Extended Abstracts of the 2013 International Conference on Solid State Devices and Materials, Fukuoka, 2013, pp130-131

Co-Design of Application Software and NAND Flash Memory for Database Storage System

Kousuke Miyaji[1,2], Chao Sun[1,3] and Ken Takeuchi[1]

[1]Chuo University, Department of Electrical, Electronic, and Communication Engineering, E-mail: miyaji@takeuchi-lab.org

[2]Shinshu University, Japan, [3]University of Tokyo, Japan

Abstract

An optimized storage system is proposed by co-designing database (DB) application software and NAND flash memory based solid-state drive (SSD). The DB storage engine (SE) utilizes physical information about the NAND flash which is supplied from the flash translation layer (FTL) implemented in the SSD controller. Also, the query (DB unit command) is optimized for NAND flash memory's operation. By these treatments, page-copy-less garbage collection (GC) is achieved and data fragmentation in the NAND flash memory is suppressed. As a result, SSD performance increases by 3.8 times, power consumption decreases by 46% and write/erase (W/E) cycles decreases by 62%.

Introduction

Performance in big data applications, such as structured query language (SQL) DB [1] (Fig. 1), is generally limited by the storages' speed and reliability. A NAND flash memory based SSD is expected to be a key storage device to overcome this problem for its fast random access speed compared with HDD. In NAND flash memory, a page (cells sharing a same word-line) and block (a group of pages bounded by select gates) are the write and erase unit, respectively (Fig. 2(a)). The number of pages per block tends to increase as the technology node shrinks (Fig. 2(b)). Therefore, SSDs need to handle growing block sizes and write-erase size asymmetry. Since the present SQL DBs are originally designed for HDDs, a new DB system that considers the SSD property like above is required for exploiting the SSD potential.

A data unit in the DB is a row. Row data is stored in the SSD, and managed by SE, which responds to queries from the SQL server (Fig. 1). When a storage device is specifically used for DB application, its performance is greatly improved by bypassing the file system (FS), because the SE can directly control the storage and thus FS overhead can be removed [1]. However, in SSDs, the FTL inside the SSD controller manages parallel NAND flash operation, address translation, wear-leveling, GC and error correction. Particularly, the address translation and GC can cause unpredictable storage performance degradation, which is out of SE's control. Moreover, the DB row's data size of a few 100B is much smaller than NAND's page size of 16kB, which further aggravates data fragmentation and GC [2] (Fig. 3). In this paper, SE, FTL and NAND flash memory are co-designed to enhance overall DB system performance. Unlike the previous works [2,3], this performance improvement can be achieved without a storage class memory (ReRAM), which is still in developing phase.

Conventional Database Storage System Issues

The addressing hierarchy in the DB is shown in Fig. 4, in which SE manages the row data logical address (LA). In the FTL, since the write data unit in NAND is a page, LAs are mapped to the logical page addresses (LPAs). LPA is the quotient of the LA divided by the NAND page size. The NAND physical page location is defined as a physical page address (PPA). Due to NAND's W/E cycling limitation, LPA and PPA are translated in the FTL. Fig. 5 shows the physical write operation with NAND. When SE overwrites a given LPA, the FTL translates the target LPA to PPA and read the old data (step 1). Then, in the SSD controller (FTL), new data from SE is merged with the old data read from NAND (step 2). Since physical overwrite is not allowed in NAND, the merged data is written to a new PPA page in the NAND (step 3). The old PPA page in the NAND becomes invalid (step 4). After many writes, invalid pages in the SSD accumulate, and GC is triggered to reclaim free space (Fig. 6). In GC, the remaining valid pages in the old NAND block are read to the SSD controller (seq. 1, 2) and then written to a new NAND block (seq. 3). Seq. 1 to 3 is a page copy. Finally, the old block is erased (seq. 4). GC repeats page copy for each of the valid pages (N_{valid}) in the block-to-be-freed, possibly over 100 times. Thus, GC latency can be over 100ms, which is a severe SSD performance overhead. It will even get worse in the future SSD, since the number of pages per block is increasing as discussed in Fig. 2(b).

