Improving Lifetime of SSD-based RAIDs using Dedup-assisted Partial Stripe Writes

Taejin Kim, Jisung Park, and Jihong Kim

Department of Computer Science and Engineering
Seoul National University, Korea
{*taejin1999, jspark, jihong*}*@davinci.snu.ac.kr*

ABSTRACT

For SSD-based storage systems, the Differential RAID technique has been proposed to reduce the probability of correlated multiple failures among SSDs caused by the even parity distribution of RAID-5. In order to differentiate the age among SSDs, the full stripe writes should be replaced to the partial stripe writes by allocating more writes to a single SSD. During the full stripe write replacement, the lifetime of SSDs is inevitably reduced due to the increased number of parity updates. In this paper, we propose a deduplication-assisted partial stripe write replacement technique to improve the lifetime of SSD-based RAIDs. With the deduplication, a full stripe write can be changed to a partial stripe write so that the age of SSDs can be differentiated without additional parity updates. Our experimental results show that the proposed technique can effectively create the age difference among SSDs similar to the original Diff-RAID without additional parity updates. Furthermore, with the elimination of duplicated data, we can extend the lifetime of SSD-based RAIDs by 44% compared to Diff-RAID.

1. INTRODUCTION

As the price-per-byte of NAND flash memory is rapidly decreasing, NAND flash-based solid-state drives (SSDs) are emerging as a viable replacement for hard disk drives (HDDs). However, as NAND flash memory technology scales down to 20-nm and below, storing data reliably in NAND flash memory becomes a more challenging design requirement of NAND-based storage systems [1]. In particular, the increased bit error rate (BER) and reduced maximum program/erase (P/E) cycle of NAND flash memory requires a special care to guarantee the reliability of flash-based SSDs.

In order to achieve a higher level of reliability in SSDs, system-level redundancy techniques such as Redundant Array of Independent Disks (RAID) [2, 3] are popular for storage systems. In RAIDs, which combine multiple disks into a single logical device, data are distributed among multiple disks in a redundant fashion depending on the required redundancy level (often called as the RAID level). Most RAID levels employ an error correction scheme (mostly based on parity bits) that provides a data recovery capability to the RAID for the event of a disk failure. For example, in a RAID-5 configuration, writes are grouped by a chunk (which consists of multiple NAND pages) and the parity is distributed across the SSDs. Figure 1 shows how the data chunks (e.g., `A0`, `B1`, and `C2`) and parity chunks (e.g., `Ap` and `Bp`) are allocated in RAID-5. Since a parity chunk is calculated by `xor`ing all the data chunks in the same stripe (which is composed of multiple chunks at the same offset over disks), it should be updated when at least one data page in the same stripe is updated. There are two methods for writing a stripe in RAID-5 depending on the size of the write request. When the size of a write request fits to the size of a stripe, the parity can be directly calculated by data to be written without reading existing one. Therefore, data and parity of the stripe can be written together, which is called a *full stripe write*. On the other hand, when the size of a write request is smaller than the size of a stripe, existing data and parity in the stripe should be read so that the new parity can be calculated. Since the stripe is partially updated, this is called a *partial stripe write*.

In the RAID-5 array, parity chunks are evenly distributed across SSDs so that extra load from updating parity chunks

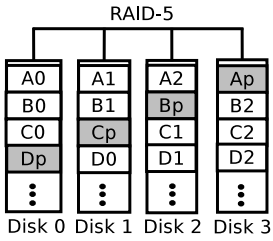


Figure 1: An example RAID-5 configuration with 4 disks.

can be equally shared among SSDs. For instance, if a chunk consists of a single page and four random pages (which are allocated to chunk `A0`, `B1`, `C1`, and `D2`) are updated in Figure 1, all the pages are written by the partial stripe write method so each SSD receives both data and parity updates. As a result, the overhead of the parity updates is equally distributed among four SSDs in RAID-5.

However, as discussed in [4], for a SSD-based RAID-5 array, an even distribution of parity updates significantly reduce the reliability of a SSD-based RAID because all SSDs may be worn out at similar times, thus increasing a probability of correlated multiple failures among SSDs. Diff-RAID [4] attempts to create and maintain the age difference among SSDs to guarantee that at least some SSDs have lower BER to avoid the correlated SSD failure. For this purpose, Diff-RAID allocates more parity chunks to an older SSD to differentiate the aging rate of SSDs.

