REGULAR PAPER

Co-design of application software and NAND flash memory in solid-state drive for relational database storage system

To cite this article: Kousuke Miyaji *et al* 2014 *Jpn. J. Appl. Phys.* **53** 04EE09

Japanese Journal of Applied Physics **53**, 04EE09 (2014) **REGULAR PAPER**
http://dx.doi.org/10.7567/JJAP.53.04EE09

Co-design of application software and NAND flash memory in solid-state drive for relational database storage system

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

[1]*Department of Electrical, Electronic and Communication Engineering, Faculty of Science and Engineering, Chuo University, Bunkyo, Tokyo 113-8505, Japan*
[2]*Department of Electrical and Electronic Engineering, Faculty of Engineering, Shinshu University, Nagano 380-8553, Japan*
[3]*Department of Electrical Engineering and Information Systems, Graduate School of Engineering, The University of Tokyo, Bunkyo, Tokyo 113-8656, Japan*

Received September 17, 2013; revised January 10, 2014; accepted January 21, 2014; published online March 10, 2014

A relational database management system (RDBMS) is designed based on NAND flash solid-state drive (SSD) for storage. By vertically integrating the storage engine (SE) and the flash translation layer (FTL), system performance is maximized and the internal SSD overhead is minimized. The proposed RDBMS SE utilizes physical information about the NAND flash memory which is supplied from the FTL. The query operation is also optimized for SSD. By these treatments, page-copy-less garbage collection is achieved and data fragmentation in the NAND flash memory is suppressed. As a result, RDBMS performance increases by 3.8 times, power consumption of SSD decreases by 46% and SSD life time is increased by 61%. The effectiveness of the proposed scheme increases with larger erase block sizes, which matches the future scaling trend of three-dimensional (3D-) NAND flash memories. The preferable row data size of the proposed scheme is below 500 byte for 16 kbyte page size.
© 2014 The Japan Society of Applied Physics

1. Introduction

Performance in big data applications, such as enterprise relational database management systems (RDBMSs), which are often implemented by structured query language (SQL) databases,[1,2] are generally limited by the storages' speed and reliability. Although NAND flash memory was originally developed for mobile devices, its ever-decreasing bit cost is driving its expansion to high density solid-state drive (SSD). Replacing hard-disk drive (HDD) with the conventional high-density SSD provides higher performance due to the faster random read speed, and improves energy consumption.[3–16] Figure 1 shows a SSD-based storage system optimization for a high performance RDBMS. A data unit in the SQL database is a row. Row data is stored in the SSD, and managed by storage engine (SE), which responds to queries (command unit of SQL database) from the SQL server. However, because the existing RDBMS SE was developed based on HDD's constraints, modification is needed to realize optimum SSD performance.[17–22] Previous works have focused on leveraging the fast random access speed of SSDs in the top SE layer. New commands have been introduced,[19] the internal parallelism of flash is harnessed for multiple command processing,[20] and various data structures to improve scan and sort times have been proposed.[21] An additional report also proposed using SSD as a cache for HDD.[22] However, these approaches mainly treat the inner workings of the SSD as a black box, and the physical hardware limitations of the NAND flash memory are not considered.

In a NAND flash memory, write unit is a page (cells sharing a same word-line) and erase unit is a block (a group of pages bounded by select gates) as shown in Fig. 2(a), which means that page write and block erase sizes are asymmetric.[4–16] Due to this memory array organization and severe program disturb (false write by the electron injection to the floating gate of unselected cells), data cannot be overwritten in the same physical location.[4–16] In order to overwrite to the same physical page, the whole block must be erased first. Sparseness of data within the block

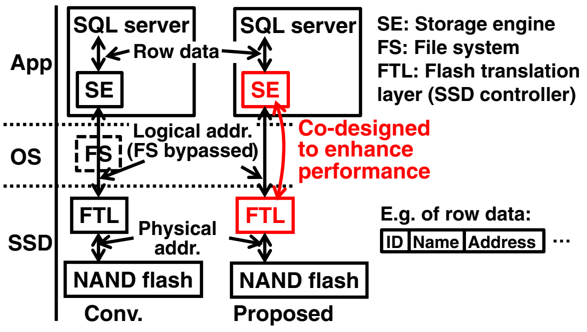

Fig. 1. (Color online) Conventional and proposed storage system configuration of RDBMS using SSD.

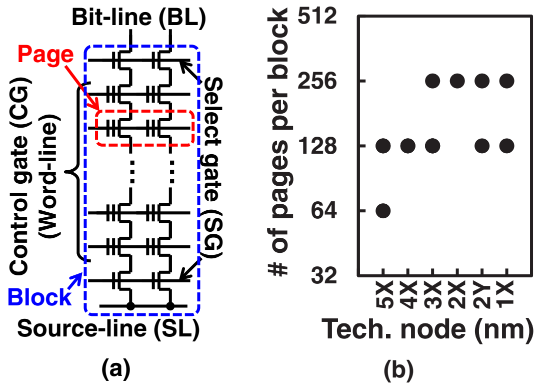

Fig. 2. (Color online) (a) Array structure of NAND flash memory. (b) Number of pages in a NAND flash memory block as a function of technology node.

contributes to performance overhead. The number of pages per block tends to increase as the technology node shrinks [Fig. 2(b)], and thus, SSD needs to handle growing block sizes and write–erase size asymmetry. This trend is even more apparent at three-dimensional (3D-) NAND flash memory architecture,[23–27] whose block size increases by increasing the number of layers. This topic is discussed in

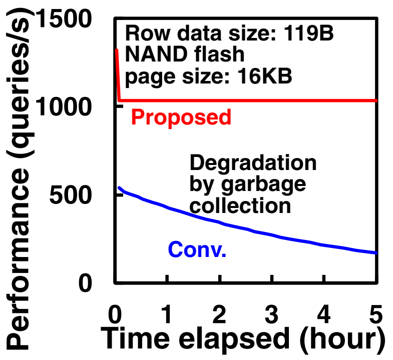

Fig. 3. (Color online) Typical RDBMS performance with SSD as time elapsed. The conventional scheme shows performance degradation due to GC, which is not controlled by SE.

