With this writing method, it achieves fast recovery time and high write performance. As the coverage of SSDs in use is extended rapidly, several benchmark tests have been conducted to ensure the benefit of adopting NILFS2 to SSDs, and the results have been shown that NILFS2 has better performance than other file systems in most of cases [4].

In this letter, we give an insight in designing the essential parts of an SSD, particularly when adopting NILFS2 as a file system. In order to accomplish the goal, *(1)* we introduce a new trace-driven SSD simulator that enables representing more in details of the SSD architecture than other simulators, and *(2)* analyze the IO trace of NILFS2 obtained from a benchmark test, then *(3)* explore several SSD architectures through various configurations.

2. A Trace-Driven SSD Simulator

We implemented a trace-driven SSD simulator in SystemC language [5], a C++-based language that models both hardware and software together. For exploring various SSD architectures in a short time, we designed our simulator abstractly in which an elapsed time is traced to the end just by summing the latencies virtually.

Figure 1 shows the overall architecture of our SSD simulator. The embedded processor with SRAM executes firmware such as the buffer manager and the FTL. The host interface is a device-side storage protocol such as SATA and PATA, which handles all read and write requests from the host machine. SDRAM and flash controllers translate user-level operations into a series of hardware operations. To reduce bus contention between the hardware components, we implemented two buses separately: control and data buses. While the control signals such as address and enabling signals are transferred through the control bus, only user-made

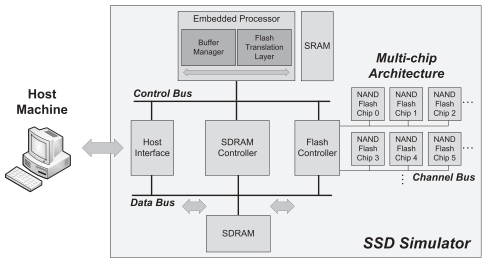


Fig. 1 Our SSD simulator architecture.

Table 1 Two types of NAND technology.

	SLC NAND [6]	MLC NAND [7]
Page Size (Bytes)	2048	4096
# of pages in an FEB	64	128
Spare Area Size (Bytes)	64	128
Read latency (μs)	25	60
Write latency (μs)	200	800
Erase latency (μs)	1500	1500

data are transferred through the data bus.

2.1 NAND Flash Memory

A NAND flash memory consists of a set of blocks called FEBs, and each FEB contains a number of pages. A page is a unit of read and write operations, and one FEB is a unit of erase operation. Additionally, each page keeps *spare area*, which is typically used for storage of an error correction code (ECC), and the remaining space may used to store metadata bytes.

Currently, there are two types of widely used NAND flash memory, such as SLC (Single-Level Cell) and MLC (Multi-Level Cell). Table 1 describes the general specification of the two NAND types. In the spare area of a SLC NAND chip, a few bytes (typically $12 \sim 16$ bytes) must be used for the ECC. In a MLC NAND chip, the spare area should be used most of bytes for the ECC due to the high bit error rate (BER) of the memory cells.

2.2 Multi-Chip Architecture

As the demand for larger capacity and higher performance is growing, SSDs have usually adopted a multi-chip architecture [8]. According to the bus configuration, the architecture are broadly classified into two categories: *shared control* and *shared bus* architectures [9]. In the shared control architecture, all flash chips have their own data path with a shared control path. In the shared bus architecture, several flash chips share a data bus and separates the control path.

A shared control architecture shows better performance than a shared bus architecture since it transfers multiple data in parallel. In the shared control architecture, however, the data path is implemented as a number of hardware wires, resulting in high implementation cost. In the shared bus architecture, the number of chips sharing one data bus should be determined carefully due to the bus contention overhead.

In order to enhance the multi-chip architecture in performance, another design, *multi-channel architecture*, has been adopted. A channel consists of a set of flash chips with one data bus, and general SSDs employ multiple channels to transfer data concurrently without the bus contention. Therefore, we design an SSD simulator on a basis of the multi-chip architecture in which adopts shared bus and multi-channel architectures.

2.3 Flash Translation Layer (FTL)

In order to emulate a block device interface, SSD has special management layer to hide the characteristics of NAND flash memory. Flash Translation Layer (FTL), a software layer of a storage device, is in charge of the management with two main functions: *(1) address translation* with mapping table between the storage space visible to the host and physical locations of NAND flash memory and *(2) garbage collection* that reclaims invalid pages by moving valid pages to another FEB and erasing the obsolete FEBs.

According to the mapping granularity, previous FTL schemes can be categorized as two-fold: page and block mapping schemes. A page mapping scheme, a fine-grained one, writes all logical pages to anywhere in NAND flash memory [10], which is profitable for many random writes that make small fragmentations. However, it consumes large memory resources to manage the whole page-level mapping information.

