Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD

Soojun Im and Dongkun Shin, *Member*, *IEEE*

Abstract—Solid-state disks (SSDs), which are composed of multiple NAND flash chips, are replacing hard disk drives (HDDs) in the mass storage market. The performances of SSDs are increasing due to the exploitation of parallel I/O architectures. However, reliability remains as a critical issue when designing a large-scale flash storage. For both high performance and reliability, Redundant Arrays of Inexpensive Disks (RAID) storage architecture is essential to flash memory SSD. However, the parity handling overhead for reliable storage is significant. We propose a novel RAID technique for flash memory SSD for reducing the parity updating cost. To reduce the number of write operations for the parity updates, the proposed scheme delays the parity update which must accompany each data write in the original RAID technique. In addition, by exploiting the characteristics of flash memory, the proposed scheme uses the partial parity technique to reduce the number of read operations required to calculate a parity. We evaluated the performance improvements using a RAID-5 SSD simulator. The proposed techniques improved the performance of the RAID-5 SSD by 47 percent and 38 percent on average in comparison to the original RAID-5 technique and the previous delayed parity updating technique, respectively.

Index Terms—Redundant arrays of inexpensive disks (RAID), flash memory, solid-state disk (SSD), reliability, dependability.

1 Introduction

During the last decade, there have been dramatic changes in data storage systems. The evolution of NAND flash memory has enabled several portable devices, such as MP3 players, mobile phones, and digital cameras, to be accommodated with a large amount of data storage. Flash memory has the features of low-power consumption, nonvolatility, high random access performance, and high mobility, and hence, it is well-suited for portable consumer devices. Recently, due to the dramatic price reduction, flash memory has been extending its domain to mass storage systems for desktop PCs or enterprise servers. As a result, the flash memory solid-state disks (SSDs), which are composed of multiple flash chips, are replacing hard disk drives (HDDs) in the mass storage market [1], [2]. SSDs are appreciated especially for energy efficiency over HDDs due to the absence of mechanical moving parts, and thus, they are attractive to the power-hungry data center.

However, the cost per bit of NAND flash memory is still high. In recent years, multilevel cell (MLC) flash memories have been developed as effective solutions to increase storage density and reduce the cost of flash devices. However, MLC flash has slower performance and less reliability than those of single-level cell (SLC) flash for the sake of its low cost. Therefore, the performance and reliability problems should be solved for dependable and high-performance flash memory SSD. To enhance the performance of SSDs, we can use parallel I/O architectures, such as multichannel and interleaving [3], which increase the I/O bandwidth by allowing concurrent I/O operations over multiple flash chips. However, the reliability problem is still a critical issue when designing large-scale flash storage systems.

Current NAND flash products ensure reliability by employing error-correcting codes (ECCs). Traditionally, SLC flash memory uses single-bit ECC, such as Hamming codes. These ECCs are stored in the spare area of flash blocks, the extra space per page for metadata. When a page is read from the flash device, the flash memory controller calculates a new ECC with the page and compares it with the ECC that is stored in the spare area in order to detect and correct bit errors before the page data are forwarded to the host.

However, MLC flash memory shows a much higher bit-error rate (BER) that can be managed with single-bit error-correcting codes. Multiple bits are stored in each memory cell of the MLC flash memory by programming each cell with multiple threshold levels. Therefore, the reduced operational margin significantly degrades the reliability of flash memories. In addition, as silicon technology evolves, cell-to-cell interference is increasing. As a result, codes with strong error-correction capabilities, like BCH or Reed-Solomon (RS) codes, are used. However, these ECCs require a high hardware complexity and increase the read and write latencies.

Another approach for reliability is to adopt redundancy in storage level. The Redundant Arrays of Inexpensive Disks (RAID) [4] technique uses an array of small disks in order to increase both performance and reliability. Current SSD products employ RAID level 0 (RAID-0) striping architecture, which spreads data over multiple disks to improve performance. Concurrent accesses to multiple flash chips are allowed so as to improve sequential access. However, RAID-0 does not improve the reliability, since it does not use redundant data. RAID-4 and RAID-5 architectures are

• *The authors are with the School of Information and Communication Engineering, Sungkyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon, Gyeonggi-do 440-746, Korea.*
E-mail: {lang33, dongkun}@skku.edu.

Manuscript received 24 Jan. 2010; revised 05 June 2010; accepted 18 Aug. 2010; published online 28 Sept. 2010.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2010-01-0044.
Digital Object Identifier no. 10.1109/TC.2010.198.

0018-9340/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

widely used to provide redundancy. In such architectures, one disk is reserved to store the parity, and thus, the multibit burst error in a page, block, or device can be easily corrected. Parity requires negligible calculation time compared to error correction codes. There are also file-system-level parity schemes [5], where one flash block of a segment is dedicated to the parity of other blocks in the segment and one flash page of a block is also dedicated to the parity of other pages in the block. Such file-system-level schemes can be used independently or in combination with RAID storage architectures.

In order to implement RAID-4 or RAID-5 technology in flash storage, we should consider the characteristics of flash memory. To manage parity data, frequent write requests are necessary, which significantly deteriorate the performance due to the slow write performance of NAND flash memory. Whenever a page is updated, the other pages need to be read to calculate a new parity, and the new parity should be written into the flash memory. Therefore, we need a flash-aware RAID technique to implement reliable and high performance flash memory SSDs.

In this paper, we propose a novel RAID-5 architecture for flash memory SSD in order to reduce the parity updating cost. It is a delayed parity update scheme with a partial parity caching technique. To reduce the number of write operations for these parity updates, the proposed scheme delays the parity update which must accompany each data write in the original RAID-5 technique. The delayed parities are maintained in the parity cache until they are written to the flash memory. In addition, the partial parity caching technique reduces the number of read operations required to calculate a new parity. By exploiting the characteristics of flash memory, the partial parity caching technique can recover the failed data without full parities. We provide performance evaluation results based on a RAID-5 SSD simulator.

The rest of the article is organized as follows: Section 2 introduces backgrounds on flash memory, SSD, and RAID techniques. In Section 3, related works on flash memory SSD and RAID-based SDD are introduced. Section 4 describes the proposed delayed parity update scheme. The experimental results are presented in Section 5, and Section 6 concludes with a summary and descriptions of future works.

2 Background

2.1 Flash Memory

Flash memory has several special features, unlike the traditional magnetic hard disk. The first one is its "erase-before-write" architecture. To write data into a block, the block should first be cleaned by the erase command. The erase operation in flash memory changes all of the bits in the block to the logical value 1. The write operation changes some bits to the logical value 0 but cannot restore the logical value 0 to the logical value 1. The second feature is that the unit sizes of the erase and write operations are asymmetric. While the write operation is performed by the unit of a page, the flash memory is erased by the unit of a block, a bundle of several sequential pages. For example, in a large block-based MLC NAND flash memory [6], one block is composed of

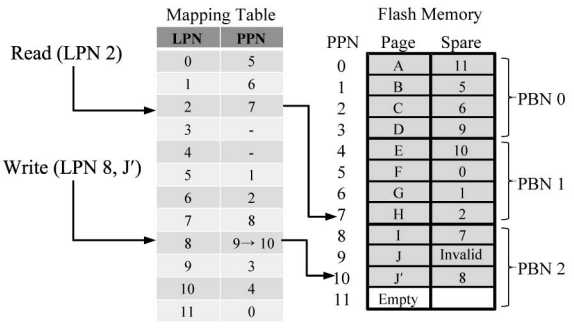

Fig. 1. Page-level address mapping in flash memory.

128 pages, and the size of a page is 4 KB. Due to these two features, special software called the flash translation layer (FTL) is required, which maps the logical page address from the host system to the physical page address in flash memory devices. Flash memory SSDs also require an embedded FTL, which performs on the SSD controller.

The address mapping schemes of the FTL can be divided into three classes depending on the mapping granularity: page-level mapping, block-level mapping, and hybrid mapping. Fig. 1 shows an example of page-level mapping, where a logical page can be mapped to any physical page in flash memory. When a host sends the read request with a logical page number (LPN), FTL finds the physical page number (PPN) from the mapping table. Since the mapping table is generally maintained in SRAM, each physical page also has its LPN in the spare field against sudden power failures. If an update request is sent for the data that have already been written to flash memory, page-level mapping technique writes the new data to an empty page, invalidates the old data, and changes the mapping information for the logical page number since the flash memory page cannot be overwritten. The invalidation of old data is marked at its spare field. The drawback of the page-level mapping technique is that its mapping table size is inevitably large, and thus, it requires a large SRAM for mapping table.

In block-level mapping, only the mapping information between the logical block number (LBN) and the physical block number (PBN) is maintained. Therefore, a page should be in the same page offset within both the logical block and the physical block. Block-level mapping requires a small-sized mapping table. However, when a logical page is updated, the page should be written to a new clean flash block, and all of the nonupdated pages of the old block should be copied into the new flash block. Block-level mapping, therefore, invokes large page migration costs.

