Efficient Lifetime Management of SSD-based RAIDs Using Dedup-Assisted Partial Stripe Writes

Taejin Kim, Sungjin Lee*, Jisung Park, and Jihong Kim
Department of Computer Science and Engineering, Seoul National University
{taejin1999, jspark, jihong}@davinci.snu.ac.kr
Department of Computer Science and Information Engineering, Inha Univeristy
sungjin.lee@inha.ac.kr

***Abstract*—For SSD-based RAID systems, the Diff-RAID technique has been proposed to reduce the probability of correlated multiple failures among SSDs by differentiating the amount of written data to each SSD. Although Diff-RAID works well for workloads with many small random writes (which require frequent parity updates), it does not perform well with recent data center workloads (e.g., key-value stores) which are dominated by large sequential writes (i.e., full stripe writes). In order to efficiently differentiate the amount of written data to each SSD for data center workloads, full stripe writes should be converted to partial stripe writes. However, a naive solution such as using a large chunk size significantly degrades the lifetime of SSDs because of a large increase in parity updates. In this paper, we propose a new lifetime management technique, DA-RAID, which overcomes the limitation of Diff-RAID by using deduplication-assisted partial stripe writes. In DA-RAID, a full stripe write can be converted to a partial stripe write so that the age differentiation among SSDs can be supported without increasing the amount of parity updates. Our experimental results show that DA-RAID can effectively sustain the age difference among SSDs even for data center workloads (for which Diff-RAID fails) without additional parity updates. DA-RAID achieves similar age differences among SSDs as Diff-RAID with a large chunk size while reducing the amount of parity updates by 32% over Diff-RAID.**

I. Introduction

A redundant array of independent disks (RAID) [1] is a popular solution for enterprise storage systems because of its unique advantages, including fault tolerance, high performance and high capacity. In a RAID system, which is composed of multiple disks where user data are distributed, an error correction scheme is employed to provide a data recovery capability. A typical parity bit-based recovery scheme keeps additional parity chunks for multiple user-data chunks, called stripes, which are a group of user data kept in the same logical block offset over disks. Using parity chunks, original data can be recovered in the event of a disk failure. For example, if a single parity chunk is kept for a stripe, a RAID system can sustain up to one disk failure.

Recently, RAID is widely used for flash-based solid-state disks (SSDs). Using the conventional RAID architecture with SSDs, however, may significantly decrease data reliability because several disk failures can be highly correlated in SSD-based RAID systems. Because of its mechanical nature, failures of hard disk drives (HDDs) usually occur in a random manner; that is, a probability of more than two HDDs being malfunctioning at similar times is very low. On the other hand, the reliability of an SSD is largely decided by the amount of data written to it. Since a RAID system tends to write all incoming data to different disks equally, SSDs in the same RAID group are likely to be worn out at similar times, which greatly increases a possibility of multiple simultaneous disk failures. Once several SSDs die simultaneously, it is difficult to recover original data, regardless of the existence of parity chunks.

There were a lot of studies that attempt to resolve the problem with correlated failures in SSDs [2], [3], [4], but many of them stem from a common idea of Diff-RAID [5]. Diff-RAID intentionally creates and maintains the age differences among SSDs so that some SSDs are worn out earlier than others. Diff-RAID is based on a key observation that a majority of data is written by partial stripe writes whose lengths are shorter than that of a stripe. When a partial stripe write is requested, only a small part of a destined stripe needs to be updated. A RAID controller, however, has to calculate a new parity chunk with the up-to-date stripe and write the new parity to disks again for future data recovery. For workloads with many partial stripe writes, parity chunk writes account for a non-trivial portion of total writes. Diff-RAID exploits such a characteristic - by unevenly distributing parity chunks to certain SSDs, it can automatically create the age differences among SSDs, resolving the correlated disk failures problem without any additional I/O costs.