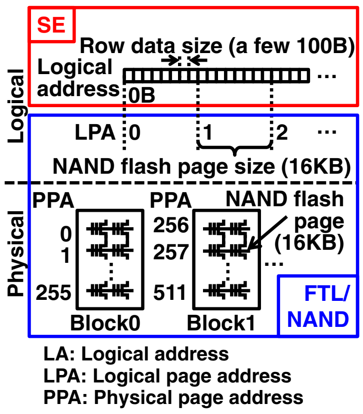

Fig. 4. (Color online) Address hierarchy of storage system in RDBMS. FTL maps logical address and LPA to PPA.

detail at Sect. 4.2. Moreover, the maximum write/erase cycles of a memory cell is limited to 3×10^3 due to the cell reliability degradation by the high operation voltage stress.[13] Therefore, system design is needed to extend the SSD lifetime and reduce the storage exchange cost.

Conventionally, the flash translation layer (FTL) implemented in the SSD controller chip hides these constraints from the SE.[4,13] This abstraction is realized by implementing logical–physical address translation and schemes to track the valid/invalid pages in order to: (i) handle repeated writes to the same logical address and large erase block size, (ii) perform wear-leveling to maximize the NAND flash chip's lifetime, and (iii) execute garbage collection (GC) to reclaim free memory space. The FTL also performs other internal operations like parallel NAND flash memory operation and error-correction code (ECC).[28,29] Normally, this abstraction is very powerful for the file system (FS) because it does not need to handle the cumbersome SSD physical constraints management. However, high performance RDBMS often requires transparency of the storage devices to fully exploit their potential and to guarantee stable performance. Indeed, bypassing FS to remove its performance overhead is often effective (Fig. 1).[1] In such case, the functions of the FTL, particularly the address translation and GC, can cause unpredictable storage performance degradation (details are explained in Sect. 2), which is out of SE's control. Moreover, the RDBMS row's data size of a few 100 B is much smaller than the NAND flash memory page size of 16 KB, which further aggravates data fragmentation and GC (Fig. 3).[4,13]

Unlike previous works in which optimization occurs within the SE layer or in the FTL-NAND flash memory interface, this paper presents co-design of the SE, FTL, and NAND flash memory to minimize the impact of NAND flash memory's internal operations and maximize performance for enterprise RDBMS. This performance improvement can be achieved without a storage class memory that includes alternative memory technologies, such as resistive random access memory (ReRAM), phase change random access memory (PRAM), and ferroelectric random access memory (FRAM).[13–16] The proposed system also accommodates the trend of increasing NAND flash memory block size and it is applicable to 3D-NAND structure.

This paper is organized as follows. The basic operations of SSD are overviewed and the issues of SSD in RDBMS (mainly the data fragmentation and GC) are focused in Sect. 2. In Sect. 3, the proposed RDBMS that SE and FTL cooperates is explained in detail. Two techniques, *Insert* address assist (IAA) and *Update* to *Delete* + *Insert* (U2DI), are proposed to realize the page-copy-less GC operation. The proposed scheme is evaluated through SSD emulator in Sect. 4. The size of SSD free space, NAND flash memory block and row are varied to demonstrate the effectiveness of the proposed scheme. Finally, conclusions are given in Sect. 5.

2. Issues of SSD in RDBMS

As mentioned before, the address translation and GC can cause unpredictable storage performance degradation, especially in RDBMS applications where the data unit size is small. Data fragmentation and frequency of GC are highly dependent on the method of address translation adopted. The addressing hierarchy in the NAND-flash-memory-based RDBMS is shown in Fig. 4, where SE manages the row data logical address. Here, the page-level logical–physical mapping is considered.[5,13] In the FTL, since the minimum write data unit in the NAND flash memory is a page, logical addresses are mapped to the logical page addresses (LPA). LPA is the quotient of the logical address divided by the NAND flash memory page size. The NAND flash memory physical page location is defined as a physical page address (PPA). Due to NAND flash memory's write/erase cycling limitation, LPA and PPA are translated in the FTL using table. Figure 5 shows the physical write operation with NAND flash memory. When SE overwrites a given LPA, the FTL translates the target LPA to PPA and read the old PPA data (step 1). Then, in the SSD controller (FTL), new data from SE is merged with the old PPA data read from NAND flash memory (step 2). Since physical overwrite is not allowed in NAND flash memory, the merged data is written to a new physical page (allocated to a new PPA) in the NAND flash memory (step 3). The old PPA in the NAND flash memory becomes invalid (step 4). After many writes, invalid pages in the SSD accumulate, and GC is triggered to reclaim free space as shown in Fig. 6. In GC, the remaining valid pages in the target erase NAND flash memory block (old block) are read to the SSD controller (seq. 1, 2) and then written to a new NAND flash memory block (seq. 3). Seq. 1 to 3 is called as a page copy. Finally, the old block is erased

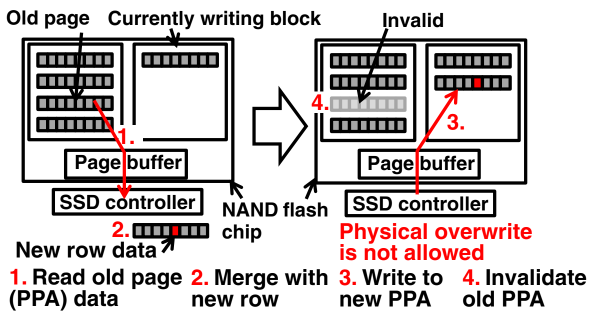

Fig. 5. (Color online) Basic procedure of write operation in SSD.

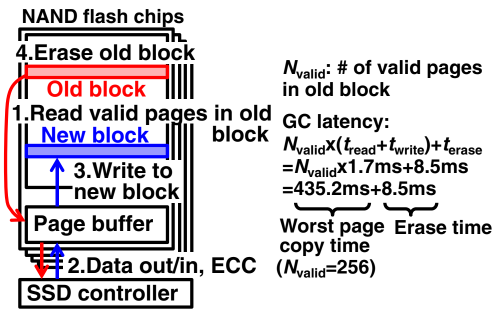

Fig. 6. (Color online) Procedure of GC operation. If N_{valid} is large and page copy occurs many times, GC latency will be huge.

