Delayed Partial Parity Scheme for Reliable and High-Performance Flash Memory SSD

Soojun Im
School of ICE
Sungkyunkwan University
Suwon, Korea
Email: lang33@skku.edu

Dongkun Shin
School of ICE
Sungkyunkwan University
Suwon, Korea
Email: dongkun@skku.edu

***Abstract*—The I/O performances of flash memory solid-state disks (SSDs) are increasing by exploiting parallel I/O architectures. However, the reliability problem is a critical issue in building a large-scale flash storage. We propose a novel Redundant Arrays of Inexpensive Disks (RAID) architecture which uses the delayed parity update and partial parity caching techniques for reliable and high-performance flash memory SSDs. The proposed techniques improve the performance of the RAID-5 SSD by 38% and 30% on average in comparison to the original RAID-5 technique and the previous delayed parity update technique, respectively.**

I. Introduction

Recently, due to the dramatic price reduction of flash memory, flash memory solid-state disks (SSDs) are replacing hard disk drives (HDDs) in the mass storage market [1]. SSDs are appreciated especially for energy efficiency over HDDs due to the absence of mechanical moving parts in SSDs.

However, the cost per bit of NAND flash memory is still high. In recent years, multi-level cell (MLC) flash memories have been developed as effective solutions for increasing the storage density and reducing the cost of flash storages. However, MLC flash memory has a slower performance and a less reliability than single-level cell (SLC) flash memory for the sake of its low cost. To enhance the performance of SSDs, we can use parallel I/O architectures such as multi-channel and interleaving [2], which increase I/O bandwidth by allowing concurrent I/O operations over multiple flash chips. However, the reliability problem is still a critical issue in building a large-scale flash storage.

Current NAND flash products ensure reliability by employing error-correcting codes (ECC). Traditionally, SLC flash memory uses single-bit ECC, such as Hamming codes. However, MLC flash memory shows a much higher bit-error rate (BER) than single-bit error-correcting codes can cover. As a result, codes with strong error-correction capabilities, like BCH or Reed-Solomon (RS) codes, are used. However, these ECC require a high hardware complexity and increase the read and write latencies.

Another approach for reliability is to leverage redundancy in storage level. Redundant Arrays of Inexpensive Disks (RAID) [3] 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 to improve sequential access performance. However, RAID-0 does not improve the reliability since it uses no redundant data. RAID-5 architectures are widely used to provide redundancy. In such architectures, one disk is reserved to store parity and thus multi-bit burst error in a page, block or device can be easily corrected. Parity requires negligible calculation time compared to error correction codes.

In order to implement 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 performance due to 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 in 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 SSDs in order to reduce the parity update cost. We use the delayed parity update and partial parity caching techniques. To reduce the number of write operations for parity updates, the proposed scheme delays the parity update which must accompany each page write in the original RAID-5 technique. The delayed parities are kept in the parity cache until they are written in 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.

II. Related Works

There have been intensive researches on the I/O parallelism of SSDs such as [2], [4], [5], [6] but they do not address the redundancy issue for SSDs. Similar to

978-1-4244-7153-9/10/$26.00

RAID-0, these techniques use an array of flash chips and stripe (interleave) data across the arrays only to improve parallelism and throughput.

More recently, several approaches are proposed to improve both performance and reliability of SSDs using redundant chip. Greenan *et al.* [7] proposed a RAID-4 SSD architecture. It uses an non-volatile RAM (NVRAM) to hold the parity temporarily in order to prevent frequent parity update for write requests. All parity updates for a page stripe are delayed in the NVRAM until all of the dependent data has been written in flash memory. As a result, it can reduce the parity update overhead.

FRA [8] scheme also uses a delayed parity update scheme. The parity is calculated and written to the flash storage at idle time. However, 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 know which parity has been delayed. FRA has a critical drawback in terms of reliability. There is no method to recover failed data of which parity update is delayed.

III. Background

A. Flash Memory SSD

Flash memory has several special features unlike the traditional magnetic hard disk. The first one is its "erase-before-write" architecture. To write a data in a block, the block should be first erased. 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 that is a bundle of several sequential pages. 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 need an embedded FTL, which executes on the SSD controller.