To reduce the memory limitation, a block mapping scheme, a coarse-grained one, has been proposed [11]. In this scheme, the mapping table is managed in a unit of FEBs, and each data is written to the fixed position in an FEB determined by its address. It is profitable for sequential writes since an FEB that has sequentially written pages can be switched without migrating valid pages. For random writes, further researches have been worked [12]. The trend is adopting a hybrid mapping scheme that a block mapping scheme is a primary basis applying a page mapping scheme appropriately.

2.4 Buffer Manager

In order to improve performance, SSDs generally adopt a buffer implemented by a small-sized DRAM that is relatively faster than NAND flash memory. By buffering data from a host and flushing the data to NAND flash memory later, the buffer can process a number of data requests concurrently during the flash operations. Furthermore, it can play a role of absorbing frequently accessed data as a *buffer cache*.

The buffer cache is controlled by a buffer manager operated as firmware. Especially, the policies of managing the buffer cache, such as how to select and replace victims, have a great impact on the FTL performance directly. The policies in most researches on SSDs [13]–[15] are focused on a write buffer except a read buffer since writing cost is much

more expensive than reading cost in NAND flash memory. For example, BPLRU [15] is focused on enhancing the random write performance particularly by filling a victim FEB with valid pages when flushing the FEB.

3. NILFS2 Workload

To analyze the behavior of NILFS2, we collected its IO trace that has the address space ranges from 0 to 1 GB. We used Postmark [16] benchmark as an user-level workload to make a random access pattern that is weak for SSDs. Postmark is one of benchmarks that perform intensive metadata operations. It consists of three phases. In our experiment, Postmark initially builds 2 subdirectories in the root directory. In the first CREATE phase, it creates 30000 files with 9 KB ∼ 15 KB of data in a randomly chosen subdirectory. In the second MIXED phase, 100000 mixed operations that consist of create/delete and read/append operations are performed randomly. Finally, in the DELETE phase, all the remaining files are deleted.

In NILFS2, there are two modules: a system module and a *cleaner*. The system module processes all read/write operations from applications by appending logs continuously. On the other hand, the cleaner, a user-level thread, reclaims the obsolete old data periodically. Therefore, a cleaning interval should be determined carefully to balance the speed gap between appending and reclaiming data. In our test, we set five seconds to the interval that is relatively short since the workload consists of IO-intensive operations.

Figure 2 shows apparently the characteristic of a log-structured file system. In the figure, all write requests are sequential, and a number of random read operations are reflected. Particularly, many write requests are induced by the cleaner. This tendency of the NILFS2 behavior is well-suited to SSDs that show high sequential write performance, and thus a trend adopting log-structured file systems for SSDs is highly recommended in the market place.

In such a trend, we need to consider differently all the directions of previous studies on SSDs. Concretely, we should rethink the policies to enhance performance for random writes since LFS makes sequential writes only. For this, we set our simulator as follows, basically optimized to the sequential write characteristic.

(1) To maximize the chip parallelism, the simulator adopts the multi-chip architecture that consists of 16 flash chips. Also, it adopts the multi-channel architecture, thus having 4 channels each of which has 4 flash chips equally.

(2) Among several FTL schemes, it adopts a simple block mapping scheme for the small-sized DRAM. Because the other mapping schemes are focused on random write performance intensively.

(3) For the intuitive buffer manager, we separate read and write buffers. The write buffers store the requested data intermediately in a First-In First-Out (FIFO) order, since most of the data face cold misses and further optimization policies for random writes are unnecessary.

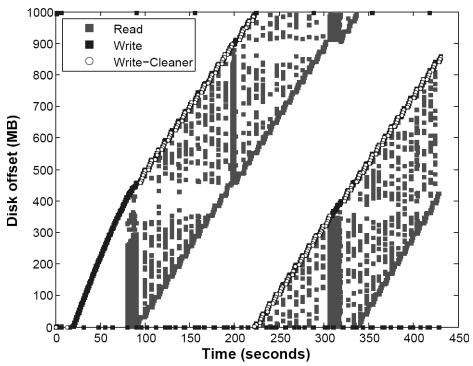
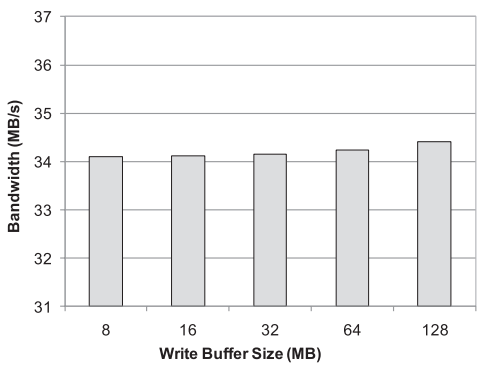


Fig. 2 The IO trace of NILFS2.