Hybrid mapping [7], [8], [9], [10], [11] is a compromise between page-level mapping and block-level mapping. In this scheme, a small portion of physical blocks is reserved as a log buffer. While the log blocks in the log buffer use the page-level mapping scheme, the normal data blocks are handled by the block-level mapping. When a write request is sent to the FTL, the data are written to a log block, and the corresponding old data in the data block are invalidated. When there is no empty space in the log buffer, one of the log blocks is selected as a victim and all of the valid pages in the log block are moved into the data blocks to make space

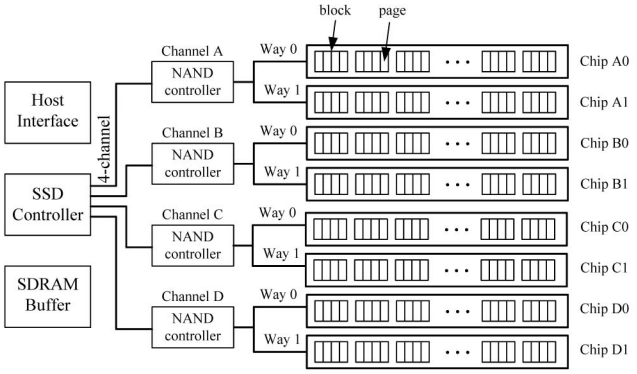

Fig. 2. Multichannel and multiway (four-channel and two-way) SSD architecture.

for on-going write requests. This process is referred to as log block merge [7].

Hybrid mapping requires a small-sized mapping table since only the log blocks are handled by the page-level mapping. In addition, unlike block-level mapping, hybrid mapping does not invoke a large page migration cost for every write request. As a result, most SSDs are employing the hybrid mapping or page-level mapping techniques.

2.2 SSD Architecture

To enhance the bandwidth of the flash memory SSD, interleaving techniques are used. Fig. 2 shows an example of the multichannel and multiway SSD architecture [3], [12]. The four channels can be operated simultaneously. Two flash chips using different channels can be operated independently, and, therefore, the page program times for the different chips can overlap. In addition, one NAND controller can access two flash chips in an interleaved manner, and, therefore, we can write to two interleaved chips simultaneously. However, since two flash chips sharing one bus cannot occupy the data bus simultaneously, the data transfer times cannot overlap.

Fig. 3 shows the parallel operations in 4-channel and 2-way SSD architecture. If the bus speed is 40 MB/s ($100\ \mu s/4$ KB) and the program time for a 4 KB page in MLC is $800\ \mu s$, the total time to program eight pages is 1 ms. We can program eight pages in parallel with 4-channel and 2-way SSD architecture. To utilize such parallel architectures, sequential data are distributed across multiple flash chips. Therefore, the parallel architecture can provide a high bandwidth for sequential requests. However, random I/O performances are poor compared to those of sequential I/O.

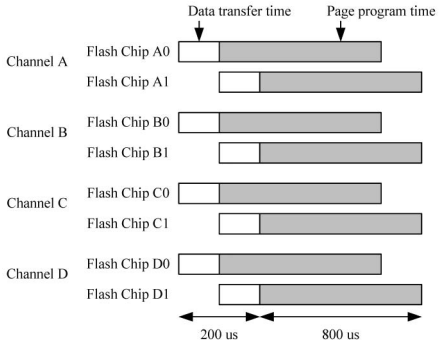

Fig. 3. Parallel operation at four-channel and two-way architecture.

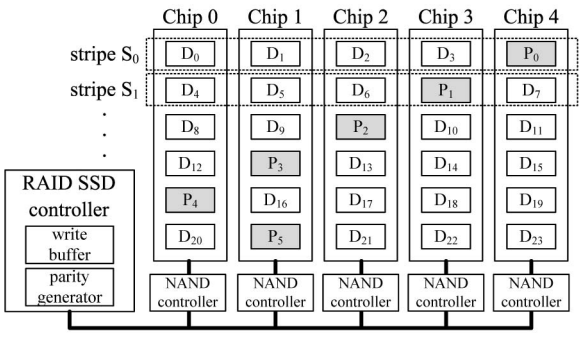

Fig. 4. $4+1$ RAID-5 SSD Architecture.

2.3 RAID Technologies

RAID enhances the reliability of storage systems by using redundant data and improves the performance by interleaving data across multiple disks. There are several levels of RAIDs; RAID-0 uses only the interleaving technique without redundant data, therefore, it does not improve the reliability. Current SSD products use the RAID-0 technique internally while utilizing the multichannel and multiway architecture in order to improve the I/O bandwidth.

RAID-4 and RAID-5 use an extra disk to hold redundant information that is necessary to recover user data when a disk fails. RAID-4 and RAID-5 stripe data across several drives, with the parity stored on one of the multiple drives. In particular, in RAID-5, each stripe stores the parity on a different drive to prevent one of the disks from being bottleneck. Therefore, RAID-5 provides high read and write performances due to parallel access to multiple disks.

The RAID-5 SSD can be implemented using several flash chips or flash drives, as shown in Fig. 4. Depending on the striping granularity, each stripe is composed of N+1 logically sequential pages or blocks, where N represents the number of disks for user data. The RAID controller has a write buffer that stores data temporarily until it is written to the flash chips. It also generates a parity of data to be written and distributes the user data and the parity across multiple flash chips. The NAND controller writes the data or parity to the flash chips. The parities are written at different chips for different stripes. When D_1 and D_6 are updated in Fig. 4, flash chips 1, 2, 3, and 4 may become busy in parallel or in an overlapped fashion, and thus, no single drive remains in a performance bottleneck.

In RAID-5 SSD, we should distinguish the logical and physical addresses. The RAID controller determines the stripe based on the logical address. Therefore, all pages of a stripe have the same logical offsets within their flash chips, but the physical offsets can be different since the logical address is translated into a physical address by the FTL.

The stripe index j of data D_i can be defined as $j = \lfloor i/N \rfloor$, where i is the logical page number of D_i. Therefore, the stripe S_j is composed of $(D_{N\cdot j}, D_{N\cdot j+1}, \ldots, D_{N\cdot(j+1)-1}, P_j)$, where P_j is the parity of S_j and is equal to $D_{N\cdot j} \oplus D_{N\cdot j+1} \oplus \cdots \oplus D_{N\cdot(j+1)-1}$. The operator $\oplus$ represents the

exclusive ORing (XORing). The chip number of each data is determined by the parity allocation structure in Fig. 4.

In order to update user data D_0, RAID-5 requires the three following steps:

1. read D_0 and P_0,
2. compute a new parity P_0' ($P_0' = D_0 \oplus D_0' \oplus P_0$), and
3. write D_0' and P_0'.

To change D_0 to D_0', RAID-5 first reads the old user data D_0 and the old parity P_0. Second, it generates a new parity P_0'. Finally, the new user data D_0' and the new parity P_0' should be placed into the storage. Therefore, the total update cost is $2 \cdot T_{read} + 2 \cdot T_{write}$, and the parity handling overhead is $2 \cdot T_{read} + T_{write}$, where T_{read} and T_{write} are the read and write costs of the flash memory, respectively. That is, the total write cost can increase over 50 percent compared to the case of no redundancy. Therefore, RAID-5 results in poor write performance in flash storage because even a small random write may incur a parity update.

A more advanced RAID technique such as RAID-6 provides multiple parity data to recover from multiple failures. In addition, there are more powerful RAID codes [13] for higher performance and reliability. However, we focus on only RAID-5 using parity since the benefit of using the MLC flash memory will be diminished if the redundant area is large.

3 Related Works

3.1 HDD-Based RAID

Several techniques such as *parity logging* [14], *floating parity* [15], and *fast write* [16] have been proposed to reduce the overhead for small writes in a HDD-based RAID-5. The *parity logging* technique writes parity updates into a dedicated log disk instead of updating the parity in the parity disk. When the log disk fills up, the logged parity update images are applied to the out-of-date parity. Such a logging technique converts small writes of parity into large sequential writes and combines successive updates of parity into a single update in the disk array. The *floating parity* technique remaps dynamically parity blocks within disk cylinders to reduce the rotational latency between reading and writing parity.

Both parity logging and floating parity attempt to reduce the disk seek time overhead of parity updates. In both schemes, the old data must be read from the disk to calculate parity. However, our proposed RAID-5 technique focuses on the flash memory SSDs. Our technique reduces the number of read operations for parity update by using a partial parity scheme while the previous schemes use full parity schemes.

The *fast write* scheme uses a nonvolatile write buffer to reduce write latency. The data in the write buffer and the corresponding parity are written at disks in the background. While the parity should be updated at disk when the data is evicted from the write buffer in the *fast write* scheme, our proposed scheme delays the parity update even when the related data are evicted from the write buffer. Moreover, our scheme uses partial parities to reduce flash read operations exploiting the special feature of flash memory.