Proposed Database Storage System

In the proposed scheme, SE, FTL and NAND flash are co-designed to avoid page copy (Fig. 7). In the conventional scheme, the physical address layer is not visible to the SE. Writes are dispersed to all NAND flash blocks due to the policy of logical-physical translation in FTL. Therefore, valid pages could remain in the next erase block when GC starts. On the other hand, in the proposed scheme, SE concentrates writes to the block to be erased at the next GC. Therefore, when GC starts, there is no valid page left in the target block and page copy is unnecessary (Figs. 7,8). As a result, the SSD performance is enhanced.

The detailed SE operations for conventional and proposed scheme are shown in Fig. 9. *Insert*, *Delete* and *Update* are the query commands that add, eliminate and change the row data. Conventionally (Fig. 9(a)), during *Insert*, to avoid data fragmentation in LA space, the new row is appended to the last row. *Delete* disables a row in the LA space, which has no corresponding operation in the FTL (no NAND operation). *Update* modifies a row in-place for the same LA. Fig. 10(a) shows an example of the conventional SE and FTL response when "*Insert* → *Update* → *Insert*" are issued. As many rows are added or modified beforehand, the order of the LPA becomes random in physical address space. When an LPA becomes completely filled with row data (like LPA4, 12, 36), the corresponding PPA cannot be invalidated by subsequent *Insert* since there is no free space in that LPA. These filled up LPA (PPA) pages remain valid unless the row data are overwritten by *Update* or an empty row space is created by *Delete*. When SSD stored data is increased, the number of the filled up pages also increases. Hence, N_{valid} can be large during GC, inducing huge page copy times.

In the proposed scheme, two techniques are introduced (Fig. 9(b)). First, *Insert Address Assist* (IAA) allows SE to understand LPA-level situations by the following information received from FTL, (i) page size of NAND flash memory to calculate corresponding LPA of the row LA to be written, (ii) LPAs of the remaining valid pages (PPAs) that are in the target next block to be erased. Thus, SE can write row data to the free space in the corresponding LPAs to invalidate the pages whenever *Insert* is issued. By these interactions between the SE, FTL and NAND flash, invalid pages can be concentrated in the next block to be erased. Also, for each *Insert*, the target LPA is changed (ex. LPA0→ LPA1→LPA2…) to evenly distribute data over all the available pages, and thus avoid the filled up page situation. Second, *Update* is modified to a *Delete* and *Insert* (*Update* to *Delete* + *Insert*: U2DI) sequence. The updated row data can be moved to the target LPA in the next erase block by this sequence. Fig. 10(b) shows an example of the proposed SE and FTL response. Since Block 0 is the next erase block, FTL informs LPAs that are allocated to PPA0~3, therefore LPA4~7, to SE by IAA. By combining IAA and U2DI, the proposed storage can invalidate all PPAs in the next erase block and thus avoid the page-copy penalty, regardless of the query access patterns.

Results

A DB system is developed by implementing SE in MySQL [1], and using a transaction-level-modeling-based SSD emulator for the FTL and NAND flash [2,3]. Fig. 11 shows the SSD performance, energy consumption and W/E cycles as a function of SSD free space during a pattern of random queries of *Insert*, *Delete* and *Update*, with varying probabilities. In the conventional scheme, as the SSD fills up and free space decreases, performance, energy consumption and W/E cycles degrade, because page copy increases as the number of valid pages with full row data is increased. The extra writes from page copy increases energy consumption and W/E cycles. In contrast, no degradation is observed in the proposed scheme, due to the page-copy-less GC operation (See also Fig. 8). The proposed scheme has 3.8 times higher performance, 46% less energy consumption and 62% less W/E cycle, when the SSD free space is 20%. Fig. 12 shows the individual and combined effects of the proposed IAA and U2DI. IAA and U2DI should be implemented together to maximize the performance for any amount of SSD free space.

Conclusion

SE in SQL DB, SSD FTL, and NAND flash are co-designed for the DB storage system. The proposed IAA and U2DI scheme achieve page-copy-less GC. SSD performance increases by 3.8 times, power consumption decreases by 46% and W/E cycles decrease by 62% .

Acknowledgement This work is partly supported by NEDO.

References [1] http://www.mysql.com [2] H. Fujii et al., *Symp. VLSI Circ.*, pp. 134-135, 2012. [3] C. Sun et al., *NVMTS*, pp. 87-88, 2012.