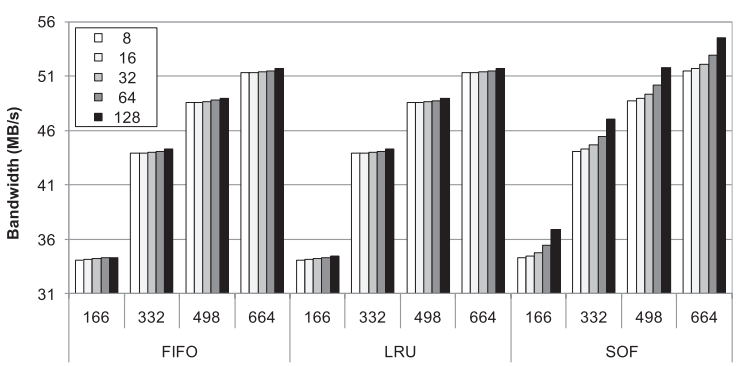
4. Simulation Results

Our simulator measures total elapsed time by summing up the bus, channel, and chip latencies internally, but the processor time is ignored since the firmware can be optimized by various styles. The chip latencies are based on the specification of MLC NAND type as described in Tabl 1. We observe the performance gaps according to the sizes of buffers, the bus bandwidth, and buffer replacement policies.

Figure 3 (a) shows the write buffer effect in size while the read buffer size is fixed to 8 MB. Note that the bus band-



(a) Write Buffer Effect



(b) Read Buffer (MB), Bus Bandwidth (MHz), and Buffer Replacement Policies

Fig. 3 SSD simulation results.

width is set to 166 MHz, and the channel bandwidth is set to 40 MHz. In the figure, the SSD bandwidth is nearly unchanged since all write requests are sequential.

Instead, to enhance the performance, random read requests should be handled efficiently. For this, we measured the performance according to the bus bandwidth, the read buffer size, and buffer replacement policies while the write buffer size is fixed to 8 MB. Note that the channel bandwidth is always set to the 1/4 bus bandwidth. We adopted three buffer replacement policies such as FIFO (First In First Out), LRU (Least Recently Used), and SOF (Smallest Offset First). In the SOF policy, a victim buffer is selected by its disk offset, and a buffer haing the smallest offset is flushed at first.

Figure 3 (b) shows the results in which x axis represents the bus bandwidth in MHz and bars indicate from 8 to 128 MB of read buffer sizes. As shown in the figure, the SSD bandwidth is mostly improved as the bus bandwidth increases, but it is saturated when the bus bandwidth exceeds 332 MHz. While FIFO and LRU policies show similar patterns, SOF shows much higher performance, since it takes into consideration the characteristic of existing the valid data zone that is starting from the last garbage collected disk offset to the current logging offset in NILFS2. Furthermore, the effect of SOF becomes large as the read buffer size increases.

5. Conclusion

Recently, SSDs are widely used in the world, and much effort to develop high performance SSDs have been conducted from architecture to operating system levels. In the mean time, many studies show that LFS is well-suited to SSDs, and thus, this letter gives a brief insight, particularly related to the buffer management, of SSD architectures especially for LFSs. From the simulation results, we conclude that the read buffer management is an important factor to improve the performance. In the concrete, although higher bus bandwidth improves the performance significantly, 332 MHz is enough. For better performance, therefore, it needs further researches to track the valid data zone in NILFS2 for caching data efficiently.

References

[1] A.I.A. Wang, G.H. Kuenning, P.L. Reiher, and G.J. Popek, "The Conquest file system: Better performance through a disk/persistent-RAM hybrid design," TOS, vol.2, no.3, pp.309–348, 2006.
[2] 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.
[3] 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.
[4] D. Shin, "About SSD," USENIX Linux Storage & Filesystem Workshop (LSF'08), 2008.
[5] SystemC Initiative, 2007.
[6] Samsung Electronics a., "2Gx8 bit NAND flash memory (K9WAG-08U1A)," 2006.
[7] Samsung Electronics b., "2Gx8 bit NAND flash memory (K9GAG-08U0M)," 2006.
[8] J.U. Kang, J. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance nand flash-based storage system," J. Syst. Archit., vol.53, no.9, pp.644–658, 2007.
[9] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M.S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," USENIX Annual Technical Conference, pp.57–70, 2008.
[10] M.L. Chiang, P.C.H. Lee, and R.C. Chang, "Using data clustering to improve cleaning performance for plash memory," Softw. Pract. Exper., vol.29, no.3, pp.267–290, 1999.
[11] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho, "A space efficient flash translation layer for compactflash systems," IEEE Trans. Consum. Electron., vol.48, no.2, pp.366–375, 2002.
[12] S.W. Lee, D.J. Park, T.S. Chung, D.H. Lee, S. Park, and H.J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," ACM Trans. Embed. Comput. Syst., vol.6, no.3, article 18, 2007.
[13] S.Y. Park, D. Jung, J.U. Kang, J. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," CASES, pp.234–241, 2006.
[14] H. Jo, J.U. Kang, S.Y. Park, J.S. Kim, and J. Lee, "FAB: Flash-aware buffer management policy for portable media players," IEEE Trans. Consum. Electron., vol.52, no.2, pp.485–493, 2006.
[15] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," FAST, pp.239–252, 2008.
[16] J. Katcher, "Postmark: A new file system benchmark," Report of Network Appliance Tech TR3022, 1997.