3.2 Flash Memory SSD

Many commercial products of flash memory SSD have been introduced by several companies such as Samsung [17], Intel [18], etc. These products are composed of multiple flash chips to provide a large storage capacity. To increase the I/O bandwidth, the user data are interleaved over multiple flash chips using a multichannel and multiway controller, an architecture similar to RAID-0.

Park et al. [3] proposed a multichannel and multiway controller for SSD that supports parallel write operations. The firmware intervention is minimized by automatic interleaving hardware logic. A hybrid mapping scheme is used in the internal FTL.

Kang et al. [19] proposed three techniques to exploit the I/O parallelism of SSD: striping, interleaving, and pipelining. The striping technique spreads a request across multiple channels. In the interleaving technique, several requests are handled in parallel using several channel managers. The pipelining technique overlaps the processing of two requests on a single channel.

Agrawal et al. [20] presented a range of design tradeoffs that are relevant to SSDs. They analyzed the tradeoffs using a trace-based disk simulator that can be customized to characterize different SSD organizations. They considered two interleaving schemes, i.e., async mode and sync mode. In async mode, multiple flash chips operate independently, and there are no relationships between pages assigned to different flash chips. However, in sync mode, multiple flash chips simultaneously handle one I/O request on the stripe. The pages in a stripe are sequential. Generally, the async mode can provide a better I/O performance, but it requires a larger mapping table than that of the sync mode.

Shin et al. [21] extended the SSD simulator of [20] and evaluated various page striping methods for the sync mode. The page striping method determines the order of pages within a stripe. They showed that narrow striping has an inferior performance to that of wide striping, and that using the striping unit of a block is worse than using that of a page.

These studies did not address the redundancy issue of SSDs. Instead, similar to RAID-0, their proposed algorithms use an array of flash chips and stripe (interleave) data across the arrays only to improve parallelism and throughput.

3.3 SSD with Redundancy

More recently, several approaches have been proposed to improve both the performance and reliability of SSDs using redundancy. Greenan et al. [22] proposed a RAID-4 SSD architecture that uses a page-level interleaving and a nonvolatile RAM (NVRAM) to hold the parity temporarily in order to prevent frequent parity updates for write requests. All parity updates for a page stripe are delayed in the NVRAM until all of the dependent data have been written to flash memory. As a result, it reduces the parity update overhead. However, the technique does not reduce the parity calculation overhead (i.e., reading the old data and the old parity) since it must calculate a new parity for each write request. Therefore, the parity overhead cost is not negligible.

The FRA [23] scheme also uses a delayed parity update scheme which can reduce the parity write frequency for multiple write requests to the same area. The parity is

calculated and written to the flash storage during idle time. The main difference between this scheme and that of [22] is that FRA does not use the NVRAM to store the parity data. Instead, it uses the dual-mapping scheme in the address mapping table of FTL to identify which parity has been delayed. However, FRA has a critical drawback in terms of reliability; there is no method for recovering failed data for which the parity update is delayed.

4 Partial-Parity-Based Delayed Parity Update

When there is an update request, FTL generally does not erase or update old data; instead, the data are invalidated due to the erase-before-write constraint of flash memory. The invalidated data can be utilized as implicit redundant data. The proposed delayed parity update scheme is designed to exploit the implicit redundant data in order to reduce the parity handling overhead.

We apply page-level striping since it shows better performance than does block-level striping [21]. When there is a write request from the host, the RAID SSD controller determines a stripe number and a chip number based on the logical page number and sends the data to the determined flash chip. The normal RAID controller generates the parity data for the stripe and writes it to the parity flash chip of the stripe. However, the proposed scheme delays the parity data update and stores it on a special device called a *partial parity cache* (PPC). The stored parity is a *partial parity* because it is generated with only partial data of the stripe. This is a main difference from the delayed parity scheme in [22], which stores full parities in the parity cache, and thus, requires many read operations to calculate the full parities.

Using the partial parity, we can reduce the parity generation overhead. Instead, we maintain the old version of the updated data, which is implicit redundant data. In the case of chip or page failures, we recover the failed data with the partial parity or the old version of other data. The delayed parity is written to a flash chip when there is no free space in the PPC. This step is called a *parity commit*.

4.1 Partial Parity Cache

The partial parity cache temporarily stores the delayed parities. In order to avoid losing the parities stored in the PPC at sudden power failures, it must be implemented with an NVRAM. We can use a storage class memory (SCM) [24], such as PRAM and MRAM, or a battery-backed RAM that has a redundant battery to protect against an external power failure. The capacity of the battery should be large enough to flush all delayed parities to the flash chips. Fig. 5 shows the structure of the PPC including the information on the parities that are not yet written to the flash chips. The PPC has M number of entries, each of which has a stripe index, a partial parity bitmap, and a partial parity. The bitmap represents the data indices associated with the partial parity. For example, if the bitmap of stripe S_j is "0110" for a $4+1$ RAID-5 structure, its partial parity is made up of the updated pages of $\{D_{4j+1}, D_{4j+2}\}$. The stripe whose up-to-date parity is not written to a flash chip is called an *uncommitted stripe*. We denote the set of associated data of the parity P_j as $\pi(P_j)$.

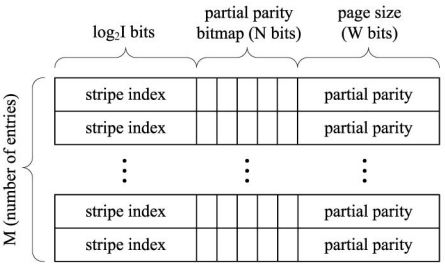

Fig. 5. Partial parity cache.

The size of the PPC can be estimated to be $M(\log_2 I + N + W)$ bits, where I, N, and W represent the total number of stripes in SSD, the number of data flash chips (excluding the extra parity chip), and the bit-width of one page, respectively.

4.2 Partial Parity Creation and Updating

When an update request changes D_i into D'_i whose stripe is S_j, a partial parity is created or updated in the PPC in the following three cases:

1. If there is no corresponding partial parity of the target logical stripe in the PPC (i.e., $S_j \notin \text{PPC}$), a new partial parity $\tilde{P}_j$ should be inserted. The value of $\tilde{P}_j$ is same as D'_i. There is no flash memory I/O overhead ($C_{overhead} = 0$).
2. If there is a corresponding partial parity of the target logical stripe in the PPC, but the partial parity is not associated with the old version of the data to be written (i.e., $S_j \in \text{PPC} \wedge D_i \notin \pi(\tilde{P}_j)$), a new partial parity is calculated by XORing the old partial parity and the new data (i.e., $\tilde{P}_j = \tilde{P}_j \oplus D'_i$). There is no flash memory I/O overhead ($C_{overhead} = 0$).
3. If there is a corresponding partial parity of the target logical stripe in the PPC, and the partial parity is associated with the old version of the data to be written (i.e., $S_j \in \text{PPC} \wedge D_i \in \pi(\tilde{P}_j)$), a new partial parity is calculated by XORing the old partial parity, the old data, and the new data (i.e., $\tilde{P}_j = \tilde{P}_j \oplus D_i \oplus D'_i$). One flash memory read cost is invoked ($C_{overhead} = T_{read}$).

For example, Fig. 6 shows the change in the PPC when the host sends the update requests on data D_1 and D_2. The RAID SSD is composed of four data chips and one spare chip. We assume that each flash chip is composed of 10 blocks (i.e., $B_0, B_1, \cdots, B_9$) and each block has four pages. Before the update requests, D_1 and D_2 were written to physical page number (PPN) 40 of chip 1 and PPN 80 of chip 2, respectively. Their parity datum P_0 has been written to PPN 160 of chip 4. Initially, the PPC has no entry.

The RAID SSD controller first writes the new data D'_1 at PPN 42 of chip 1 while invalidating D_1 at PPN 40. Since the corresponding parity datum P_0 is not updated immediately in the delayed parity update scheme, the parity remains unchanged in flash chip 4. Instead, a partial parity $\tilde{P}_0$ for D'_1 is created in the PPC since there is no partial parity of the logical stripe $S_0 = (D_0, D_1, D_2, D_3, P_0)$ in the PPC (case 1).

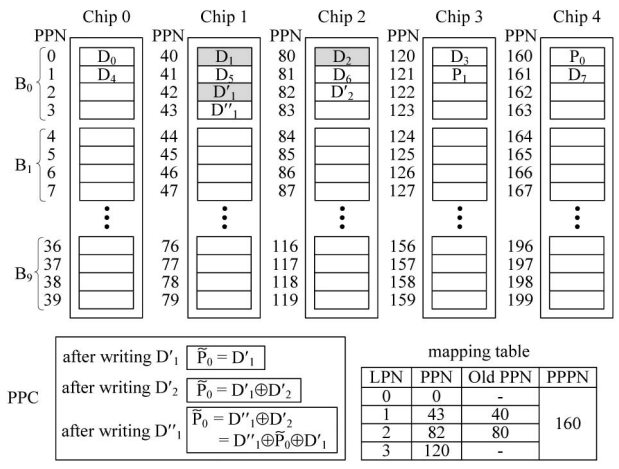

Fig. 6. Partial parity creation and updating.