(seq. 4). GC repeats page copy for each of the valid pages (the number of valid pages is defined as N_{valid}) in the block-to-be-freed. GC latency (t_{GC}) is described as

$$t_{\text{GC}} = N_{\text{valid}} \times (t_{\text{read}} + t_{\text{write}}) + t_{\text{erase}},$$

where read latency t_{read}, write latency t_{write} and erase latency t_{erase} of the NAND flash memory are around 100 µs, 1.6 ms, and 8.5 ms, respectively, for example. Since the number of page copy N_{valid} increases when SSD becomes filled with data and can be possibly over 100, t_{GC} can be over 100 ms, which is a severe SSD performance degradation. "Trim" command issued from the system side to notify the invalid region in the SSD is known to increase the SSD performance,[30] but in fact, it also requires page copy to gather invalid regions up to around a NAND block size. Since the row data size is small in RDBMS, the efficiency of the "Trim" command would be typically low and the "Trim" latency will be large when the RDBMS is frequently accessed. Also, the energy consumption of the SSD is essentially not reduced as long as the page copy exists. Therefore, in order to minimize the GC latency and energy consumption, page copy must be avoided.

3. Proposed page-copy-less GC RDBMS

In the proposed scheme, SE, FTL and NAND flash are co-designed to avoid page copy. The concept figure of the conventional and proposed scheme is shown in 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 (details will be explained in Fig. 8). 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

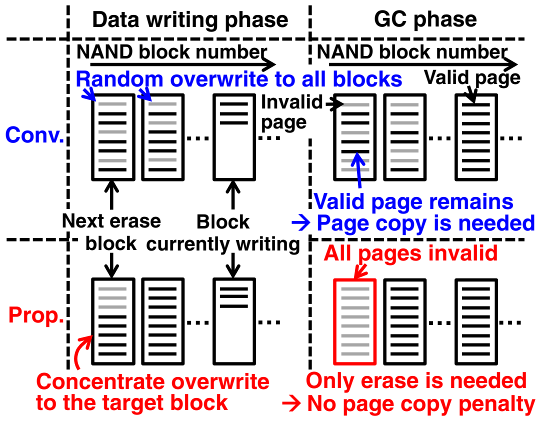

Fig. 7. (Color online) Concept figure of the proposed page-copy-less GC.

next 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. Therefore, erase can be immediately issued in GC. As a result, the SSD performance is enhanced in the proposed scheme.

The detailed SQL RDBMS SE operations for the conventional and proposed scheme are shown in Figs. 8(a) and 9(a), respectively. *Insert*, *Delete*, and *Update* are the query commands that add, eliminate, and change the row data. Conventionally, during *Insert*, a new row is appended to the last row to avoid data fragmentation in logical address space [Fig. 8(a)]. This is because logical and physical addresses are directly mapped assuming a HDD for a storage device, which means that the data fragmentation in the logical address space also results in that in the physical address space. *Delete* disables a row in the logical address space, which has no corresponding operation in the FTL (no write or erase operation). *Update* modifies a row in-place for the same logical address. In the conventional case, SE does not recognize the LPA in which the query operation has taken place. Figure 8(b) shows an example of the conventional SE and FTL response when "*Insert* → *Update* → *Insert* → *Delete* → *Insert*" is issued from the RDBMS. Here, for simplicity, the number of pages in a NAND flash block is 4 and 8 rows can be contained in a page. Suppose Block 0 is the next block to be erased and Block 15 is the block currently writing. PPAs in Block N can be expressed as PPA($4N$) ~ ($4N$+3). As many rows are added or randomly modified beforehand, the order of the LPAs in a physical block would be basically scrambled in the conventional scheme. Here, suppose the last row is in LPA43 and LPA43 is initially allocated to PPA3 in this example. When the first *Insert* is issued, a row is added in LPA43, next to the last row. By this *Insert*, LPA43 is newly allocated and written to the first empty page in Block 15, which is PPA60. Then the previous PPA (PPA3) is invalidated. When the following *Update* is issued to the second row in LPA40, the updated LPA40 is written to the next empty page in Block 15, which is PPA61, and then PPA5 is invalidated. For the next *Insert*, a new row is added in LPA43 next to the previous inserted row. LPA43 is now newly allocated to PPA62 and PPA60 is invalidated. The following *Delete* in LPA12 only changes the data managed in SE and no operation is issued to the SSD. And finally for the next *Insert*, since there is an empty