Although the key insight behind Diff-RAID is novel, the main weakness of Diff-RAID is that it may not work well for all workloads. In particular, Diff-RAID does not work properly when large sequential writes are dominant. For such workloads, full stripe writes are frequently observed. Compared with partial stripe writes, full stripe writes incur a smaller number of parity chunk updates, so Diff-RAID fails to maintain the age differences among SSDs. Considering workload characteristics of recent storage systems, in particular data center workloads, this weakness can be a serious obstacle for Diff-RAID to be widely deployed in real systems. For example, recent NoSQL-based database systems, such as Cassandra [6], MongoDB [7], and RocksDB [8], mostly issue sequential writes to storage systems because their data management engines are based on the LSM Tree [9]. For example, when we analyzed write requests of Cassandra about 94% of data were written by full stripe writes, thus severely limiting a possibility of sustaining the required age differences

among SSDs. Write workloads of log-structured and copy-on-write file systems (e.g., F2FS [10], BtrFS [11], and HDFS [12]) can cause similar problems as well for Diff-RAID because these file systems tend to append new data by sequentially appending to storage devices using full stripe writes.

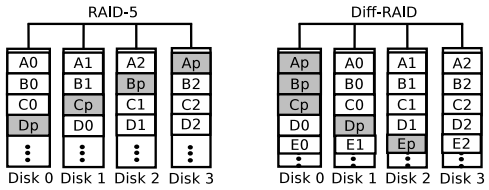
In order to efficiently differentiate the amount of written data to each SSD, full stripe writes should be converted to partial stripe writes. However, a naive solution such as using a large chunk size significantly degrades the lifetime of SSDs because of a large increase in parity updates (as explained in Section II). In this paper, we propose a new lifetime management technique, DA-RAID, for RAID systems which overcomes the technical limitation of Diff-RAID without degrading the SSD lifetime. In DA-RAID, deduplication is employed 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 can convert many full stripe writes into partial stripe writes. Since the converted partial stripe writes enable more flexibility in deciding a destination SSD for each page of partial stripe writes we can better meet the age difference requirement for among SSDs. In order to sustain the required age differences among SSDs, we also propose a simple but effective SSD re-allocation technique that adaptively changes destination SSDs for partial stripe writes by accounting for each SSD's aging pattern. Our experimental results show that DA-RAID can effectively sustain the age difference among SSDs even for data center workloads (for which Diff-RAID fails) without additional parity updates. DA-RAID achieves similar age differences among SSDs as Diff-RAID with a large chunk size while reducing the amount of parity updates by 32% over Diff-RAID.

The rest of the paper is organized as follows. Section II describes the limitations of existing RAID and Diff-RAID techniques. Section III presents the proposed DA-RAID in detail and shows how it solves problems with the existing RAID solutions. In Section IV, experimental results of DA-RAID are presented. Finally, Section V concludes with a summary and future work.

II. Motivation

In this section, we briefly explain RAID systems and introduce problems with Diff-RAID. Figure 1 shows an example of a RAID-5 and Diff-RAID system with four disks. In RAID-5, three of them are used to store user data (i.e., data chunks) while the other one is for keeping parity data. According to the specification of RAID-5, the parity is distributed across the four disks [1]. For example, in Figure 1(a), A0, B1, and C2 are data chunks and Ap and Bp are parity chunks. Since a parity chunk is calculated by xoring all the data chunks in the same stripe, 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 (i.e., a full stripe write), 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. On the other hand, when the size of a write request is smaller than the size of a stripe (i.e., a partial stripe write), existing data and parity in the stripe should be read so that the new parity can be calculated.



(a) RAID-5 (b) Diff-RAID

Fig. 1: An example RAID-5 and Diff-RAID configuration with 4 disks.

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

As pointed out in Section I briefly, this even distribution of parity chunks in RAID-5 could cause the correlated multiple failure problem because it leads that all SSDs are worn out at similar times. To avoid this, Diff-RAID allocates more parity chunks to an older SSD to differentiate the aging rate of SSDs. Figure 1(b) shows how Diff-RAID allocate parity chunk among SSDs. For example, when three different pages, A0, B1, and C2, are randomly written to the disks, Disk 0 gets three writes while the others gets single write.

Unfortunately, Diff-RAID cannot work when three pages, A0, A1, and A2, are written sequentially. 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(b), 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.