Although the key insight behind Diff-RAID is novel, the original Diff-RAID may fail to maintain the age difference among SSDs, especially under a sequential write workload. For instance, if three sequential pages (which are allocated to chunk `A0`, `A1`, and `A2`) are requested to be written for a page-sized chunk in Figure 1, the pages are written by the full stripe write so four SSDs (including the parity SSD) should be written regardless of how we re-allocate the parity chunk. Thus, there is a limited chance to differentiate the aging rate of SSDs for Diff-RAID with sequential writes.

The limitation can be overcome by increasing the chunk size so that more writes are written to a single SSD and even distribution of writes is avoided. If we change the chunk size to be three pages in the previous example, the requested three pages are allocated to the chunk `A0` together. Unlike the previous example, the pages are written by partial stripe writes instead of full stripe writes. Since only two SSDs are written, the aging rate of SSDs can be differentiated and can be managed by re-allocating the chunk `Ap` to another SSD. Although using the large chunk size can mitigate the sequential write problem in Diff-RAID, the total amount of writes is increased due to more frequent parity updates. In other words, the lifetime of SSDs should be sacrificed to achieve a higher reliability.

In this paper, we propose a novel lifetime management technique for Diff-RAID using a deduplication technique as a main instrument of converting full stripe writes to partial stripe writes. By removing duplicated pages from a full stripe write using a deduplication technique, we convert most writes into partial stripe writes. Since a partial stripe write allows more flexibility in deciding a destination SSD for each page of the partial stripe write, we can better meet the age difference requirement for each SSD of Diff-RAID without using a large chunk size. For this purpose, we propose a simple but effective SSD re-allocation technique that utilizes the location of the eliminated data for differential age distribution. Our proposed technique achieves a similar reliability level of the original Diff-RAID while significantly improving the lifetime of SSDs.

The rest of the paper is organized as follows. Section 2 describes the write amplification problem of Diff-RAID with evaluation results. Section 3 presents the dedup-assisted partial stripe write technique. In Section 4, experimental results of the proposed technique are presented. Finally, Section 5 concludes with a summary and future work.

2. WRITE AMPLIFICATION PROBLEM IN DIFF-RAID

In order to decrease the probability of correlated multiple failures among SSDs in RAID, Diff-RAID incurs skewed writes across SSDs by distributing parity unevenly. For full stripe writes, however, the uneven parity distribution can not make skewed writes because the full stripe write incurs the same amount of writes to all SSDs in a RAID array. In order to evaluate the reliability degradation problem due to the full stripe writes, We ran several traces on the Diff-RAID array that is implemented based on a Linux RAID module, MD [6] with four SSDs. Figure 2 shows the percentage of number of written pages per SSD for various traces. `Web`, `homes`, `mail` traces are from [5] and `Random` trace is a synthetic small random write workload. Since the `Random` trace incurs only partial stripe writes Diff-RAID effectively differentiate the number of written pages between SSDs. However, the difference of number of written pages is not sufficient for `web`, `homes`, `mail` traces because of the significant sequential writes ratio, which are 87%, 63%, and 94%, respectively.

Thus, increasing the partial stripe write ratio for the sequential workload is essential for Diff-RAID to achieve desirable reliability. In order to make more partial stripe writes, the authors of Diff-RAID suggested that the stripe size should be increased [4]. Since the size of a stripe is determined by `(size of chunk) * (# of data devices)`, we can increase

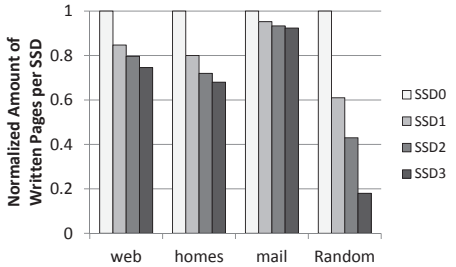


Figure 2: The difference of written pages of Diff-RAID with various traces.