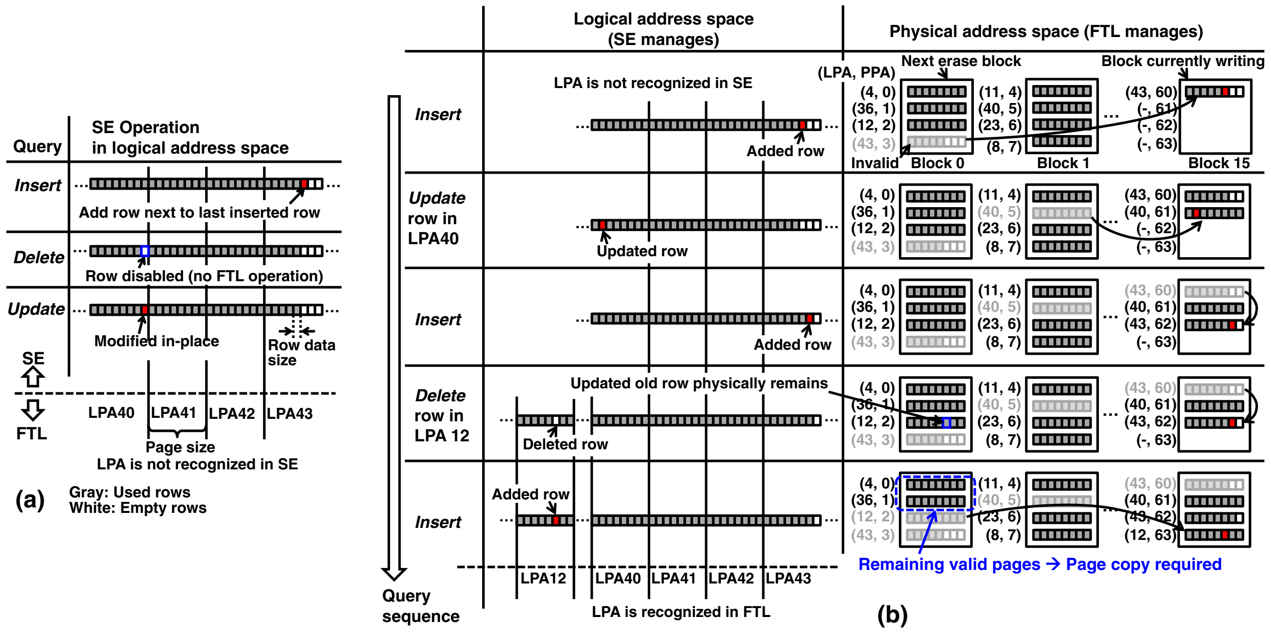

Fig. 8. (Color online) (a) Operations of the conventional SE for *Insert*, *Delete* and *Update* queries. Neither LPA nor PPA is recognized in SE. (b) Example of the SE and FTL operations in the conventional system. Filled up LPAs remain in the next erase block.

row in LPA12 created by the previous *Delete*, the new row is inserted to this empty row in LPA12. Therefore, LPA12 is allocated to PPA63 and previous PPA2 is invalidated. Now, to reclaim free space, GC is needed to be triggered. However, in the conventional case, LPA4/PPA0 and LPA36/PPA1 in Block 0 shown in Fig. 8(b) are still valid and they need page copy before erasing Block 0. This example reveals that when an LPA becomes completely filled with row data, they cannot be invalidated by the subsequent *Insert* since there is no free space in those LPAs. These filled up LPAs (PPAs) remain valid unless the row data are overwritten by *Update* (like the case in LPA40) or an empty row space is created by *Delete* (like the case in LPA12). When SSD stored data is increased, the number of the filled up pages are also increased and they exist everywhere in the NAND flash chip. Hence, $N_{\rm valid}$ can be large during GC, which leads to page copy times of over 100 ms.

In the proposed scheme, two techniques are introduced to avoid generating filled up LPAs that contributes to page copy [Fig. 9(a)]. First, *Insert* address assist (IAA) allows SE to understand LPA-level situations by using following information received from FTL, (i) page size of NAND flash memory to calculate corresponding LPA of the row logical address to be written, (ii) LPAs of the remaining valid pages (PPAs) that are in the target next block to be erased. Whenever *Insert* is issued, SE writes the row data to the free space in the corresponding LPAs of the PPAs in the next erase block received from FTL. Also, for each *Insert*, the target LPA is moved to the next LPA that has been previously written [SE writes to LPA4 → LPA5 → LPA6 → LPA7··· for each *Insert* in Fig. 9(a)], to evenly distribute data over all the available pages, and thus avoid the filled up page situation. By these interactions between the SE, FTL and NAND flash memory, invalid pages can be concentrated in the next block to be erased. 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 using IAA. Otherwise, valid pages remain in the next erase block and invalid pages will be created at the other blocks since IAA cannot be used in *Update*. Although LPA data fragmentation occurs by this scheme, it is much more important to avoid leaving valid pages in the next erase block, which will be shown later in Figs. 12 and 14. Also, the empty rows can be eventually filled by *Insert* as will be explained in Fig. 11. Figure 9(b) shows an example of the proposed SE and FTL response. Suppose LPA4–7 are allocated to PPA0–3 beforehand and Block 0 is the next erase block. In this case, PPA0–3 are needed to be invalidated before the GC. By IAA, FTL informs LPAs that are allocated to PPA0–3. Therefore, LPA4–7 are informed to SE. When the *Insert* is issued, the new row is added to LPA4. LPA4 is newly written to PPA60 and PPA0 is invalidated. Similarly for the next *Insert*, the new row is added to LPA5 and it is written to PPA61 followed by invalidating PPA1. For the next *Update* to LPA8, the updating row in LPA8 is deleted and newly added to LPA6 by U2DI. Therefore, PPA2 can be invalidated and the LPA6 is now allocated to PPA62. In these manners, 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 of *Insert*, *Delete*, and *Update*. In the proposed scheme, the SSD write data unit is not limited to row size, as shown in these examples. The proposed scheme is also effective with buffer write scheme as long as the write unit size can achieve stable IAA operation (see Sect. 4.3 for details).

Figure 10 shows the procedure to hand the LPA requests from FTL to SE by IAA. In the FTL, write/erase count of each NAND flash memory block is stored in the wear-leveling table. FTL refers to the wear-leveling table and selects the next erase block according to the wear-leveling policy. Using the physical address of the next erase block,

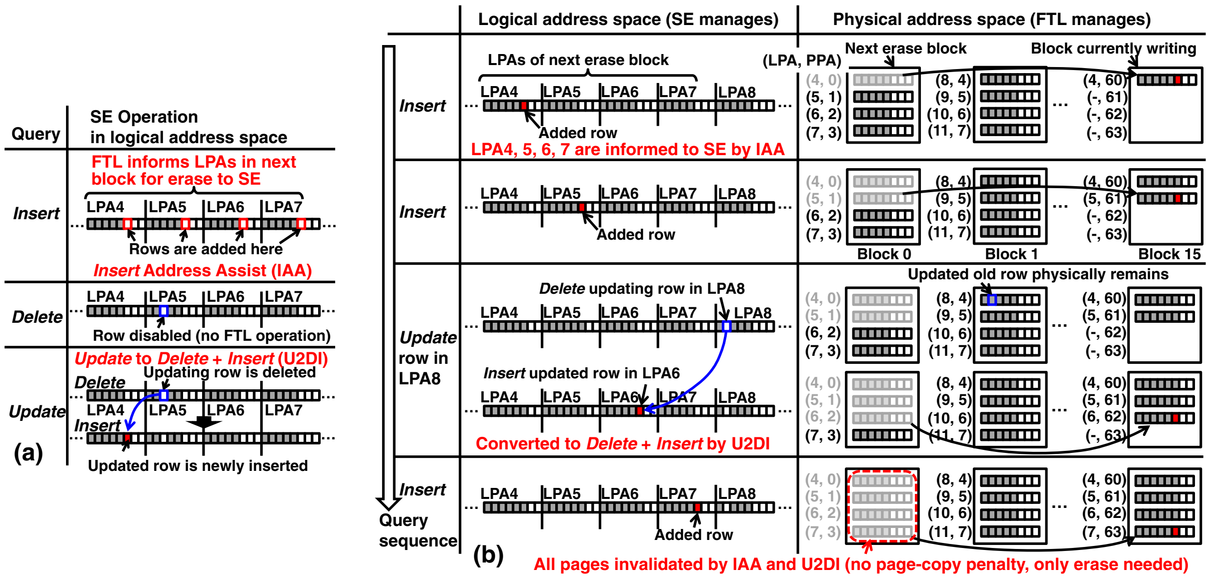
Query
SE Operation in logical address space
FTL informs LPAs in next block for erase to SE
Insert
Rows are added here
Insert Address Assist (IAA)
Delete
Row disabled (no FTL operation)
Update to Delete + Insert (U2DI)
Updating row is deleted
Update
Updated row is newly inserted
(a)
Logical address space (SE manages)
Physical address space (FTL manages)
LPAs of next erase block
(LPA, PPA)
Next erase block
Block currently writing
Added row
LPA4, 5, 6, 7 are informed to SE by IAA
Block 0
Block 1
Block 15
Updated old row physically remains
Delete updating row in LPA8
Update row in LPA8
Insert updated row in LPA6
Converted to Delete + Insert by U2DI
Query sequence
(b)
All pages invalidated by IAA and U2DI (no page-copy penalty, only erase needed)