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 [13] with four SSDs. Figure 2 shows the percentage of number of written pages per SSD for various traces. `Web`, `homes`, `mail` traces are from [14] 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

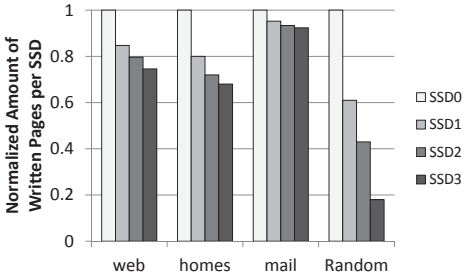


Fig. 2: The difference of written pages of Diff-RAID with various traces.

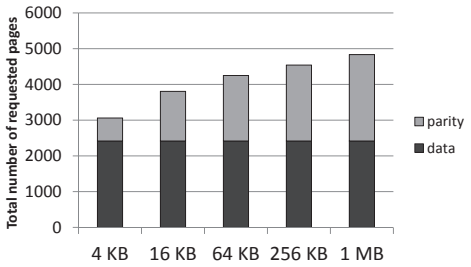


Fig. 3: The amount of written data varying chunk sizes.

not sufficient for `web`, `homes`, `mail` traces because of the significant sequential writes ratio, which are 87%, 63%, and 94%, respectively.

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.

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 (512 KB) 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.

III. Endurance Imbalancing Technique using Dedup-assisted Partial Stripe Writes

A. Dedup-assisted Partial Stripe Writes

In this section, we describe the lifetime improvement technique for SSD-based RAID using dedup-assisted partial stripe

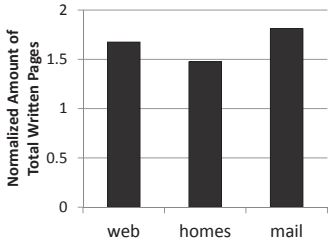


Fig. 4: The amount of amplified writes of Diff-RAID.

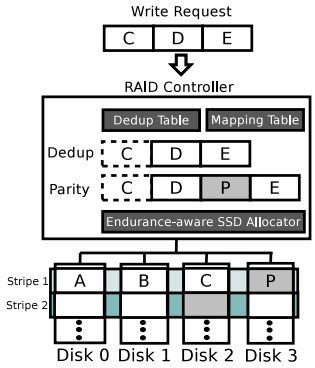


Fig. 5: An example of replacing full stripe write by deduplication.

writes. As discussed in Section II, 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 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. The fingerprint computation can be supported by the hash instruction (e.g., Intel SHA Extensions) to decrease the overhead of 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 mean 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.

B. Dynamic SSD Allocation

Since the SSD location of duplicated data cannot 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

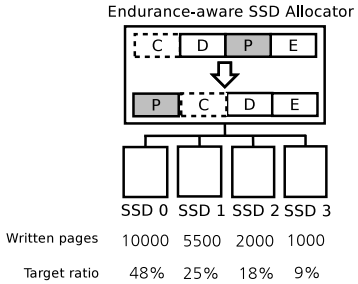


Fig. 6: An example of dynamic SSD allocation.

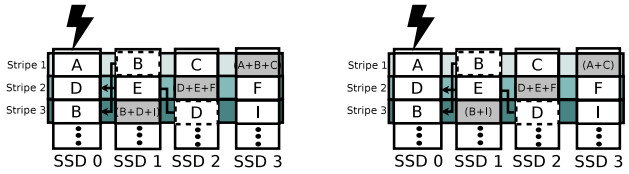
full stripe write that contains duplicated data. The endurance-aware SSD allocator in Figure 5 re-assigns the location of data before the stripe is written to the RAID array. In order to create the age difference among SSDs same as Diff-RAID, the number of issued writes per SSD and the target age distribution 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 re-allocated 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 technique, 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 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.

C. 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



(a) Dedup after parity calculation example

(b) Dedup before parity calculation example

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

complicated. In this section, we describe how the recovery process can be simplified in the proposed method. As explained in Section III-A, 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.