$\tilde{P}_0$ is equal to the data D'_1. While the normal RAID-5 algorithm should read both the old data D_1 and the old parity P_0 to calculate the full parity, our scheme does not require read operations to generate the parity data.

The mapping table for the translation between the LPN and the PPN is also shown in Fig. 6. The table manages a PPN, an old PPN, and a physical parity page number (PPPN) for each LPN. The PPPN is the physical page number on which the parity data of a stripe are written. The old PPN points to the physical page on which the old version of the logical page is contained, where the old version is associated with the parity data written to the PPPN. Therefore, the old PPN value of LPN 1 is 40 after the update of D_1.

Generally, flash storage does not maintain the old PPN of a logical page. Instead, the garbage collector (GC) maintains the list of all invalid flash pages so as to reclaim them when there is no sufficient free space. However, our scheme regards the invalid pages as implicit redundant data that can be used to recover failed data in the stripe since the old data are associated with the out-of-date parities. In this paper, we call such invalid but useful data *semivalid data* since it is valid data in terms of failure recovery. The FRA technique proposed in [23] also uses the old PPN information in its dual-mapping scheme. However, the FRA uses the information to identify delayed parities without an NVRAM cache for the delayed parity. If the old PPN and the PPN for an LPN are different, the FRA detects that the parity of the LPN is not written to the flash chip. However, our scheme uses this information to recover failed data.

Since the mapping table is maintained at SRAM, its space overhead should be small. The PPPN field is necessary to manage RAID-5 striping architecture even when the partial parity scheme is not used. The only additional overhead invoked by the proposed scheme is the old PPN field. To minimize the space overhead, we allocate the memory space for the old PPN dynamically only when the corresponding LPN is one of uncommitted stripes, and use a hashing function to map between an LPN and its old PPN. The allocated space is freed when the LPN is committed. The required number of old PPNs is the same as the number of partial parities in the partial parity cache. Therefore, the maximum memory space for old PPNs is limited by the size of PPC.

When the host sends the update request on LPN 2 with the new data D'_2, the SSD controller writes it to chip 2 and updates the partial parity $\tilde{P}_0$ by XORing the old value of $\tilde{P}_0$ and D'_2 since there is the corresponding partial parity $\tilde{P}_0$ in the PPC but $D_2 \notin \pi(\tilde{P}_0)$ (case 2). It can be determined whether a partial parity is associated with a page by examining its partial parity bitmap information in the PPC. After the update of D_2, the old PPN of LPN 2 becomes 80.

If the host sends another update request on LPN 1 with the new data D''_1, D'_1 is invalidated and the partial parity $\tilde{P}_0$ is updated by XORing $\tilde{P}_0$, D''_1 and D'_1 (case 3). This case invokes one read operation for D'_1. The old PPN of LPN 1 is unchanged since the old parity P_0 is associated with D_1 at PPN 40.

4.3 Partial Parity Commit

There are two kinds of parity commits for which the delayed parity should be written to the flash chips.

- *Replacement Commit*: when there is no free space in the PPC for a new partial parity after handling several write requests, one of partial parities should be replaced.
- *GC Commit*: before the GC erases the semivalid pages of the uncommitted stripe, the corresponding delayed parity should be committed.

Since the PPC has only partial information, the semivalid data are needed to cope with failure. Therefore, we should commit the corresponding partial parity before the GC erases semivalid data.

To commit the partial parity, the RAID controller should first build the full parity with the pages that are not associated with the partial parity. To reduce the parity commit cost, we should consider the number of associated pages of a partial parity $\tilde{P}_j$, which is equal to $|\pi(\tilde{P}_j)|$ (i.e., the number of elements in $\pi(\tilde{P}_j)$). The partial parity commit operations can be divided into two cases for $N+1$ RAID-5 SSD:

- When $|\pi(\tilde{P}_j)| \geq \lceil N/2 \rceil$, the full parity is generated by XORing the partial parity and the nonupdated pages of the stripe. The maximum number of flash memory reads for the nonupdated pages is $\lceil N/2 \rceil$.
- When $|\pi(\tilde{P}_j)| < \lceil N/2 \rceil$, the full parity is generated by XORing the partial parity, the old full parity, and the old data of the associated pages. The maximum number of flash memory reads for both the old full parity and the old data is $\lceil N/2 \rceil$.

For instance, in Fig. 7a, the set of associated pages of partial parity $\tilde{P}_0$ is $\{D''_1, D'_2, D'_3\}$. The nonupdated data D_0 must be read in order to calculate the full parity. After writing the full parity at PPN 162, the old PPN information of the strip is cleared, and the value of the PPPN is updated. However, in Fig. 7b, the partial parity $\tilde{P}_0$ has only one associated page D'_1. Therefore, the full parity is calculated with D_1, P_0, and $\tilde{P}_0$. The physical location of D_1 can be determined from the old PPN field of the mapping table.

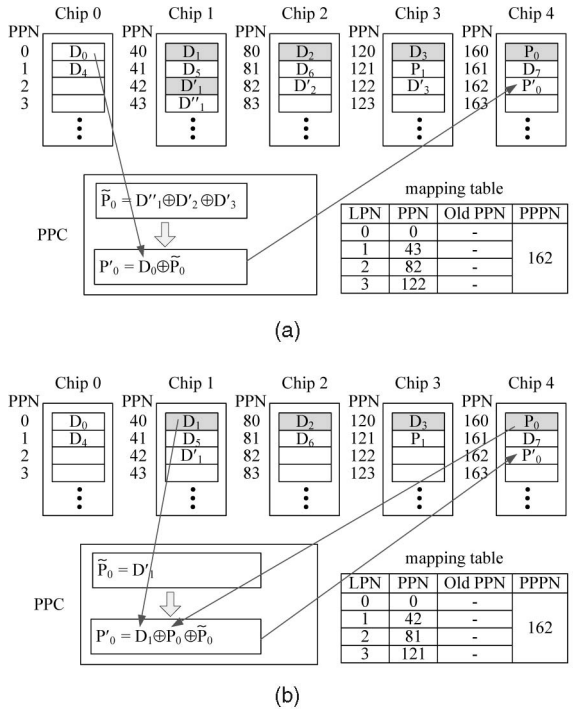

Fig. 7. Partial parity commit. (a) Number of associated pages $\geq \lceil N/2 \rceil$. (b) Number of associated pages $< \lceil N/2 \rceil$.

Therefore, the parity commit cost $C_{commit}(\tilde{P}_j)$ for a partial parity $\tilde{P}_j$ is as follows:

$$(N - |\pi(\tilde{P}_j)|)T_{read} + T_{write} \text{ if} |\pi(\tilde{P}_j)| \geq \lceil N/2 \rceil$$
$$(|\pi(\tilde{P}_j)| + 1)T_{read} + T_{write} \quad \text{otherwise},$$

where N represents the number of parallel logical flash chips; $(N - |\pi(\tilde{P}_j)|)T_{read}$ or $(|\pi(\tilde{P}_j)| + 1)T_{read}$ represents the cost of page reads from the flash chips to generate the full parity. We can limit the number of flash reads for partial parity commit to $\lceil N/2 \rceil$. T_{write} is the flash write cost of the full parity. If N flash chips can be accessed in parallel in the RAID architecture, the read latency for multiple interleaved pages will be the same as that for one page. However, the read operations for multiple pages degrade the I/O bandwidth of SSD, especially when there are many other read requests from the host. Therefore, it is reasonable to count the number of pages to read when estimating the commit cost.

The PPC scheme generates the read requests only when a partial parity is committed while the previous schemes [22], [14] should read the old data at each data write to maintain the full parity. In addition, a partial parity includes most of the new data of the associated pages due to data access locality when it is committed, as shown in Fig. 7b. Then, the PPC scheme reads only a small number of pages in order to commit a partial parity. For the best case, the PPC scheme can commit a partial parity without any read operation if it includes all the associated pages.

4.4 Commit Cost-Aware PPC Replacement

When there is no free space in the PPC, we should commit at least one partial parity. Therefore, a partial parity replacement policy is required. General cache systems use the least-recently-used (LRU) replacement policy, which will also provide a high hit ratio to the PPC. However, for the parity cache, we should consider the parity commit costs that are different depending on the number of associated pages of the partial parity. Therefore, we propose a commit cost-aware replacement policy that considers both the recency and commit cost of a parity. We use the following priority function to select a victim parity:

$$\alpha \cdot Prob_{update}(\tilde{P}_j) + (1 - \alpha) \cdot C_{commit}(\tilde{P}_j), \tag{1}$$

where $Prob_{update}$ and C_{commit} are the update probability and the commit cost of a parity, respectively, and α is the weight value between two metrics. The update probability can be approximated with the LRU ranking. As the priority value of a parity is smaller, it has a higher probability to be selected as a victim. By experimental analysis, we could determine a proper value for α under the given target workloads.