Fig. 9. (Color online) (a) Operations of the proposed SE for *Insert*, *Delete*, and *Update* queries. SE recognizes LPA segmentation and LPAs in the next erase block provided from the FTL. IAA and U2DI are implemented in the SE. (b) Example of the SE and FTL operations in the proposed system. Invalid pages always concentrate on the next erase block thanks to IAA and U2DI.

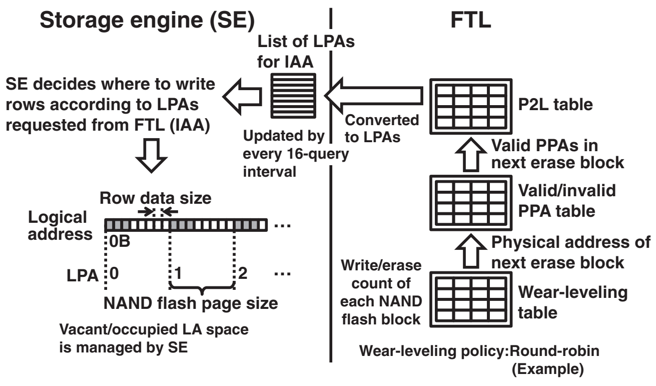
Storage engine (SE)
FTL
List of LPAs for IAA
SE decides where to write rows according to LPAs requested from FTL (IAA)
Updated by every 16-query interval
Converted to LPAs
P2L table
Valid PPAs in next erase block
Valid/invalid PPA table
Physical address of next erase block
Write/erase count of each NAND flash block
Wear-leveling table
Wear-leveling policy:Round-robin (Example)
Row data size
Logical address
LPA
NAND flash page size
Vacant/occupied LA space is managed by SE

Fig. 10. Procedure of IAA to obtain a list of LPAs to write from FTL.

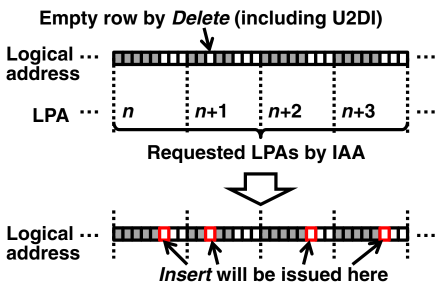
Empty row by Delete (including U2DI)
Logical address
LPA
n
n+1
n+2
n+3
Requested LPAs by IAA
Insert will be issued here

Fig. 11. (Color online) IAA *Insert* policy when an empty row is created by *Delete*.

valid PPAs in this block is obtained from the valid/invalid PPA table. Then, the valid PPAs in the next erase block are converted to the corresponding LPAs by the physical–logical address table (P2L table) and passed to SE. In this work, the list of the LPAs provided to SE is updated by every 16 queries. The important aspect of IAA is that the abstraction of the SSD is still realized since the SE can leave the management of the whole wear-leveling or GC function to the FTL. Therefore, IAA can be generally used regardless of the SSD or FTL type. Figure 11 shows the IAA policy when an empty row is created by *Delete* (including that induced by U2DI). IAA gives priority to that empty row for the next target row to issue *Insert*.

4. Evaluations and results

4.1 SSD free space dependence

A DB system is developed by implementing the proposed SE in MySQL,[1] and transaction-level-modeling-based SSD emulator for the FTL and NAND flash memory.[13] In this section, the row data size is fixed to 119 B. For simplicity, a 1 Gbyte SSD is emulated. The SSD configuration is 16 KB page size, 256 pages/block, 32 blocks/plane, 2 planes/chip, and 4 chips. I/O and SSD latency parameters can be found in the previous work.[13] In this work, round-robin and page-level mapping are used for wear-leveling and logical–physical mapping, respectively, for example. Figure 12 shows RDBMS performance, SSD energy consumption and write/erase cycles as a function of SSD free space. When SSD free space is 20%, 80% of the pages in NAND flash memories are valid and completely filled up in the conventional scheme while in the proposed scheme, all the pages are valid and filled with 80% of the page size. A data pattern of 1.6×10^6 random queries of *Insert*, *Delete*, and *Update*, with varying probabilities are issued after filling the SSD up to the target SSD free space by initial *Insert*, considering that the random access pattern is the most severe condition for the RDBMS. Probabilities of *Insert* and *Delete* are made the same so that the macroscopic SSD free space does not effectively change in this evaluation. The results in Fig. 12 are obtained during the period of random query phase. In the conventional scheme, as the SSD fills up and free space decreases, performance, energy consumption and write/erase cycles degrade, because page copy increases as the number of valid pages filled up with row data is increased. The extra