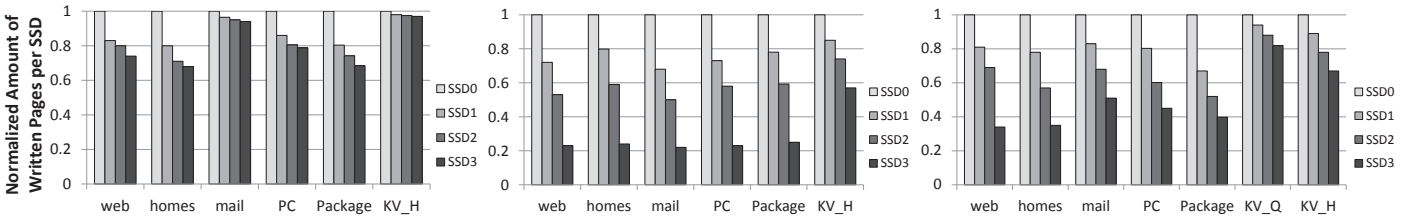
Figure 7 shows the data recovery process when `SSD 0` has failed for the cases that the deduplication is applied after the parity calculation and the reverse order. In both example, data `B` at `Stripe 1` and `D` at `Stripe 3` is deduplicated and the location where the original data were written is linked 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 stripe is 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 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.

IV. Experimental Results

A. Experimental Settings

In order to evaluate the effectiveness of DA-RAID, we conducted a set of experiments on the Linux operating system. As a baseline RAID-5 platform, we selected Linux's software-based RAID subsystem, called a multiple device driver (MD) [13], because it was widely used in enterprise servers. We have implemented both Diff-RAID and DA-RAID on top of DM. For flash disks, we have used FlashBench [15] that emulated the detailed behaviors of NAND flash and flash controllers using host DRAM. To build a RAID-5 system with 4 disks, we attached four instances of FlashBench to the Linux MD. A default chunk size was set to 4 KB. This setup was beneficial to improving overall SSD lifetime because only a



(a) Diff-RAID with the 4KB-sized chunk (b) Diff-RAID with the 512KB-sized chunk (c) DA-RAID

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

small number of parity updates were generated. For a fair comparison, we built a trace-based experimental environment. While running real world applications, we first collected I/O traces that included timestamps, request types, logical addresses, actual data contents, and so on. Using this information, we constructed real block I/O requests and sent them to the Linux's MD layer. This experimental setup allowed us to repeat exactly the same workloads under different RAID setups (e.g., Diff-RAID and DA-RAID). While replaying I/O traces, we measured important performance numbers that included the amount of written data, deduplication ratios, and request sizes.

We used six different I/O traces for the evaluations. Three production system traces, `web`, `homes` and `mail` were from the FIU [14]. All of them included actual data. Three in-house traces, `PC`, `Package`, and `KV` traces were collected while running real-world applications. `PC` was a desktop PC workload such as a web surfing, emailing, and document editing, whereas `Package` captured all of the I/O activities while downloading and installing software packages. `KV` was a key-value store workload that was collected from YCSB [16] running on top of Cassandra [6]. We modified YCSB so that it wrote data with a specific deduplication ratio (e.g., 25% or 50%). The detailed configuration for data generation is described in Section IV-D. Table I summarizes the characteristics of the I/O traces such as amount of writes, average sequential write request size, the ratios of sequential write requests and duplicated data.

B. Inter-SSD Lifetime Fluctuation Evaluation

Figure 8 compares the amount of data written to SSDs for Diff-RAID and DA-RAID under different benchmarks. We first conducted experiments with a default chunk size, 4 KB, on Diff-RAID. As depicted in Figure 8(a), Diff-RAID fails to achieve good age distributions under workloads with the 4KB chunk. Because of its small chunk size, almost all of the write requests become full stripe writes with only few parity updates. Therefore, there is only a little chance for Diff-RAID to control age differences among SSDs. In particular, for the `mail` and `KV` traces, Diff-RAID exhibits very low age differences because of sequential dominant write traffic. To understand behaviors of Diff-RAID in detail, we also perform another experiment with Diff-RAID after increasing a chunk size to 512 KB. As shown in Figure 8(b), Diff-RAID performs very well with the 512 KB chunk - it shows excellent age distributions among SSDs. As expected, this is because partial stripe writes are mostly observed at the RAID level which cause lots of parity updates. Unfortunately, this benefit comes at the cost of more extra writes, that is, more parity updates. We will show how it badly affects overall lifetime of SSDs in the next section. As shown in Figure 8(c), DA-RAID is not seriously affected by a small chunk size. By eliminating duplicate pages from full stripe writes, DA-RAID creates many partial stripe writes. This allows us to dynamically distribute parity updates to different SSDs. Unlike Diff-RAID, as a result, DA-RAID can effectively differentiate the amount of written pages among SSD under workloads where sequential writes are dominant. This benefit is maintained even when the deduplication ratio is relatively low, for example, 25% (`KV_Q`), DA-RAID still shows good age differences, achieving a much lower correlated failure ratio than Diff-RAID.