To enhance I/O bandwidth, current flash memory SSDs access multiple flash chips with multi-channel and multi-way architecture [2], [6], where the multiple channels can be operated simultaneously and each channel can access multiple flash chips at interleaved manner. Two flash chips using different channels can be operated independently and therefore the page transfer times (from the NAND controller to the flash chip) and page program times for different chips can overlap. For two flash chips sharing a same channel, the data transfer times cannot overlap but the page program times can overlap. 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 sequential I/O performances.

B. RAID technologies

RAID enhances the reliability of storage systems using redundant data. It also improves the performance by interleaving data across multiple disks. Among several levels of RAIDs, RAID-5 makes use of an extra disk to hold redundant information necessary to recover user data when a disk fails. It stripes data across several drives, with parity stored on one of multiple drives.

RAID-5 SSD can be implemented using several flash chips or flash drives as shown in Fig. 1. Depending on striping granularity, each stripe is composed of N+1 logically sequential pages or blocks, where N means the number of disks for user data. The five pages in a stripe can be programmed simultaneously under the parallel NAND architecture. RAID controller has a write buffer to store data temporarily until it is written in the flash chips. It also generates the parity of the stripe to be written and distributes user data and the parity across multiple flash chips. NAND controller writes the data or parity in flash chips. Parities are written at different chips for different stripes. When D_1 and D_6 are updated in Fig. 1, flash chips 1, 2, 3 and 4 may become busy in parallel or in an overlapped fashion.

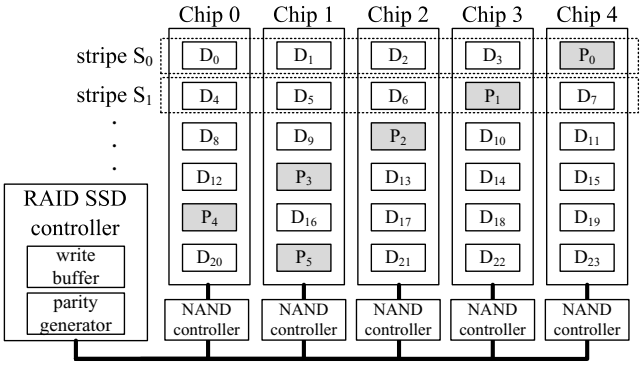

Fig. 1. 4+1 RAID-5 SSD Architecture

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 $(D_{N\cdot j}, D_{N\cdot j+1}, \cdots, D_{N\cdot(j+1)-1}, P_j)$, 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 $\oplus$ means the exclusive ORing (XORing). The chip number of each data is determined by the parity allocation structure in Fig. 1.

In order to change user data D_0 to D_0', RAID-5 needs three steps as follows: 1) read D_0 and P_0; 2) compute a new parity P_0' ($P_0' = D_0 \oplus D_0' \oplus P_0$); 3) write D_0' and P_0'. 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 flash memory, respectively. That is, the total write cost can be increased by over 50% compared with the case of having no redundancy. Therefore, RAID-5 results in poor write performance in flash storage because even a small random write may incur parity update.

IV. Delayed Parity Update Architecture

When there is an update request, FTL generally does not erase or update old data, instead invalidates it 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 to reduce the parity handling cost.

We apply the page-level striping since it shows better performance than the block-level striping. When there is a write request from the host, SSD RAID 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 the parity data to the parity flash chip of the stripe. However, the proposed scheme delays the parity data update and stores it at a special device called *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 with the delayed parity scheme in [7] which stores full parities in parity cache and thus invokes many read operations to calculate the full parities.

By using the partial parity, we can reduce the parity generation overhead. Instead, we maintain the information on the old version of updated data which is an 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 data. The delayed parity is written to flash chip when there is no free space in PPC. This step is called *parity commit*.

A. Partial Parity Cache

Partial parity cache is a storage to hold the delayed parity temporarily. It has the information on parities which are not yet written at flash chips. In order to avoid losing data stored in PPC at sudden power failures, it must be implemented with an NVRAM. Fig. 2 shows the entry of PPC, 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 the 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 flash chip is called *uncommitted stripe*. We denote the set of associated data of the parity P_j as $\pi(P_j)$.