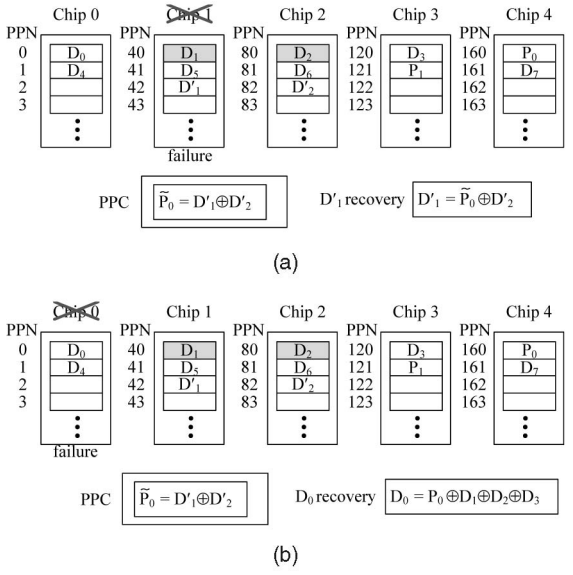

Fig. 8. Failure recovery. (a) A partial parity is associated with the failed page. (b) No partial parity associated with the failed page.

4.5 Chip Failure Recovery

The SSD controller can fail to read data from the flash memory due to page-level error, chip-level error, or flash controller-level error. The page-level error is generated when it is uncorrectable by the ECC. When there is a read failure on one flash chip, we can recover the failed data using its parity data. The recovery steps are divided into two cases as follows:

- When a delayed partial parity $\tilde{P}_j$ is associated with the failed page D'_i ($D'_i \in \pi(\tilde{P}_j)$), D'_i can be recovered using the partial parity $\tilde{P}_j$ and other associated pages in $\pi(\tilde{P}_j)$.
- When no delayed partial parity is associated with the failed page D_i, D_i can be recovered using the old full parity P_j and the associated pages in $\pi(P_j)$.

For example, assume that we failed to read the data D'_1 due to the failure of chip 1 as shown in Fig. 8a. The PPC has

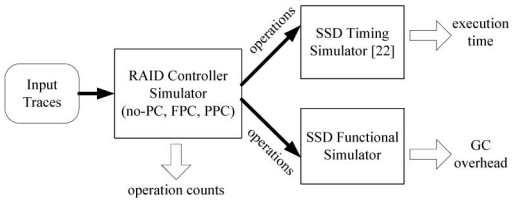

Fig. 9. RAID-5 SSD simulator.

an uncommitted partial parity $\tilde{P}_0$ that is generated with D'_1 and D'_2. Since the partial parity is associated with the failed page D'_1, we can recover the data by XORing $\tilde{P}_0$ and D'_2. D'_2 should be read from flash chip 2. If the data D_0 cannot be read due to the failure of chip 0, as shown in Fig. 8b, it cannot be recovered with the partial parity $\tilde{P}_0$ that is not associated with D_0 (case 2). In this case, the old parity P_0 and its associated old data are used. By XORing P_0, D_1, D_2 and D_3, the data D_0 can be recovered. D_1 and D_2 can be accessed with the old PPN information in the mapping table. This second case exploits the semivalid pages to recover the failed data.

4.6 Parity-Cache-Aware Garbage Collection

When there are many invalid pages in flash chips, the GC is invoked. It selects a victim flash block that may have many invalid pages. If the block also has valid pages, the GC moves them into other clean blocks, changes the page mapping information and erases the victim block for future uses. Even when the victim block has semivalid pages of the uncommitted stripe, the GC removes them since they are invalid pages in the flash memory. To avoid losing the semivalid data, we should commit the corresponding partial parity before the GC erases them (GC commit). The GC should check whether the invalid page is a semivalid page by examining the mapping table. If the physical address of the invalid page is found at the old PPN field of the mapping table, the page is a semivalid page. Even when the proposed delayed partial parity scheme is not used, the mapping table should be accessed by the GC to read and update the PPN values. Therefore, the identification of semivalid data imposes no significant additional overhead. In addition, since the maximum number of semivalid pages is not too large, the timing cost to identify the semivalid pages is negligible.

Since the GC commit invokes several flash memory read and write operations, it is better to avoid this case when possible. For this purpose, we propose the parity cache-aware victim selection policy for GC. The general algorithm for the victim block selection of GC considers the page migration cost. The algorithm selects the block with the smallest number of valid pages because it invokes the lowest page migration cost during the GC. However, in the parity cache-aware policy, the GC commit cost is taken into account additionally. This scheme uses the following equation to estimate the GC cost of block B:

$$GC_{cost}(B) = C_{migration}(B) + C_{commit}(B), \quad (2)$$

where $C_{migration}(B)$ denotes the valid page migration cost of block B and $C_{commit}(B)$ denotes the sum of the GC commit costs invoked before erasing block B. Therefore, the parity cache-aware GC selects the victim block whose GC_{cost} is smallest among all of the victim candidates.

Generally, the victim block selected by the GC is not a recently-allocated block since the recently-allocated blocks have a large number of valid pages, and thus, require a large page migration overhead. Therefore, if the parity cache size is quite small, and thus, the partial parities remain in the parity cache during short intervals, compared to the garbage collection frequency, there is little possibility for a selected victim block to contain the semivalid data since the parity cache commits the stripes related to the victim block prior to the garbage collection.

TABLE 1
Parameters of SSD Timing Simulator

operation	latency	operation	latency
page read	0.025 ms	page write	0.2 ms
block erase	1.5 ms	page transfer	0.025 μs

5 Experiments

5.1 Experimental Setup

To evaluate the proposed scheme, we implemented a RAID-5 SSD simulator as shown in Fig. 9. It is composed of the RAID-5 controller simulator, the SSD timing simulator, and the SSD functional simulator. The RAID-5 controller simulator uses the disk I/O trace as an input. It forwards the data I/O operations to SSD simulators after inserting the parity handling operations between the normal operations. The logical address of each data I/O request is modified since parities should be inserted at every stripe. The RAID-5 controller simulator internally manages a write buffer and a parity cache.

To obtain the performance simulation results considering the parallel I/O architecture of SSD, we used the SSD timing simulator introduced in [20], exploiting the provided page striping function. However, this simulator cannot be configured to simulate various FTL mapping schemes and cannot generate the FTL-related information such as garbage collection overhead. To compensate for these drawbacks, we implemented our own SSD functional simulator that can be configured to use various address mapping schemes and can report the garbage collection overhead. The RAID controller simulator sends the same I/O operations to both SSD simulators. The timing simulator outputs the total execution times needed to handle the input requests, and the functional simulator outputs the garbage collection overhead. Table 1 shows the parameters used to configure the SSD timing simulator. The simulated RAID-5 SSD has $4 \sim 8$ parallel flash chips, and each chip has its own NAND controller. Sequential pages from host are distributed across multiple flash chips. The page size is 2 KB.

We used two real disk I/O traces, OSDB and Financial, and two benchmark traces generated by Iozone and Postmark. The OSDB trace was collected by executing the PostgreSQL 8.4.2 with the Open Source Database Benchmark suite in 32 multiuser configuration. The Financial

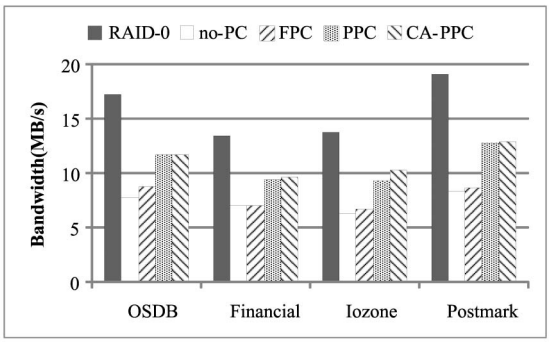

Fig. 10. I/O performances of no-PC, FPC, PPC and CA-PPC schemes (4+1 RAID-5).

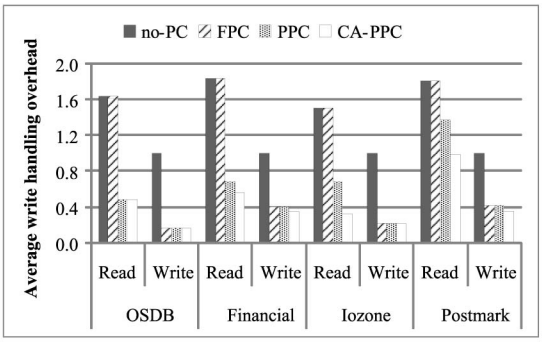

Fig. 11. Average write handling overheads $(4+1$ RAID-5).

trace is the OLTP application trace provided in [25]. While the Iozone trace has many sequential write requests, random requests are dominant in other traces.