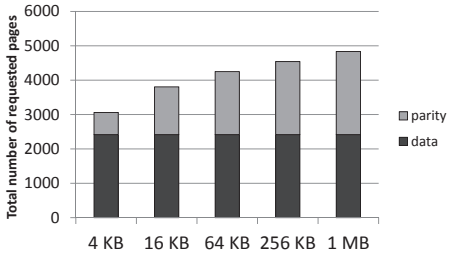


Figure 3: The amount of written data varying chunk sizes.

the stripe size by setting the size of a chunk to be large. However, a partial stripe write incurs more parity updates than a full stripe write so the lifetime of SSDs is reduced. We evaluated how the chunk size affects the amount of parity update by running a workload that issues 1 MB-sized write requests on top of MD. Figure 3 shows the total amount of written pages varying chunk sizes. As shown in Figure 3, the amount of written parity page is significantly increased as the chunk size increases due to the frequent parity updates.

Figure 4 shows the amplified writes of Diff-RAID when a large chunk size is used. Since only a part of SSDs receive writes when the chunk size becomes large, the number of written page difference is similar to the `Random` trace. As mentioned above, however, the number of parity updates increases due to the large chunk size. Particularly, the writes are amplified up to 1.8x for `mail` trace, significantly decreasing the lifetime of RAID. In conclusion, Diff-RAID may fail to satisfy the endurance and the reliability requirements at the same time when the portion of full stripe writes become large.

3. ENDURANCE IMBALANCING TECHNIQUE USING DEDUP-ASSISTED PARTIAL STRIPE WRITES

3.1 Dedup-assisted Partial Stripe Writes

In this section, we describe the lifetime improvement technique for SSD-based RAID using dedup-assisted partial stripe writes. As discussed in Section 2, for the sequential workload, the amount of written data is inevitably increased due to the large chunk size for differentiating the aging rates of SSDs in Diff-RAID. Instead of using a large-sized chunk, we can efficiently replace the full stripe writes to partial

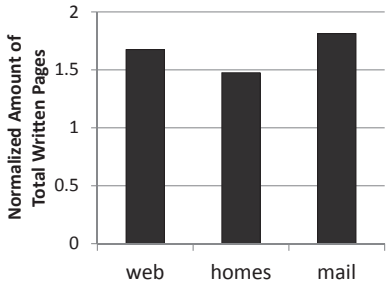


Figure 4: The amount of amplified writes of Diff-RAID.

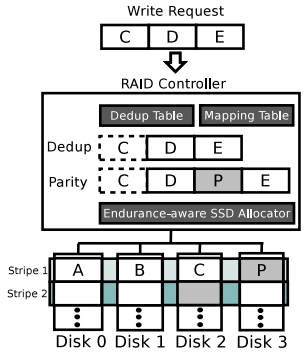


Figure 5: An example of replacing full stripe write by deduplication.

stripe writes by removing duplicated data in the full stripe writes. Figure 5 shows how the deduplication is combined with RAID to increase the ratio of partial stripe writes. In Figure 5, the deduplication stage is added to the RAID controller so that we can find duplicated data across SSDs in the RAID-5 array. When RAID controller receives a write request, it computes fingerprint of each page using a collision-resistant hash function. After fingerprinting, each fingerprint is looked up in the dedup table which maintains the fingerprints of written data to SSD. Each entry of the dedup table is composed of a key-value pair, {*fingerprint*, *location*}, where the location indicates a SSD number and address of written data. If the same fingerprint is found, it is not necessary to write data. Instead, the mapping table is updated so that the corresponding mapping entry points to the location of previously written data. If there is no matched fingerprint in the dedup table, the new fingerprint is inserted into the dedup table with its location.

For example, three sequential pages, whose contents are `C`, `D`, and `E`, are requested to be written and data `A`, `B`, and `C` is already written at the first stripe of RAID. Since data `C` is duplicated in the example, we need to write only two pages, which means a full stripe write is replaced by a partial stripe write. After deduplication, the write request is assigned to the second stripe and the parity is calculated using the non-duplicated data, `D`, and `E` in the example. Before the stripe is written, the endurance-aware SSD allocation step can change the location of eliminated data in order to make sure the difference of written pages across SSDs is maintained. The detailed method for SSD allocation will be explained in the following section.