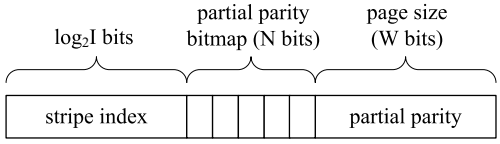

Fig. 2. The Entry of Partial Parity Cache

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

B. Partial Parity Creation & Updating

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

1) If there is no corresponding partial parity of the target logical stripe in PPC (i.e., $S_j \notin$ PPC), a new partial parity $\tilde{P}_j$ should be inserted. There is no flash I/O overhead ($C_{overhead} = 0$).
2) If there is the corresponding partial parity of the target logical stripe in 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 I/O overhead ($C_{overhead} = 0$).
3) Otherwise (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 read cost is invoked ($C_{overhead} = T_{read}$).

For example, Fig. 3 shows the change of PPC when the host sends the update requests on data D_1 and D_2. The RAID SSD is composed of five flash chips and one of which is used for parity. We assume that each block has four pages. Before the update requests, D_1 and D_2 were written at physical page number (PPN) 40 of chip 1 and PPN 80 of chip 2, respectively. Their parity data P_0 has been written at PPN 160 of chip 4. Initially, PPC has no entry.

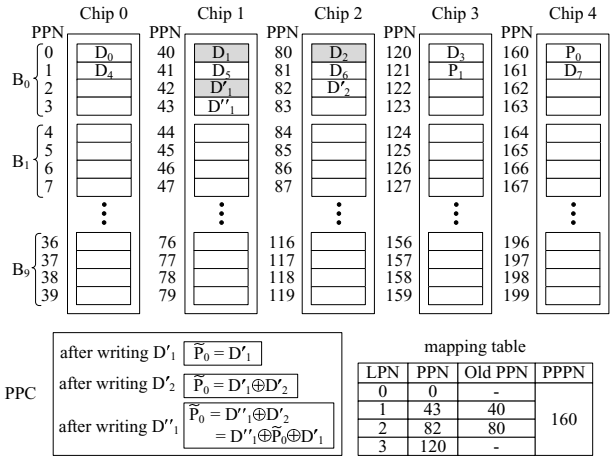

Fig. 3. Partial Parity Creation & Updating

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 data P_0 is not updated immediately in the delayed parity update scheme, the parity remains unchanged. Instead, a partial parity $\tilde{P}_0$ for D'_1 is created in PPC since there is no partial parity of the logical stripe

$S_0 = (D_0, D_1, D_2, D_3, P_0)$ in PPC (case 1). $\tilde{P}_0$ 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 needs no read operation to generate the parity data.

The mapping table for translation between logical page number (LPN) and PPN is also shown in Fig. 3. The table manages a PPN, an old PPN and a physical parity page number (PPPN) for each LPN. PPPN is the physical page number where the parity data of a stripe is written. The old PPN points to the physical page which has the old version of the logical page, where the old version is associated to the parity data written at PPPN. Therefore, the old PPN value of LPN 1 has the value of 40 after the update on D_1.

Generally, flash storage does not maintain such an old PPN of a logical page. Instead, garbage collector maintains the list of all invalid flash pages to reclaim them when there is no sufficient free space. However, our scheme regards the invalid pages as implicit redundant data which can be used to recover a failed data in the stripe since it is associated with out-of-date parities. In this paper, we call such an invalid but useful data a *semi-valid data* since it is valid in terms of failure recovery.

When the host sends the update request on LPN 2 with the new data D'_2, SSD controller writes it at chip 2 and updates the partial parity $\tilde{P}_0$ by XORing its old value and D'_2 since there is the corresponding partial parity $\tilde{P}_0$ in PPC but $D_2 \notin \pi(\tilde{P}_0)$ (case 2). We can know whether a partial parity is associated with a data by examining its partial parity bitmap information in PPC. After the update on 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 has no change since the old parity P_0 is associated to D_1 at PPN 40.

C. Partial Parity Commit

When there is no free space in PPC for a new partial parity after handling several write requests, one of delayed partial parities should be replaced (replacement commit). In addition, before the garbage collection (GC) of flash memory erases the semi-valid pages of uncommitted stripe, the corresponding delayed parity should be committed (GC commit). Since PPC has only partial information, we need the semi-valid data to cope with failure. Therefore, we should commit the partial parity before GC erases the semi-valid data.

