DS-RAID: Efficient parity update scheme for SSDs

Jaeho Kim*[1], Jongmin Lee[1], Jongmoo Choi[2], Donghee Lee[1], and Sam H. Noh[3]

[1] *University of Seoul, {kjhnet10, jmlee, dhl_ express}@uos.ac.kr*
[2] *Dankook University, choijm@dankook.ac.kr*
[3] *Hongik University, samhnoh@hongik.ac.kr*

One serious problem of current SSDs is it's low reliability originating from it's primary component, flash-memory, which has high error rates and limited erase count. In particular, the reliability issue is becoming more important as SSDs employing multi-level cell (MLC) flash memory become more prevalent. A solution in enhancing reliability is to provide RAID-5 configuration with the chips comprising the SSD device. The focus of this research is in proposing a mechanism to provide RAID-5 reliability for SSDs that is efficient in terms of performance and lifetime.

Let us start by reviewing how RAID is supported in current SSDs with an example shown in Fig. 1. Fig. 1(a) shows the internal structure of an SSD and how the data would be dispersed among the chips within the SSD. In a typical SSD, there are N_C flash memory chips, and in each chip there are multiple blocks. In each block, there are pages, each associated with a physical page number (PPN). A stripe consists of N_C pages, each of which belong to each individual chip. Typical SSDs today employ a large N_C value typically 10∼16 chips, and more in some SSDs, though in Fig 1 we use an example with only 5 chips. Dx and Px denote user data and parity of the x-th stripe, respectively. The management that we describe below is done at the SSD & RAID controller component in Fig. 1(a).

Take the initial situation in Fig. 1(a) where D0∼D7 are user data and there is a parity per stripe. Stripe 0 consists of pages D0∼D3 and P0, while Stripe 1 consists of pages D4∼D7 and P1. The Stripe map table in the controller holds information regarding each stripe where the number pairs represent the chip number and the PPN within the chip.

Assume data pages D1 through D4 are updated. Note that unlike disk-based RAID-5 where each old strip would be overwritten, this cannot happen with flash memory based SSDs. Instead of overwriting, new data must be written to a new location, and to employ every chip as a RAID component, the new data must be written on the same chip of the original data. Then using either read-modify-write (as for Stripe 1) or reconstruct-write (as for Stripe 0), existing data must be read to calculate the new parity, so that the new data and parity can be written to the chips. The Stripe map table is then updated to reflect the changes as shown in Fig. 1(b).

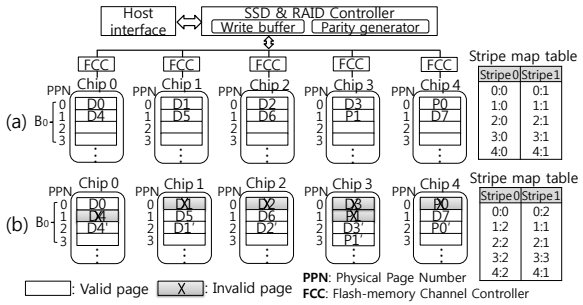

Figure 1: Write sequence of RAID-5 Architecture

There are limitations to this approach. First, whether read-modify-write or reconstruct-write is employed, reading of existing data must precede new parity calculations. This is also true will traditional disk-based RAID-5 systems. Second, when new data is written, the data cannot be written until the stripe becomes full, leaving open a window of vulnerability. For example, if new data D8 and D9 arrive, in our example above, a parity page cannot be calculated for these pages alone, and thus these pages cannot be written until another two new pages arrive. Third, once a data and parity page is allocated to a particular chip, this relation is fixed. Hence, if a particular page is written with higher frequency, then that particular chip will be written to more frequently. Also, the chip in which the parity page resides is more prone to wear out as it must be written to more frequently. These fixed relations eventually lead to higher cleaning costs and decreased lifetime of the SSD.