3.2 Dynamic SSD Allocation

Since the SSD location of duplicated data can not be guaranteed, converting to partial stripe write may not be able to incur the desired difference of written pages across SSDs. In order to satisfy the number of written page difference of Diff-RAID, we propose a dynamic SSD allocation technique for a full stripe write that contains duplicated data. The

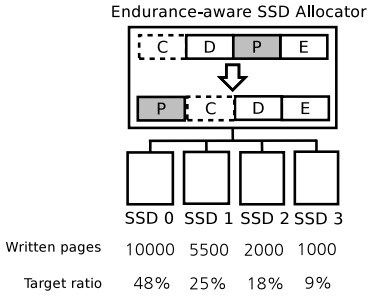
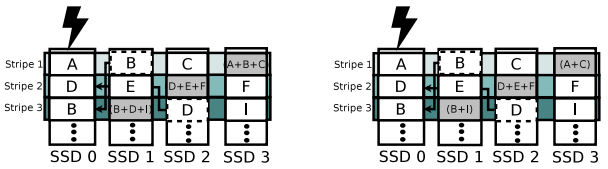


Figure 6: An example of dynamic SSD allocation.

endurance-aware SSD allocator in Figure 5 re-assigns the location of data before the stripe is written to the RAID array. In order to create the age difference among SSDs same as Diff-RAID, the number of issued writes per SSD and the target age distribution are maintained. The target ratio is obtained from Diff-RAID with (82, 6, 6, 6) allocation. Then, the allocator re-assigns the SSD location to meet the target ratio by the following policy. First, since the parity should be updated whenever data in the same stripe is updated, the parity is allocated to the SSD which has the most target ratio. Second, when the duplicated data is eliminated, it is located to the SSD which has the least target ratio so that the SSD would receive less writes than other SSDs. Third, if an SSD receives more writes than the target ratio, the SSD is not allocated. Instead, we change the SSD location to an SSD whose written page ratio is below the target if available. When there are multiple SSDs that exceed the target ratio, the SSD with larger target ratio is selected to be written because it is more reliable to have more young SSDs when an SSD has failed.

Figure 6 shows an example of the dynamic SSD allocation. A full stripe write (data `C`, D, and E) is requested to be written and `SSD 2` is the original location of parity. Since the full stripe write is deduplicated, we can apply the dynamic SSD allocation policy. As mentioned above, the parity is reallocated to `SSD 0` so we can make an SSD with the most target ratio get more writes. Moreover, `SSD 1` received more writes than the target ratio so we allocate the eliminated data `C` to `SSD 1`. Since Diff-RAID utilizes only the uneven parity distribution, the writes are distributed indirectly. The proposed technqiue, however, dynamically changes SSD location so that the age difference among SSDs is created more effectively than Diff-RAID.

Furthermore, the proposed SSD allocation technique is applicable only to the full stripe writes because the partial stripe writes require different mechanism for changing the allocation. The different SSD allocation mechanism between the full stripe writes and the partial stripe writes comes from the different parity calculation. For the full stripe writes, since the old parity and old data is not needed to calculate new parity, we can freely overwrite the old data and



(a) Dedup after parity calculation example

(b) Dedup before parity calculation example

Figure 7: Examples of data recovery when an SSD 0 has failed.

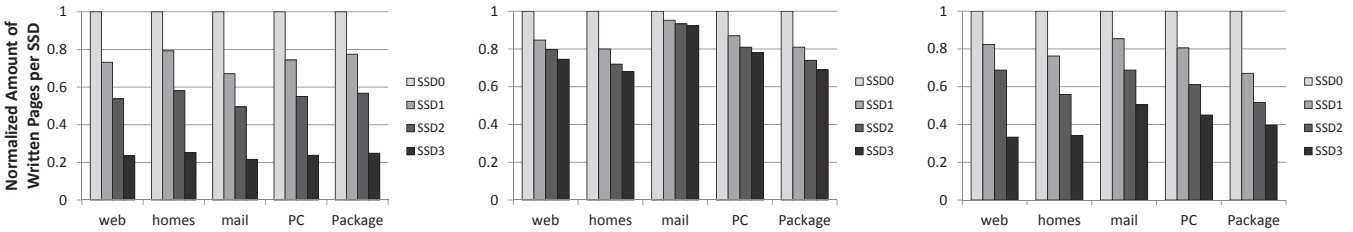
parity. Unlike the full stripe writes, old data and old parity are necessary for the parity calculation for the partial stripe writes. If we re-allocate `SSD 0` to `SSD 1`, the old data at `SSD 1` should be moved to `SSD 0` to avoid the data loss by overwriting.