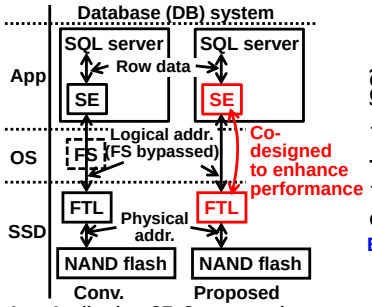

Fig. 1 DB system architecture.

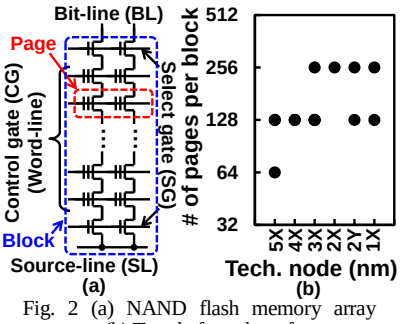

Fig. 2 (a) NAND flash memory array structure. (b) Trend of number of pages per block in NAND flash memory.

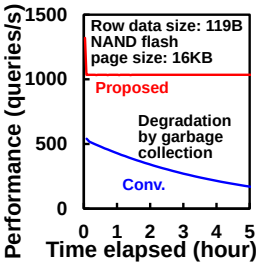

Fig. 3 SSD performance vs. elapsed time. Performance degrades in conventional system by data fragmentation and garbage collection (GC).

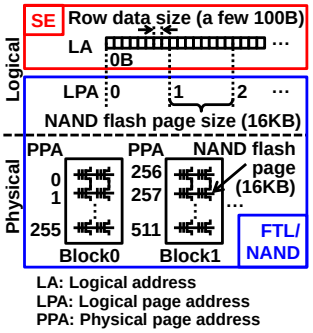

Fig. 4 Logical and physical data structures and addresses.

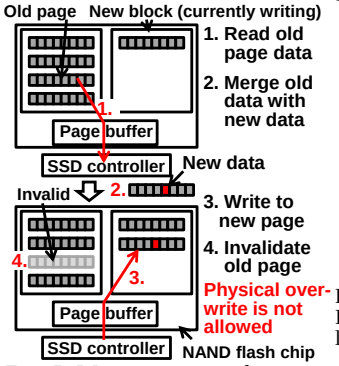

Fig. 5 Write operation of SSD (NAND flash).

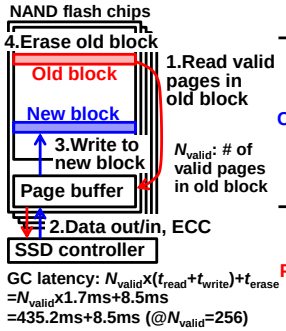

Fig. 6 Conventional GC. If N_{valid} is large, GC latency is over 100ms.

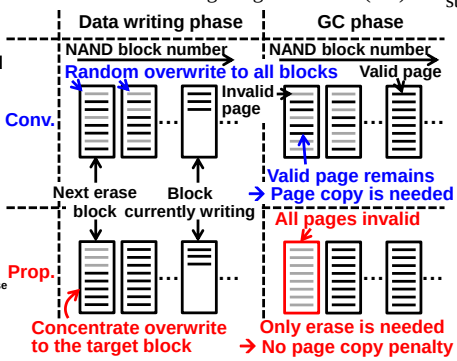

Fig. 7 Concept of the proposed page-copy-less GC scheme.

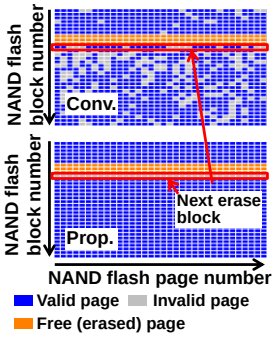

Fig. 8 Valid page location in the emulated NAND flash.

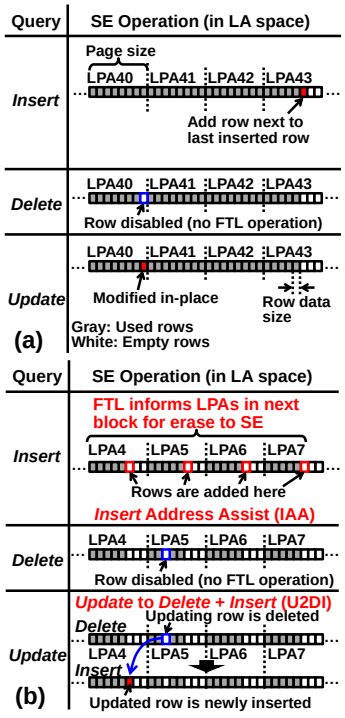

Fig. 9 SE operations performed in LA space for *Insert*, *Delete* and *Update* SQL queries in the (a) conventional and (b) proposed scheme.