We compared the I/O performances of the four kinds of RAID-5 SSD schemes: no-PC, FPC, PPC, and CA-PPC. The no-PC scheme does not use the parity cache, and thus, it is the same as the native RAID-5 scheme. The FPC scheme uses the parity cache that maintains the delayed full parities as proposed in [22]. The PPC scheme is our proposed scheme that exploits the partial parity cache. The CA-PPC scheme is the same as the PPC scheme except that it uses the commit cost-aware PPC replacement policy explained in Section 4.4. We examined the optimal value for α in (1) using the target workloads. The performances of the CA-PPC scheme were best when $\alpha = 0$, for all workloads. Therefore, we used the optimal value in all of the CA-PPC experiments.

We assumed that the RAID-5 controller has a nonvolatile memory space for the write buffer and the parity cache. The write requests from the host are first stored in the write buffer. When there is no free space in the buffer, several pages that belong to one stripe are selected as victims and are written to the flash chips. Then, the parity of the stripe is written to the flash chip or to the parity cache, depending on the RAID-5 controller scheme. Each size of both the write buffer and the parity cache is 32 KB.

5.2 Overall Performance

Fig. 10 shows the write bandwidth (MB/s) values of the evaluated schemes. The bandwidths of RAID-0 scheme are also provided for comparison. (Note that there are many idle intervals in the input traces, and thus, the bandwidth values are quite lower than the peak I/O bandwidths of real storage devices.) Since the RAID-5 schemes should manage the redundant data, they provide inferior results to those of RAID-0. The average bandwidths normalized to RAID-0 scheme are 0.46, 0.49, 0.68, and 0.70 for the no-PC, FPC, PPC, and CA-PPC schemes, respectively. PPC improves the performance by 47 percent and 38 percent on average compared to those of no-PC and FPC, respectively.

To analyze the performance improvement of the PPC scheme, as shown in Fig. 11, we measured the average write handling overhead that is required to handle the parity for each write request. The no-PC scheme theoretically requires two read operations and one write operation additionally to write the parity for each page write request. However, it generates smaller numbers of additional read operations $(1.5 \sim 1.8)$ due to the write buffer, which invokes no read operation in calculating the parity if it has all the pages belong to a stripe. Since the FPC scheme maintains full parities in its parity cache, the number of read operations is the same as that of the no-PC scheme. However, the number of write operations is reduced due to the delayed parity write scheme of FPC.

The PPC scheme requires read operations for the partial parity update, and both read and write operations for the partial parity commit. The average numbers of the additional reads and writes for each page written in the PPC scheme are 0.8 and 0.3, respectively. The CA-PPC scheme shows smaller numbers of read and write operations compared to those of the PPC scheme for most of workloads since CA-PPC selects a victim partial parity considering the commit cost. However, there are no differences between PPC and CA-PPC in the numbers of operations for the OSDB workload. This is because most of partial parities in the PPC have similar commit costs for the OSDB workload.

5.3 The Effect of Commit-Cost-Aware Scheme

Fig. 12 shows the effect of α in (1) on the performance of CA-PPC scheme. Since the parity commit cost of OSDB does not change depending on α, the result of OSDB is omitted. As the value of α decreases, the number of flash read operations for parity commit decreases while the number of flash write operations increases. However, the changes of write counts are insignificant compared to the changes of read counts. As a result, the performances of the CA-PPC scheme are best when $\alpha = 0$, i.e., only the commit cost of the victim partial parity is considered ignoring the update probability. If α is 1, CA-PPC is same as PPC. From the result, we can know that the update probabilities of partial

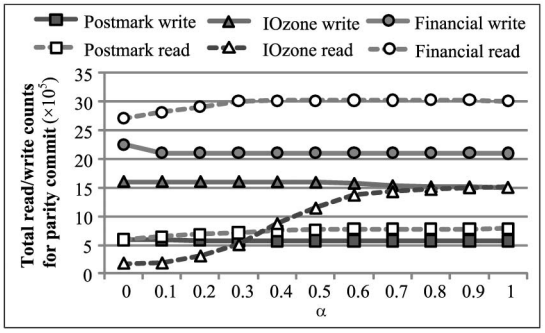

Fig. 12. Parity commit costs varying the value of α in CA-PPC.

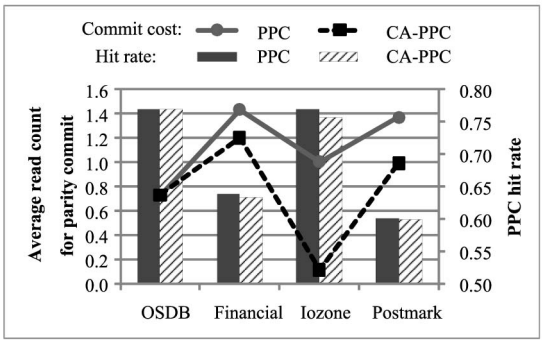

Fig. 13. Comparisons between PPC and CA-PPC.

parities have no significant effects on the total parity commit cost, and thus, it is better to use a small value for α.

The OSDB workload requires similar numbers of read counts both when $\alpha = 0$ and $\alpha = 1$. Although the OSDB has a random IO pattern, most of the requests have the size of 8 KB (four pages), which is same to the stripe size in the $4 + 1$ RAID-5. Therefore, there are no large differences between the commit costs of partial parities, and thus, the CA-PPC has no chance to reduce the commit cost further.

Fig. 13 shows the superiority of the CA-PPC scheme over that of the PPC scheme. It compares the average parity commit costs of the PPC and CA-PPC schemes. We counted the number of read operations required to calculate the full parity during the partial parity commit. The CA-PPC scheme reduces the commit cost significantly compared to that of the PPC scheme.

The sequential request-dominant Iozone workload has quite a smaller commit cost than do the random request-dominant workloads (Financial and Postmark). Especially, the Iozone workload requires almost no read operation for the CA-PPC scheme since most of the selected victim parities in the PPC are full parities due to the highly sequential access pattern.

Fig. 13 also shows the hit rates of the partial parity cache. Since CA-PPC prefers the victim parity with the smallest commit cost to the least-recently-used victim parity, the hit rates of CA-PPC are lower than those of PPC. However, the differences in the hit rates are insignificant due to the special behavior of CA-PPC. CA-PPC tends to give a high priority to be replaced to sequential request since it has a low commit cost. Generally, the sequential request has a low temporal locality. Therefore, it is probable for CA-PPC to select a victim that will not be accessed within the near future.

5.4 The Effect of Architectural Parameters

5.4.1 Stripe Size

Fig. 14 shows the average write handling overhead of the PPC scheme when the number of parallel flash devices is changed, thus changing the size of a strip. We counted the number of write operations required for the parity commit, the number of read operations required for updating the partial parity, and the number of read operations required for the parity commit. The number of data pages with which one parity is associated increases as the number of parallel flash devices increases. Therefore, the number of additional

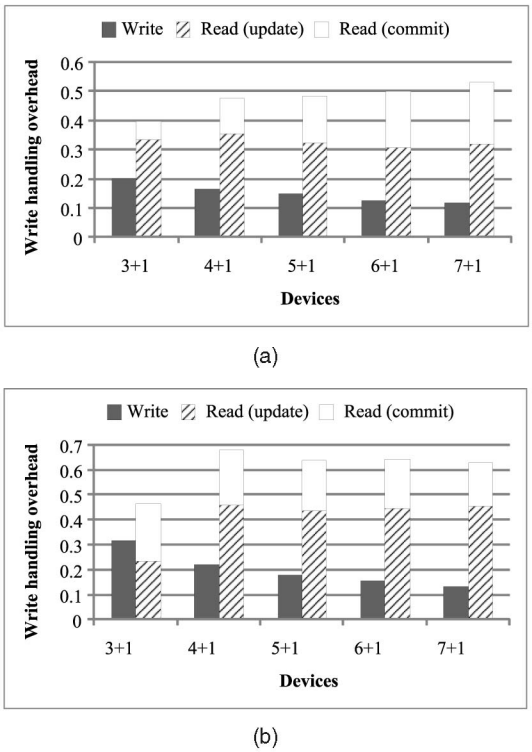

Fig. 14. Write handling overhead of PPC while varying stripe size (write buffer = 32 KB, PPC = 32 KB). (a) OSDB. (b) Iozone.

write operations per one data page write decreases in both the OSDB and Iozone workloads. However, the changes in the number of read operations are different depending on the access pattern of the workload. While the OSDB workload with a random access pattern has a large number of read operations when the stripe size is large, the Iozone workload with a sequential access pattern shows only a small change as the stripe size changes.