To commit the partial parity, RAID controller should first make 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 as follows:

- If $|\pi(\tilde{P}_j)| < \lceil N/2 \rceil$, the full parity is generated by XORing the partial parity, the old full parity and the old data of the associated pages.
- Otherwise, the full parity is generated by XORing the partial parity and the non-updated pages of the stripe.

For instance, in Fig. 4, 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 known from the old PPN field of mapping table.

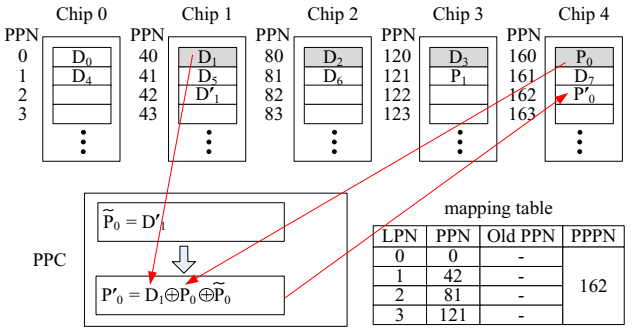

Fig. 4. Partial Parity Commit

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

$$\begin{cases} (|\pi(\tilde{P}_j)|+1)T_{read} + T_{write} & \text{if } |\pi(\tilde{P}_j)| < \lceil N/2 \rceil \\ (N - |\pi(\tilde{P}_j)|)T_{read} + T_{write} & \text{otherwise.} \end{cases}$$

where N represents the number of parallel logical flash chips. $(|\pi(\tilde{P}_j)|+1)T_{read}$ or $(N-|\pi(\tilde{P}_j)|)T_{read}$ means the cost for page reads from flash chips to generate the full parity. The number of flash reads for partial parity commit is limited to $\lceil N/2 \rceil$. T_{write} is the flash write cost of the full parity.

D. Chip Failure Recovery

The SSD controller can fail to read data from flash memory due to page-level error, chip-level error or flash controller-level error. The flash page-level error is generated when it is uncorrectable by 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 with the partial parity $\tilde{P}_j$ and other pages in $\pi(\tilde{P}_j)$.
- Otherwise, D_i can be recovered with the old full parity P_j and the associated pages of $\pi(P_j)$.

For example, if the data D_0 cannot be read due to the failure of chip 0 as shown in Fig. 5, 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 semi-valid pages to recover failed data.

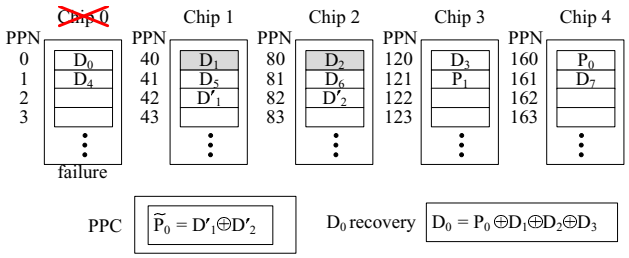

Fig. 5. Failure Recovery

V. Experiments

To evaluate the proposed scheme, we implemented a RAID-5 SSD simulator, which can provide the performance simulation results considering the parallel I/O architecture of SSD. We used two real disk I/O traces, pcNTFS and Financial, and two benchmark traces generated by Iozone and Postmark. pcNTFS trace was collected in a desktop system by executing several applications such as word processor, moving picture, web browsing, games, and so on. Financial is an OLTP application traces used in [9]. While pcNTFS and Iozone traces have many sequential write requests, random requests are dominant in Financial and Postmark traces.

We compared the I/O performances of the three kinds of RAID-5 SSD schemes, `no-PC`, `FPC`, and `PPC`. The `no-PC` scheme does not use the parity cache and thus it is same to the native RAID-5 scheme. The `FPC` scheme uses the parity cache that keeps the delayed full parities as proposed in [7]. The `PPC` scheme is our proposed one which exploits the partial parity cache.