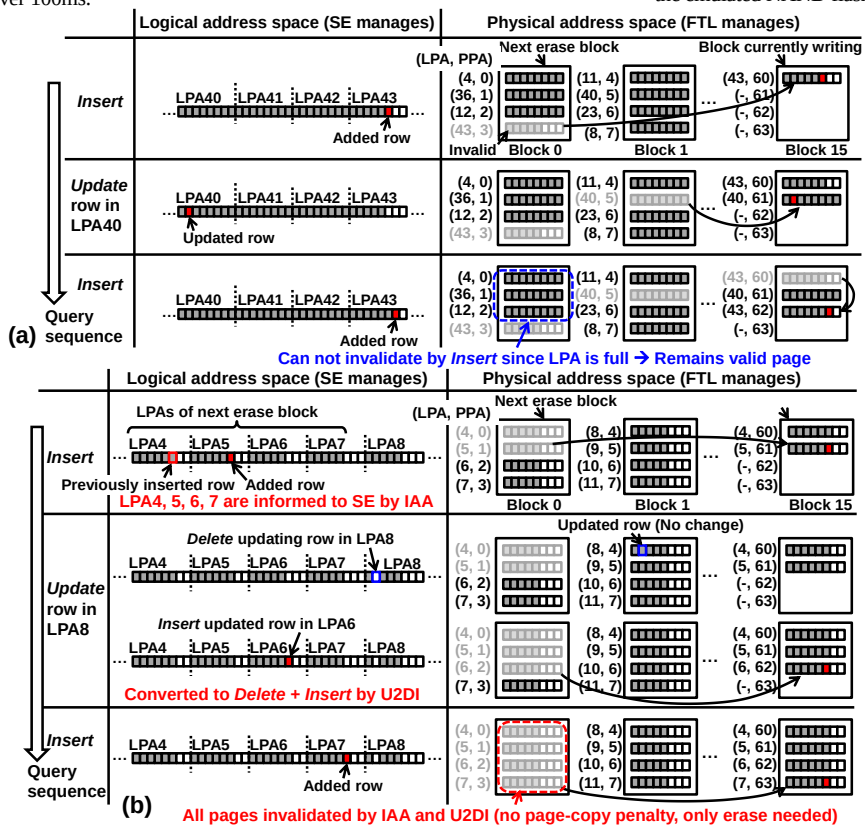

Fig. 10 Example of SE and FTL response in (a) conventional system and (b) proposed system.

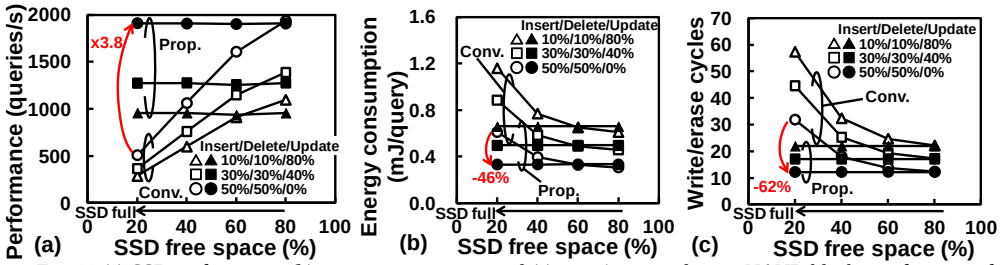

Fig. 11 (a) SSD performance, (b) energy consumption and (c) write/erase cycles per NAND block as a function of SSD free space during random access using *Insert*, *Delete* and *Update* queries. Row data size is 119B.

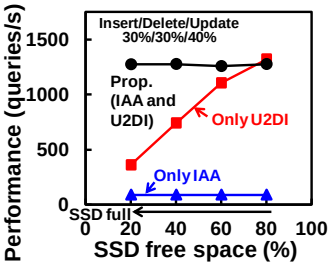

Fig. 12 Effect of IAA and U2DI.