Fig. 15 shows how the performance changes when the number of parallel flash devices changes. As the parallelism increases, the performance increases slightly due to the decrease in the parity handling overhead.

5.4.2 Parity Cache Size

We evaluated the effects of the parity cache size on the proposed schemes. Fig. 16a shows the parity commit costs

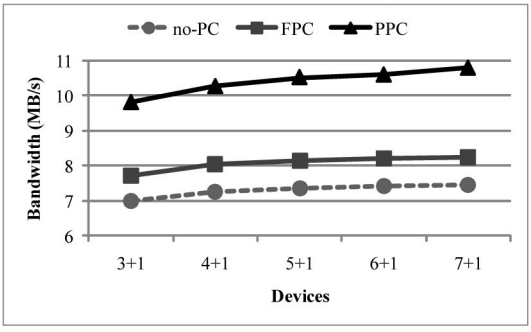

Fig. 15. Performance comparison while varying stripe size (Iozone, write buffer = 32 KB, PPC = 32 KB).

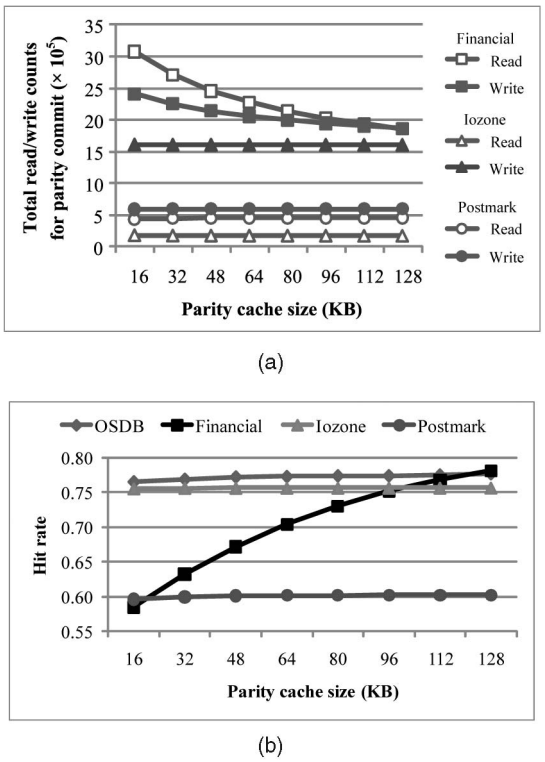

Fig. 16. The effects of parity cache size in CA-PPC scheme ($4+1$ RAID-5, write buffer $= 32$ KB). (a) Commit cost. (b) PPC hit rate.

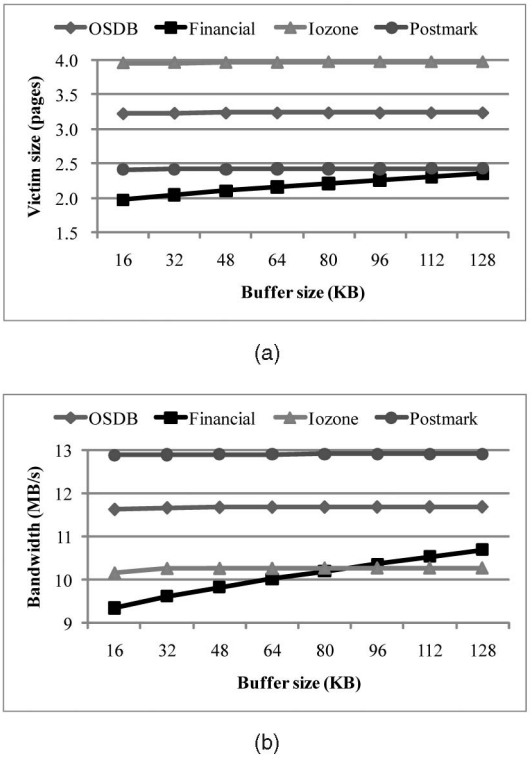

Fig. 17. The effects of write buffer size in CA-PPC scheme ($4+1$ RAID-5, PPC $= 32$ KB). (a) Average victim size. (b) Bandwidth.

of CA-PPC while varying the size of the parity cache. While the Financial workload is sensitive to the parity cache size due to its high temporal locality, other workloads with low localities cause no significant changes to the commit costs as the parity cache size changes. Fig. 16b shows the hit rates of the parity caches. The OSDB and Iozone workloads have higher hit rates. There is no significant change in all workloads except for Financial as the parity cache size increases. From this result, we can know that the proposed delayed parity scheme can improve the I/O performance even with a small parity cache.

5.4.3 Write Buffer Size

We also evaluated the effects of the write buffer size. Fig. 17a shows the changes to the average victim size while varying the buffer size. The victim size represents the number of pages belonging to one stripe and are evicted as a group from the write buffer. For an $N+1$ RAID-5 architecture, the maximum value of the victim size is N. A large victim size is favorable for reducing the parity handling overhead. For example, if the victim size is four in the $4+1$ RAID-5 architecture, the RAID controller can calculate the full parity without any flash I/O operations. Since the Iozone workload has a sequential access pattern, its victim size approaches four pages. However, the victim size is small for the Financial workload due to its random access pattern, but it increases as the buffer size increases since a large buffer can merge more pages of a stripe before they are evicted.

Fig. 17b shows the performance changes while the buffer size varies. Only the Financial workload shows a significant improvement in the bandwidth since it has a high locality. However, we should consider the tradeoff between performance improvement and hardware cost of the write buffer that should be implemented with an NVRAM.

5.5 The Effect on FTL

To show that the proposed techniques also reduce the garbage collection overheads and the erase counts of blocks in flash memory, we performed several experiments with the SSD functional simulator that models the RAID-5 SSD using the hybrid mapping FTL [7]. We assumed that each flash chip has $40 \sim 240$ log blocks.

5.5.1 Garbage Collection Overhead

Fig. 18a shows the garbage collection overhead of the PPC scheme varying the number of log blocks. As the number of log blocks increases, the garbage collection overhead decreases. Generally, the hybrid mapping FTL invokes frequent garbage collections for random access patterns. Since the OSDB and Financial workloads have random access patterns, the overheads are reduced significantly when sufficient log blocks are provided.

Fig. 18b shows the garbage collection overheads of the PPC scheme normalized to those of the no-PC scheme. The PPC scheme reduces the overhead by $12 \sim 40$ percent compared to that of the no-PC scheme. Since the PPC scheme invokes a smaller number of write operations, it has a less garbage collection overhead. Moreover, frequent parity writes of the no-PC scheme have an adverse effect on the log block utilization since they disrupt the locality of the write requests. The Financial workload has a high temporal locality, and the Iozone workload has a high spatial locality. Therefore, these workloads show large improvements in the garbage collection overhead for the PPC scheme compared

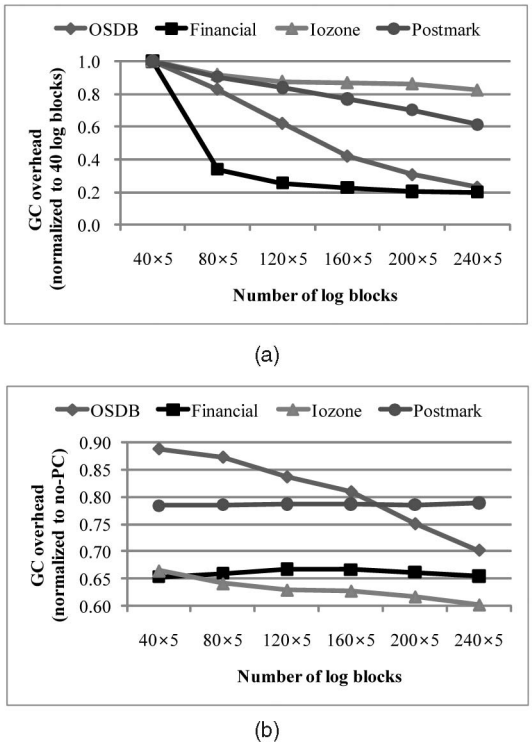

Fig. 18. GC overheads of PPC in a hybrid mapping FTL ($4 + 1$ RAID-5, write buffer $= 32$ KB, PPC $= 32$ KB). (a) GC overhead normalized by 40×5 log blocks. (b) GC overhead normalized by no-PC GC overhead.