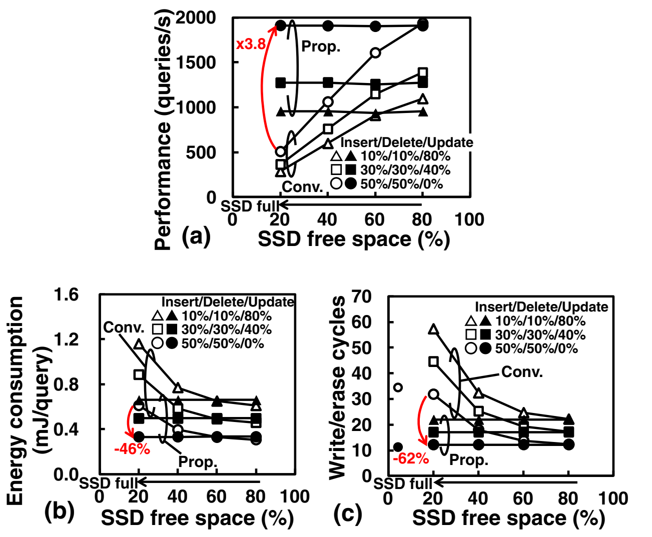

Fig. 12. (Color online) (a) SSD performance, (b) energy consumption, and (c) write/erase cycles of a NAND flash memory block as a function of SSD free space. 1.6×10^6 queries are issued randomly with various *Insert*, *Delete*, and *Update* probabilities.

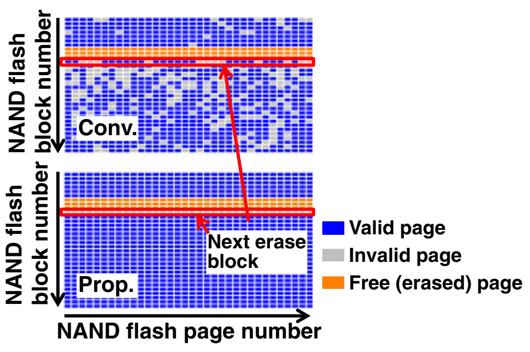

Fig. 13. (Color online) Valid and invalid page locations in the conventional and proposed SSD obtained from the simulations in Fig. 12. The concept of page-copy-less GC in Fig. 7 is achieved in the proposed scheme.

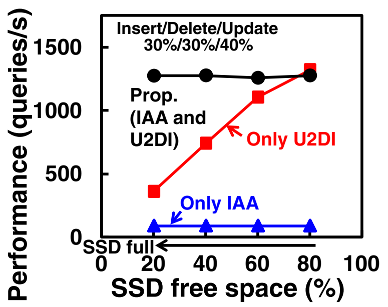

Fig. 14. (Color online) Effect of IAA and U2DI. Both IAA and U2DI must be used at the same time.

Table I. Average RDBMS performance during SSD free space gradually decreasing from 80 to 20% by applying 2.4×10^7 random queries. Row size is 119 byte and *Insert*/*Delete*/*Update* probability is 40%/20%/40%.

	Performance (query/s)	Energy consumption (mJ/query)	Write/erase cycles
Conventional	429	0.625	412
Proposed	558 (+30%)	0.540 (−14%)	293 (−29%)

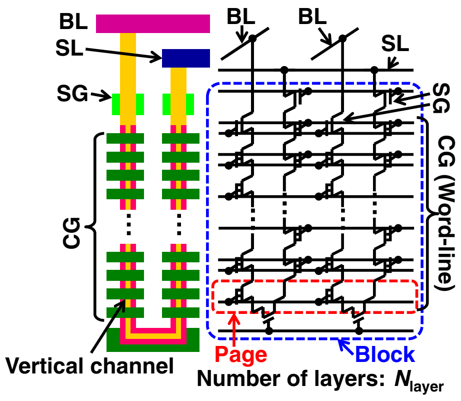

Fig. 15. (Color online) Structure and equivalent circuit of PBiCS 3D-NAND flash memory.[24,25]

writes due to the page copy increase energy consumption and write/erase cycles. In contrast, no degradation is observed in the proposed scheme, thanks to the page-copy-less GC operation. Figure 13 shows the valid page locations in the NAND flash for the conventional and proposed scheme to sustain the results. In the proposed scheme, invalid pages are concentrated in one block, whereas they are randomly located in the conventional scheme. Therefore, it is clear that the proposed scheme can eliminate the page copy but the conventional scheme cannot. The proposed scheme has 3.8 times higher performance, 46% less energy consumption and 62% less write/erase cycles, which corresponds to a 61% SSD lifetime increase, when the SSD free space is 20%. Figure 14 shows the individual and combined effects of the proposed IAA and U2DI, indicating that the two proposals must be implemented together to gain performance enhancement for any amount of SSD free space.

Table I shows the average RDBMS performance when the SSD free space gradually decreases from 80 to 20% while 2.4×10^7 random queries are issued. *Insert* probability is 40% while *Delete* probability is 20% to decrease SSD free space gradually. Since the conventional scheme is also effective when the SSD free space is small, the average performance enhancement is around 30%. The energy consumption and write/erase cycles are reduced by 14 and 29% respectively.

4.2 Compatibility with 3D-NAND structure: Block size dependence

In this section, RDBMS performance is evaluated in terms of the block size of the NAND flash memory considering the trend of the future NAND flash memory scaling by a 3D-NAND structure.[23–27] As illustrated in Fig. 15, memory cells are stacked vertically upwards in N_{layer} layers in the 3D-NAND flash memory to increase the memory density. As the technology node of 3D-NAND proceeds, N_{layer} is also increased which results in further increase of the erase block size and write–erase size asymmetry as shown in Fig. 16. Figure 17(a) shows RDBMS performance and SSD energy consumption and Fig. 17(b) shows write/erase cycles as a function of the NAND flash block size. Corresponding N_{layer} in PBiCS 3D-NAND flash memory is also shown.[24,25] The

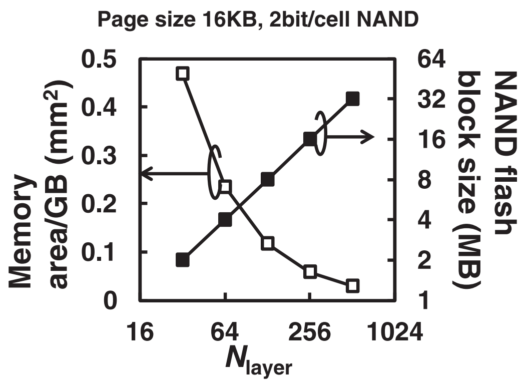

Fig. 16. Memory area per GB storage and block size as a function of N_{layer}. Here, 2 bit/cell architecture and 16 kbyte page size are assumed.