*Student, Presenter

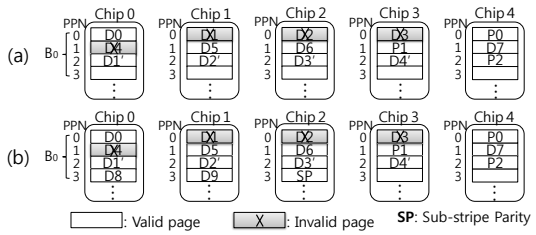

Figure 2: Write sequence of DS-RAID Architecture

Methods such as PPC [1] and Lifespan-aware scheme [2] have been proposed to resolve these limitations. However, these solutions require costly non-volatile RAM and still suffer from the drawbacks of frequent small writes.

Dynamic Striping-RAID

We propose *Dynamic Striping-RAID (DS-RAID)* that fundamentally solves the problems mentioned above. DS-RAID has the following two features. First, every strip that comprises a stripe always has the same PPN number. Hence, there is no need for a Stripe map table. Second, DS-RAID employs the sub-stripe parity scheme, which calculates and writes a sub-stripe parity for data write request smaller than the stripe size (minus 1 for parity).

Let us go through an example starting from the same Fig. 1(a) with valid user data, D0-D7. With DS-RAID, all pages of PPN value 0 and 1 comprise Stripe 0 and 1, respectively. As D1∼D4 are modified, the controller simply calculates the new parity for these pages, writes them on the pages with PPN 2 along with the parity value as shown in Fig. 2(a). After writing, the controller simply marks the old pages as obsolete. There is no need to read the old pages. Furthermore, all chips are written to evenly even if particular pages are more frequently written to, including the parity page.

Now consider the case when only part of the stripe is written. Assume again that D8 and D9 are being newly written. With DS-RAID, we do not wait for more data to arrive but calculate the parity of these data (sub-stripe parity, SP) and allocate them right away to PPN 3 of chips 0 and 1, gaining instant RAID-5 reliability as shown in Fig. 2(b). As more data arrive, say D10, then a full parity is calculated with D8 and D9 that are still in the controller cache. Finally, D10 and the full parity are stored accordingly. This method does waste some space for storing sub-stripe parity, but these will be reclaimed during garbage collection.

Evaluation Platform and Results

We implement the DS-RAID scheme by modifying the Disksim SSD extension [3]. The SSD consists of 8 flash memory chips with 8 planes. Each plane has 2048 blocks and a stripe consists of 8 pages (32KB) that have the same PPN number in each of the flash memory chips.

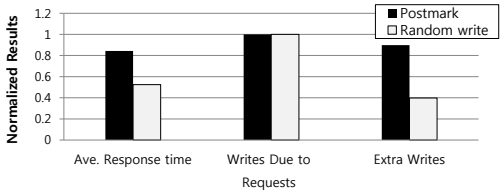

Figure 3: Results normalized to RAID-5

Fig. 3 shows the I/O performance and the write counts of DS-RAID relative to that of RAID-5 for the Postmark and random write benchmarks. The Postmark is a file system benchmark with significant sequential I/O with an average write request size of 1.6MB, while the random write benchmark is a synthetic workload that makes 250K write requests with an average size of 12KB. As the Postmark benchmark requests are large and sequential, the effect of DS-RAID should be minimal, while the random workload should benefit the most. In terms of response time, we see that performance improves roughly 20% for Postmark despite its workload characteristics. Response is reduced to half for the random workload.

In terms of lifetime, we divide the number of writes into two of which the first is the writes that occur because of the requests themselves. Naturally, for both workloads, they should be same for both types of RAID. The other writes, which include parity writes, writes incurred by garbage collection, and, in the case of DS-RAID, the sub-parity writes, are the ones that need to be controlled to the minimum to lengthen the lifetime of an SSD. The "Extra Writes" column, which represents these writes, shows that writes for Postmark are reduced by roughly 10%, while there is a 60% reduction for random writes.

References

[1] S. Im and D. Shin. Flash-aware raid techniques for dependable and high-performance flash memory ssd. *Computers, IEEE Transactions on*, 60(1):80 –92, jan. 2011.

[2] S. Lee, B. Lee, K. Koh, and H. Bahn. A lifespan-aware reliability scheme for raid-based flash storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 374–379.

[3] V. Prabhakaran and T. Wobber. SSD Extension for DiskSim Simulation Environment.