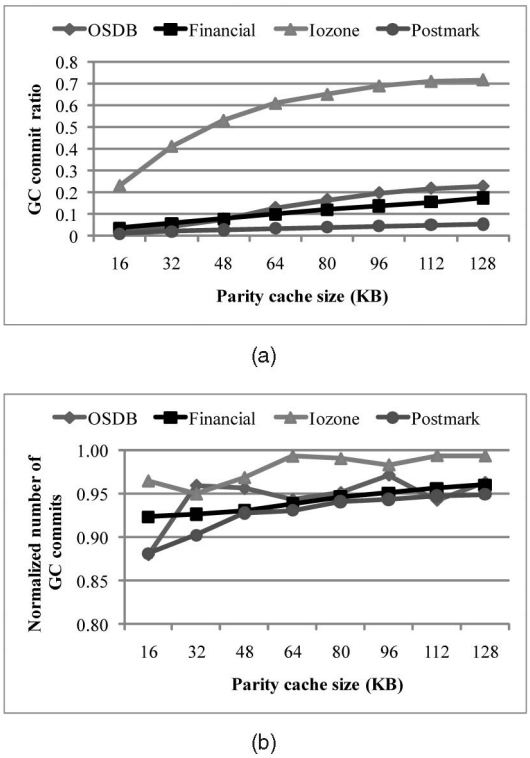

Fig. 19. GC commit cost of PPC in a hybrid mapping FTL ($4 + 1$ RAID-5, log blocks $= 160 \times 5$). (a) The proportion of GC commit. (b) GC commits in PC-aware GC.

to its use in the no-PC scheme. For the OSDB workload, the difference between the no-PC and PPC schemes in the GC overhead increases as the number of log blocks increases since the OSDB workload has random accesses for a large address space.

5.5.2 GC Commit Cost

Fig. 19 shows the GC commit costs of the PPC scheme explained in Section 4.6. Fig. 19a illustrates the ratio between the GC commit cost and the total commit cost (i.e., the GC commit cost plus the replacement commit cost). When the parity cache size is small, the GC commit costs are small. However, as the parity cache size increases, the portion of the GC commit increases because there are a large number of semivalid pages in the log blocks. The proportion of the GC commit is greater than 70 percent for the Iozone workload when the parity cache size is 128 KB. The workloads with sequential access patterns have larger ratios than do the workloads with random access patterns.

Fig. 19b shows the number of GC commits of the parity-cache-*aware* garbage collection (PC-aware GC) normalized to those of the parity-cache-*unaware* garbage collection. The PC-aware GC reduces the GC commit costs significantly when the parity cache size is small (up to 12 percent). As the parity cache increases, it is difficult for the PC-aware GC to find the victim block that will not invoke the GC commits, since most of the log blocks have many semivalid pages when the size of the parity cache is large. Therefore, the difference between the two schemes decreases as the parity cache increases.

5.5.3 Erase Count

NAND flash memory does not support an overwrite operation because of its write-once nature. Before writing new data into a block, the block must be erased by garbage collector. The total number of erasures allowed for each block is typically limited to between 10,000 and 100,000 cycles, which determines the lifespan of flash memory SSD. To increase the lifespan of SSD, the number of erasures of each block should be minimized. Fig. 20 compares the erase counts of several schemes. The delayed parity schemes require smaller numbers of erasures compared to the no-PC scheme since they reduce the number of write requests. However, there is no significant difference on the erase counts between FPC, PPC, and CA-PPC.

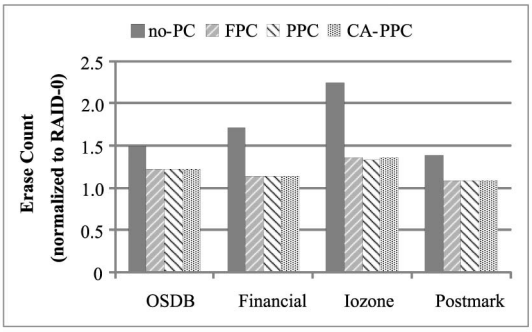

Fig. 20. Normalized erase counts ($4 + 1$ RAID-5, log blocks $= 160 \times 5$).

6 Conclusion

To build high-performance, reliable and large-scale storage systems, RAID technologies are popular. They use the interleaving technique that distributes sequential pages across multiple parallel operating disks for high performance, and they utilize redundant data to cope with disk failures. We proposed efficient RAID techniques for reliable flash memory SSDs. In order to reduce the I/O overhead for parity handling in the RAID-4 or RAID-5 SSD, the proposed scheme uses the delayed parity update and partial parity caching techniques. The delayed parity update technique reduces the number of flash memory write operations. The partial parity caching technique exploits the implicit redundant data of flash memory to reduce the number of read operations required to calculate the parity. These techniques also reduce the garbage collection overheads in flash memory. Even with a small parity cache, the proposed scheme significantly improves the I/O performance of flash memory SSDs.

In future works, we plan to build a real RAID-5 SSD and use it to evaluate the performance gain of the proposed techniques. In addition, we will study a flash-aware parity handling scheme for RAID-6 architecture that uses multiple parities for each stripe.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0010387).

References

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating Server Storage to SSDs: Analysis of Tradeoffs," *Proc. Fourth ACM European Conf. Computer Systems (EuroSys '09)*, pp. 145-158, 2009.

[2] D. Reinsel and J. Janukowicz, "Datacenter SSDs: Solid Footing for Growth," http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf, 2008.

[3] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, "A High Performance Controller for NAND Flash-Based Solid State Disk (NSSD)," *Proc. 21st IEEE Non-Volatile Semiconductor Memory Workshop*, pp. 17-20, 2006.

[4] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145-185, 1994.

[5] Y. Kang and E.L. Miller, "Adding Aggressive Error Correction to a High-Performance Compressing Flash File System," *Proc. Seventh ACM Int'l Conf. Embedded Software (EMSOFT '09)*, pp. 305-314, 2009.

[6] "Flash memory K9XXG08XXM," technical report, Samsung Electronics Co, LTD., Mar. 2007.

[7] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash Systems," *IEEE Trans. Consumer Electronics*, vol. 48, no. 2, pp. 366-375, May 2002.

[8] 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. Embedded Computing Systems*, vol. 6, no. 3, 2007.

[9] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Sixth ACM and IEEE Int'l Conf. Embedded Software (EMSOFT '06)*, pp. 161-170, 2006.

[10] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications," *ACM Trans. Embedded Computing Systems*, vol. 7, no. 4, 2008.

[11] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Rev.*, vol. 42, no. 6, pp. 36-42, 2008.

[12] C. Dirik and B. Jacob, "The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," *Proc. Int'l Symp. Computer Architecture*, pp. 279-289, 2009.

[13] D. Kenchammana-Hosekote, D. He, and J.L. Hafner, "Reo: A Generic RAID Engine and Optimizer," *Proc. Fifth USENIX Conf. File and Storage Technologies (FAST '07)*, pp. 31-31, 2007.

[14] D. Stodolsky, G. Gibson, and M. Holland, "Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93)*, pp. 64-75, 1993.

[15] J. Menon, J. Roche, and J. Kasson, "Floating Parity and Data Disk Arrays," *J. Parallel and Distributed Computing*, vol. 17, nos. 1/2, pp. 129-139, 1993.

[16] J. Menon and J. Cortney, "The Architecture of a Fault-Tolerant Cached RAID Controller," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 76-87, 1993.

[17] Samsung, http://www.samsungssd.com, 2010.

[18] Intel, http://www.intel.com/design/flash/nand, 2010.

[19] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A Multi-Channel Architecture for High-Performance NAND Flash-Based Storage System," *J. Systems Architecture*, vol. 53, no. 9, pp. 644-658, 2007.

[20] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf. (USENIX '08)*, pp. 57-70, 2008.

[21] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu, "FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications," *Proc. 23rd Int'l Conf. Supercomputing (ICS '09)*, pp. 338-349, 2009.

[22] K. Greenan, D.D.E. Long, E.L. Miller, T. Schwarz, and A. Wildani, "Building Flexible, Fault-Tolerant Flash-Based Storage Systems," *Proc. Fifth Workshop Hot Topics in System Dependability (HotDep '09)*, 2009.

[23] Y. Lee, S. Jung, and Y.H. Song, "FRA: A Flash-Aware Redundancy Array of Flash Storage Devices," *Proc. Seventh IEEE/ACM Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS '09)*, pp. 163-172, 2009.

[24] R.F. Freitas and W.W. Wilcke, "Storage-Class Memory: The Next Storage System Technology," *IBM J. Research and Development*, vol. 52, no. 4, pp. 439-447, 2008.

[25] UMass Trace Repository "OLTP Application I/O,"http://traces.cs.umass.edu/index.php/Storage/Storage, 2010.

Soojun Im received the BS and MS degrees in computer engineering from Sungkyunkwan University, Korea, in 2007 and 2010, respectively. He is currently a PhD student in the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include embedded software, file systems, and flash memory.

Dongkun Shin (S'99-M'07) received the BS degree in computer science and statistics, the MS degree in computer science, and the PhD degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000, and 2004, respectively. He is currently an assistant professor in the School of Information and Communication Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer of Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.