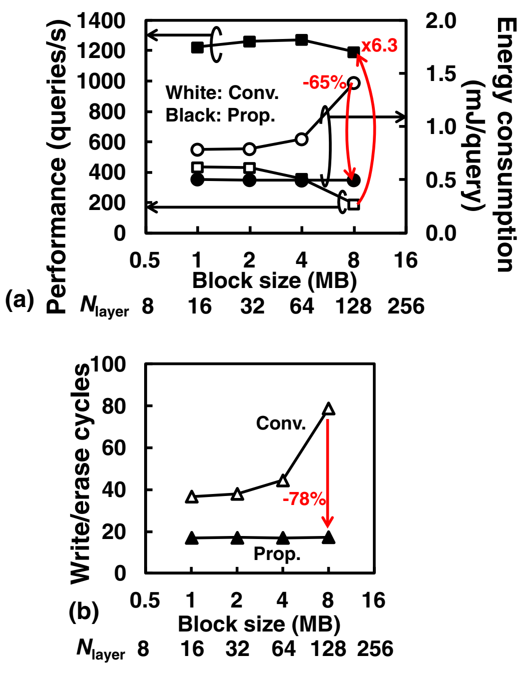

Fig. 17. (Color online) (a) SSD performance, energy consumption and (b) write/erase cycles of a NAND flash memory block as a function of block size. SSD free space is 20%. *Insert*/*Delete*/*Update* probability is 30%/30%/40%.

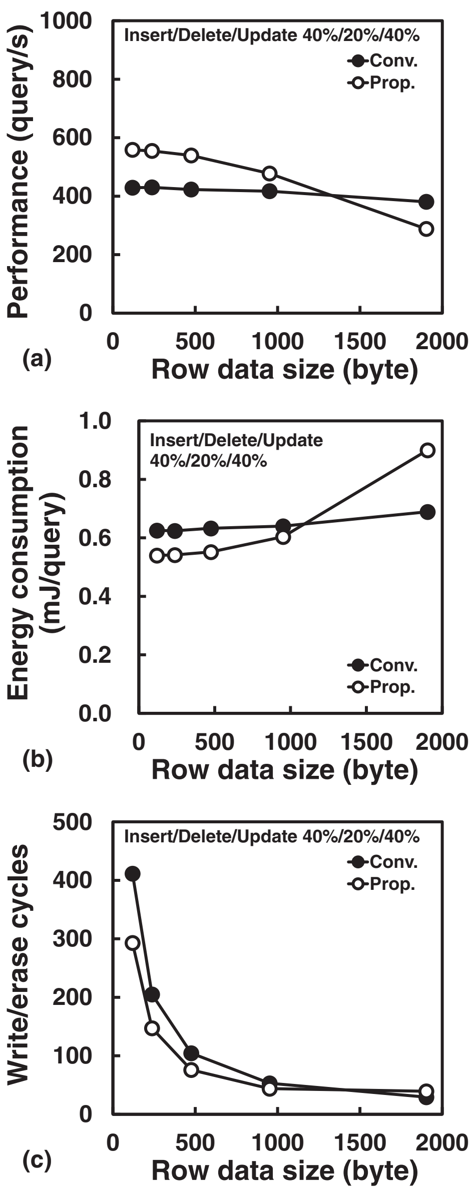

Fig. 18. Average (a) SSD performance, (b) energy consumption, and (c) write/erase cycles of a NAND flash memory block as a function of row data size. SSD free space is gradually changed from 80 to 20% by random queries.

proposed scheme is well-suited for increasing N_{layer} in 3D-NAND flash memory. When a block size increases to 8 Mbyte (128 layers), the gains in the proposed scheme's performance, energy reduction, and write/erase cycle reduction are enhanced to 6.3 times, 65%, and 78%, respectively. These results indicate that the proposed scheme is even compatible with the future NAND flash memory chip architecture.

4.3 Row size dependence

The row size (or write unit size) dependence on the RDBMS performance is also investigated in this paper. The data pattern is the same as Table I (*Insert*/*Delete*/*Update* probability is 40%/20%/40%). Figures 18(a)–18(c) are the average RDBMS performance, energy consumption, and write/erase cycles of the SSD, respectively. The performance of the proposed scheme degrades from around 500 B row size, where the conventional scheme shows less row size dependence. The reason for the performance decrease in the proposed scheme is illustrated in Fig. 19. Two cases whose row size is small and large are assumed. Suppose the total SSD free space is 37.5% of the SSD capacity in this example. However, since queries are randomly issued, the free space of the LPA is 37.5% in average but varies LPA to LPA in local. For example, some LPAs may be only subjected to *Insert*, as shown in LPA2 in Fig. 19. Others may be mostly subjected to *Delete* and *Update*, thus those LPAs have larger free space like LPA6. If the row size is large, some LPAs are filled up due to excess *Insert* accesses. In such LPAs, IAA fails and page copy occurs during GC resulting in performance degradation. However, if the row size is small enough, the possibility of the specific LPA becoming filled up is greatly reduced since the number of rows that can be inserted in the free space is large compared with the large row size case.

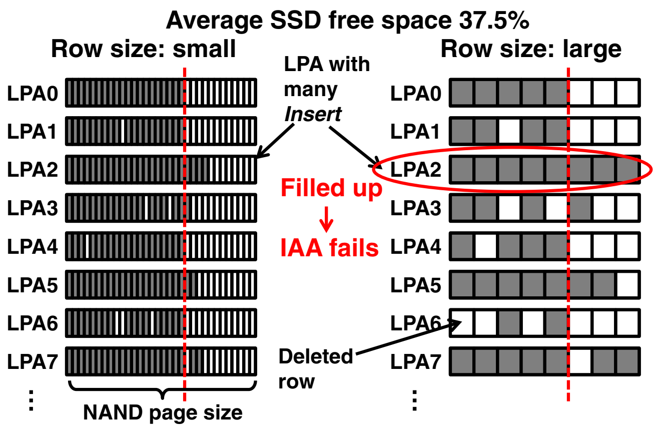

Fig. 19. (Color online) Impact of row data size to the variations of the free space in each LPA.