3.3 Excluding Duplicated Data for Parity

For applying deduplication on RAID, since the original data could be placed in other SSD, data recovery process can be complicated. In this section, we describe how the recovery process can be simplified in the proposed method. As explained in Section 3.1, the deduplication is applied before the parity calculation so that the parity does not contain eliminated data. The exclusion of duplicated data for parity enables the simple data recovery process. For a deduplicated stripe, if we apply the deduplication after the parity calculation, we need to recover the original data first. If This procedure is repeated for multiple times, called chained recovery, the recovery time would be significantly increased.

Figure 7 shows the data recovery process when `SSD 0` has failed for the cases that the deduplication is applied after the parity calculation and the reverse order. In both example, data `B` at `Stripe 1` and `D` at `Stripe 3` is deduplicated and the location where the original data were written is linked with an arrow in the figure. In Figure 7(a), the parity includes entire data in the same stripe regardless of whether the data is deduplicated because the deduplication is applied after the parity calculation. In order to recover `Stripe 1`, we need data `B`, `C` and the parity. However, since data `B` is deduplicated, we need to get original data `B`. Unfortunately, the original `B` was written at `SSD 0` so we need to recover `Stripe 3` first. A similar process is required for recovering the `Stripe 3` because data `D` is also deduplicated and the original data `D` is not able to obtain. Finally, after data `D` at `Stripe 2` is recovered, `Stripe 3` and `Stripe 1` are also recovered. As a result, more than one stripes are additionally required to be restored for recovering the deduplicated stripe.

Unlike the previous example, we do not need to recover other stripes if the deduplication is applied before the parity calculation. As shown in Figure 7(b), the parity does not include the deduplicated data. For recovering data `A`, we only



(a) Diff-RAID with a large chunk. (b) Diff-RAID with a small chunk. (c) The proposed scheme.

Figure 8: The difference of written pages among SSDs for various schemes.

need data `C` and the parity since data `B` was not included to the parity. Thus, we apply the deduplication before the parity calculation in this paper.

4. EXPERIMENTAL RESULTS

4.1 Experimental Settings

In order to evaluate the effectiveness of our proposed techniques, experiments are performed on the RAID-5 device based on the MD with FlashBench [7] as an SSD emulator. Diff-RAID and the proposed technique is implemented on MD. A trace-based experimental environment is set up to set inputs to MD by building a Linux kernel module that issues an I/O request based on a trace file to MD. The module creates a block I/O request by reading the I/O information in the block I/O trace file such as logical block address and size of the request. During the experiments, we collected the number of written pages of each SSD to see the amount of the amplified writes of Diff-RAID. We then compared them with that of the proposed technique to see the lifetime improvement.

We used five different I/O traces for the evaluations. Three production system traces, `web`, `homes` and `mail` were from the FIU [5]. Two in-house traces, `PC` and `Package` traces were collected while running real-world applications. `PC` represents a desktop PC workload such as an web surfing, emailing, and document editing whereas `Package` includes an updating and downloading software packages. Table 1 summarizes the characteristics of the I/O traces such as amount of writes, average sequential write request size, the ratio of sequential write requests, and the ratio of duplicated data.

4.2 Inter-SSD Lifetime Fluctuation Evaluation

Traces	Amount of Writes (GB)	Avg. Seq. Write Req. Size (KB)	% of Seq. Write Req. (%)	Dedup Ratio (%)
`web`	37.28	37.48	87	28
`homes`	65.27	19.96	63	39
`mail`	1483.4	75.16	94	31
`PC`	3.1	31.19	77	29
`Package`	4.9	40.44	69	20

Table 1: Write characteristics of traces used in our experiments.

As mentioned in Section 3, the proposed technique can differentiate the amount of written pages among SSDs without incurring additional parity updates. Figure 8 shows how the difference of written pages among SSDs is similar to the target ratio of Diff-RAID for the each case. As shown in Figure 8(a), Diff-RAID with a large chunk shows a very similar age distribution to the target distribution. When we use a large-sized chunk, the workload seems to be random because it is hard to have a full stripe write. On the other hands, Diff-RAID with a small chunk size fails to make a desirable age difference as shown in Figure 8(b). Especially for the mail trace, since the ratio of sequential write is significant, most of the writes are written in full stripe writes. However, Figure 8(c) shows that the proposed technique achieves a similar result with Diff-RAID with a large chunk size. By increasing the number of partial stripe writes and allocating the partial stripe writes in a dynamic fashion among SSDs, the proposed technique can effectively differentiates the amount of written pages among SSD. If we compare the `homes` trace and the `mail` trace, when there are more duplicated data, the age difference becomes more similar to the target ratio.