Fig. 6 shows the I/O operations per second (IOPS) values of several schemes normalized by those of RAID-0 scheme. Since RAID-5 schemes should manage redundant data, they provide worse results than RAID-0. The average values of normalized IOPS are 0.53, 0.56 and 0.73 for `no-PC`, `FPC` and `PPC` schemes, respectively. `PPC` improves the performance by 38% and 30% on average compared to `no-PC` and `FPC`, respectively.

Fig. 7 shows the average write handling overhead which is required to handle parity for each write request. `no-PC` scheme theoretically should generate two read operations and one write operation additionally to generate a parity for each page write request. However, it generates smaller numbers of additional read operations (1.5∼1.8) thanks to the write buffer which evicts several victim pages belonging to a stripe in a group at one replacement. Since `FPC` scheme maintains full parities in its parity cache, the number of read operations is same to that of `no-PC` scheme. However, the number of write operations is reduced due to the delayed parity write scheme of `FPC`. The average numbers

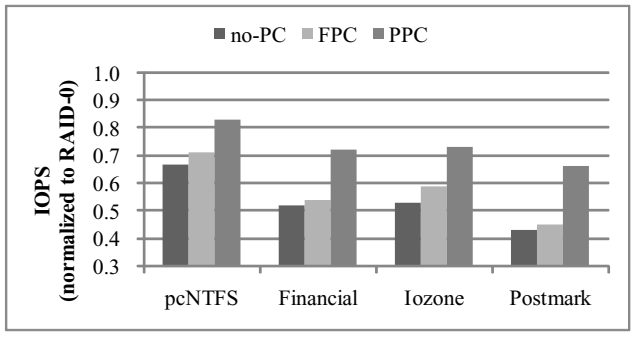

Fig. 6. I/O performances of `no-PC`, `FPC` and `PPC` schemes normalized by RAID-0 scheme (4+1 RAID-5)

of additional reads and writes for each page write in `PPC` scheme are 0.6 and 0.26, respectively. `PPC` scheme reduces the number of read operations significantly for Financial and Postmark traces. This is because these workloads have highly random access patterns. That is, `PPC` scheme is more profitable for random access workloads.

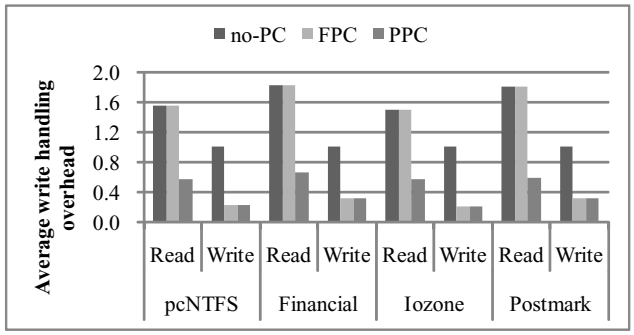

Fig. 7. Average write handling overheads (4+1 RAID-5)

VI. Conclusion

To build high-performance, reliable and large-scale storage systems, RAID technologies are popular. We proposed efficient RAID techniques for reliable flash memory SSDs. In order to reduce I/O overhead for parity handling in 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 write operations that require high costs in flash memory. The partial parity caching technique exploits the implicit redundant data of flash memory to reduce the number of read operations required to calculate parity. The proposed scheme significantly improves the I/O performance of flash memory SSDs especially for random workloads.

We have a plan to study a flash-aware parity handling scheme for other RAID architectures such as RAID-6.

References

[1] D. Reinsel and J. Janukowicz. Datacenter SSDs : Solid footing for growth. http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf, 2008.

[2] 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). In *Proc. of IEEE Non-Volatile Semiconductor Memory Workshop*, pages 17–20, 2006.

[3] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[4] 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. *Journal of Systems Architecture*, 53(9):644–658, 2007.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX'08*, pages 57–70, 2008.

[6] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. of ISCA'09*, pages 279–289, 2009.

[7] K. Greenan, D. Long, E. L. Miller, T. Schwarz, and A. Wildani. Building flexible, fault-tolerant flash-based storage systems. In *Proc. of HotDep'09*, 2009.

[8] Y. Lee, S. Jung, and Y. H. Song. FRA: a flash-aware redundancy array of flash storage devices. In *Proc. of CODES+ISSS'09*, pages 163–172, 2009.

[9] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Proc. of ISCA'08*, pages 327–338, 2008.