Basically, the proposed scheme is aimed to avoid the data fragmentation for small row size system, which is often the case in RDBMS. The proposed scheme is effective at the SSD free space below 60% and row size below 500 byte. It should be noted that the small row size does not have an impact on the FTL logical–physical mapping table size, since the table size is determined by the logical–physical mapping policy. If page-level mapping is used for example, the table size is proportional to the total number of pages in the SSD. The drawbacks for the small row size can be the increased data management cost in SE layer when the row data number is large. However, this is considered to be a general problem and not specific to the proposed scheme.

5. Conclusions

SE in SQL RDBMS, SSD FTL, and NAND flash memory are co-designed for vertically optimization. The proposed IAA and U2DI schemes achieve page-copy-less GC. When SSD free space is 20%, SSD performance increases by 3.8 times, power consumption decreases by 46% and SSD lifetime increases by 61% using the proposed scheme. The proposed scheme is even effective at the large block sizes, which matches the scaling trend of the 3D-NAND flash technology. The row size should be less than 500 byte to keep IAA effective at 16 kbyte page size.

Acknowledgment

This work is partly supported by NEDO.

1) Web [http://dev.mysql.com/doc/refman/5.0].
2) Web [http://www.postgresql.org/docs/manuals/].
3) F. Masuoka, K. Momodomi, Y. Iwata, and R. Shirota, IEDM Tech. Dig., 1987, p. 552.
4) K. Takeuchi, IEEE J. Solid-State Circuits **44**, 1227 (2009).
5) J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, IEEE Trans. Consum. Electron. **48**, 366 (2002).
6) S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, ACM Trans. Embedded Comput. Syst. **6**, 18.1 (2007).
7) S. J. Kwon and T.-S. Chung, IEEE Trans. Consum. Electron. **54**, 631 (2008).
8) M.-L. Chiao and D.-W. Chang, IEEE Trans. Comput. **60**, 753 (2011).
9) E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, IEEE Trans. Comput. **60**, 653 (2011).
10) C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, NVSMW Tech. Dig., 2006, p. 17.
11) S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung, IEEE Trans. Consum. Electron. **55**, 1392 (2009).
12) Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J. Y. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min, IEEE Trans. Comput. **59**, 905 (2010).
13) H. Fujii, K. Miyaji, K. Johguchi, K. Higuchi, C. Sun, and K. Takeuchi, Symp. VLSI Tech. Dig., 2012, p. 134.
14) G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li, HPCA Tech. Dig., 2010, p. 1.
15) J. H. Yoon, E. H. Nam, Y. J. Seong, H. Kim, B. S. Kim, S. L. Min, and Y. Cho, IEEE Comput. Archit. Lett. **7**, 17 (2008).
16) H. G. Lee, IEEE Trans. Consum. Electron. **56**, 112 (2010).
17) P. Bonnet, L. Bouganim, I. Koltsidas, and S. D. Viglas, VLDB’11, 2011, p. 1504.
18) K. Kim, S. W. Lee, B. Moon, C. Park, and J. Y. Hwang, VLDB’11, 2011, p. 1363.
19) D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, SIGMOD’09, 2009, p. 59.
20) H. Roh, S. Park, S. Kim, M. Shin, and S. W. Lee, VLDB’12, 2012, p. 286.
21) M. A. Bender, M. F. Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, HotStorage’11, 2011.
22) W. H. Kang, S. W. Lee, and B. Moon, VLDB’12, 2012, p. 1615.
23) H. Tanaka, M. Kido, K. Yahashi, M. Oomura, R. Katsumata, M. Kito, Y. Fukuzumi, M. Sato, Y. Nagata, Y. Matsuoka, Y. Iwata, H. Aochi, and A. Nitayama, Symp. VLSI Tech. Dig., 2007, p. 14.
24) R. Katsumata, M. Kito, Y. Fukuzumi, M. Kido, H. Tanaka, Y. Komori, M. Ishiduki, J. Matsunami, T. Fujiwara, Y. Nagata, L. Zhang, Y. Iwata, R. Kirisawa, H. Aochi, and A. Nitayama, Symp. VLSI Tech. Dig., 2009, p. 136.
25) T. Maeda, K. Itagaki, T. Hishida, R. Katsumata, M. Kito, Y. Fukuzumi, M. Kido, H. Tanaka, Y. Komori, M. Ishiduki, J. Matsunami, T. Fujiwara, H. Aochi, Y. Iwata, and Y. Watanabe, Symp. VLSI Circ. Dig., 2009, p. 22.
26) J. Jang, H. S. Kim, W. Cho, H. Cho, J. Kim, S. I. Shim, Y. Jang, J. H. Jeong, B. K. Son, D. W. Kim, K. Kim, J. J. Shim, J. S. Lim, K. H. Kim, S. Y. Yi, J. Y. Lim, D. Chung, H. C. Moon, S. Hwang, J. W. Lee, Y. H. Son, U. I. Chung, and W. S. Lee, Symp. VLSI Tech. Dig., 2009, p. 192.
27) S. J. Whang, K. H. Lee, D. G. Shin, B. Y. Kim, M. S. Kim, J. H. Bin, J. H. Han, S. J. Kim, B. M. Lee, Y. K. Jung, S. Y. Cho, C. H. Shin, H. S. Yoo, S. M. Choi, K. Hong, S. Aritome, S. K. Park, and S. J. Hong, IEDM Tech. Dig., 2010, p. 668.
28) S. Tanakamaru, Y. Yanagihara, and K. Takeuchi, ISSCC Tech. Dig., 2012, p. 424.
29) H. Choi, W. Liu, and W. Sung, IEEE Trans. VLSI **18**, 843 (2010).
30) T. Frankie, G. Hughes, and K. Kreutz-Delgado, ACM-SE’12, 2012, p. 59.