4.3 Endurance Evaluation

The proposed technique does not incur the additional parity updates due to the large chunk size for differentiating the amount of written pages among SSDs. Moreover, the total number of writes is considerably reduced by eliminating duplicated data. Figure 9 shows the total amount of written pages for the various traces. The results shown in Fig-

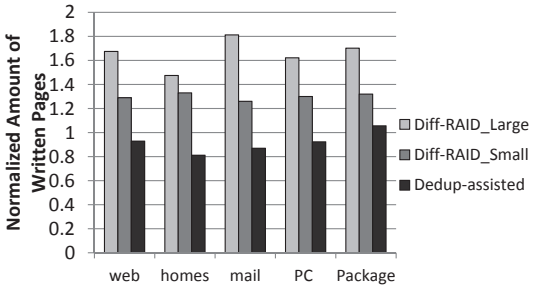


Figure 9: The amount of written pages under various schemes.

ure 9 are normalized to the number of requested pages of the traces. Since RAID-5 incurs redundant writes, i.e. parity, the amount of written pages is larger than 1 in most cases. As shown in Figure 9, Diff-RAID with large chunk size significantly increases the amount of written pages due to the large number of parity updates. For `mail` trace, we can see the largest increase of the written pages because the significant portion of sequential write results the largest size of chunk. On the other hand, the amount of amplified writes for Diff-RAID with small chunk size is almost minimal because most of the sequential writes are evenly distributed across SSDs. By exploiting the deduplication, the proposed technique shows significant reduction in the amount of written pages. Despite of the redundancy, the written pages are smaller than the amount of writes of the traces, except `Package` trace. The amount of reduction depends on the dedup ratio of the traces. As a result, the proposed technique reduces the amount of written pages by 44% on average and up to 52% for `mail` trace over Diff-RAID, extending the lifetime of SSD-based RAID by the same amount.

5. CONCLUSION

In this paper, we proposed a lifetime improvement technique for SSD-based RAIDs using dedup-assisted partial stripe writes. We convert full stripe writes to partial stripe writes which provide more flexibility in meeting the inter-SSD lifetime fluctuations required by Diff-RAID while avoiding the write amplification problem of the existing approach. Our evaluation results show that the proposed technique improves the lifetime of a SSD-based Diff-RAID up to 52% over the original Diff-RAID.

Our proposed technique can be extended in several directions. For example, we plan to integrate other lifetime enhancement techniques (such as dynamic program erase scaling [8]) in differentiating the lifetime of SSDs so that a better reliability of Diff-RAID can be achieved.

6. ACKNOWLEDGMENTS

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (MSIP) (No. 2010-0020724). This research was also supported by NRF grant funded by MSIP (NRF-2013R1A2A2A01068260). The ICT at Seoul National University and IDEC provided research facilities for this study.

7. REFERENCES

[1] L. Grupp, J. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *Proc. 10th USENIX Conference on File and Storage Technologies*, 2012.

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

[3] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *ACM SIGMOD Record*, vol. 17, no. 3, pp. 109-116, 1988.

[4] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD Reliability," in *ACM Transactions on Storage*, vol. 6, no. 2, pp. 1-22, 2010.

[5] R. Koller, and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," in *Proc. 8th USENIX Conference on File and Storage Technologies*, 2010.

[6] N. Brown, "mdadm," `http://en.wikipedia.org/wiki/Mdadm`, 2001.

[7] S. Lee, J. Park, and J. Kim, "FlashBench: A Workbench for Rapid Development of Flash-Based Storage Devices," in *Proc. 23rd IEEE International Sympoisum on Rapid System Prototyping*, 2012.

[8] J. Jeong, S. Hahn, S. Lee, and J. Kim, "Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling," in *Proc. 12th USENIX Conference on File and Storage Technologies*, 2014.