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
`KV`	1.2	509.89	96	25(Q) or 50(H)

TABLE I: Write characteristics of traces used in our experiments.

C. Endurance Evaluation

As we mentioned earlier, Diff-RAID achieves good age differences only when the chunk size is large enough (i.e., 512 KB). Even though it is beneficial to differentiate the ages among SSDs, it incurs lots of extra party updates, so degrades SSD lifetime in overall. To understand its effect, we measure the total number of pages written to the SSDs under different chunk sizes. Figure 9 shows our experimental results for the various traces. The results shown in Figure 9 are normalized to the number of original page writes from the traces. Since RAID-5 incurs redundant writes (i.e. parity updates), the amount of written pages is always larger than 1 unless data deduplication is used. As shown in Figure 9, Diff-RAID with

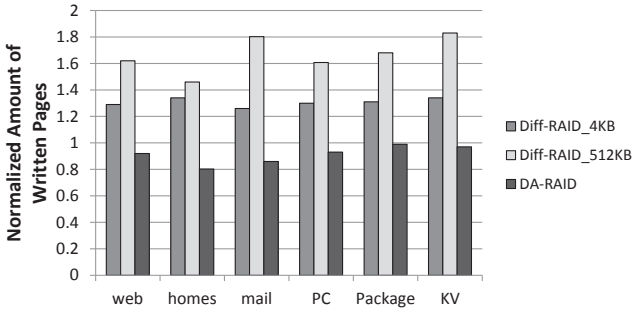


Fig. 9: The amount of written pages under various schemes.

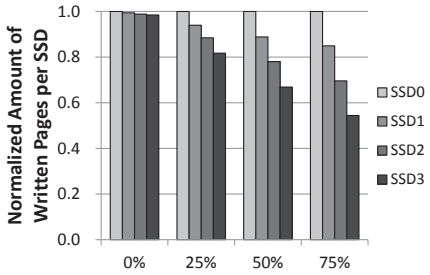


Fig. 10: The difference of written pages among SSDs under varying dedup ratios.

the 512 KB chunk experiences much higher write traffic than one with the 4 KB chunk. Considering its negative impact on longevity of SSDs, the lower correlated failure ratio of Diff-RAID with a larger chunk fades into insignificant. Figure 9 also illustrates the number of written pages in DA-RAID with the 4 KB chunk. From this figure, we can confirm that DA-RAID takes advantage of both fewer parity updates with small chunks and a low correlated failure ratio. Furthermore, by leveraging data deduplication, DA-RAID shows a significant reduction in the amount of written pages. For example, the normalized write traffic is below 1.0 - the amount of data actually written to NAND flash is smaller than that of original data requested to be written by traces. As a result, DA-RAID reduces the amount of written pages by 32% on average and up to 48% for KV trace over Diff-RAID, extending the lifetime of SSD-based RAID by the same amount.

D. Sensitivity Study on Dedup Ratio

As DA-RAID is based on the deduplication technique, the effectiveness of DA-RAID largely depends on the dedup ratio, which is the ratio of amount of eliminated (i.e., deduplicated) data to the size of an original data. We have conducted a simple sensitivity experiment to compare the age difference between SSDs when dedup ratio varies. We ran YCSB with Cassandra for this experiment. In order to control the dedup ratio of the workload, we modified the data generation function of YCSB so that it can generate data to meet the predefined dedup ratio using the data generation function borrowed from IOzone [17]. Figure 10 shows the difference of written pages among two SSDs in DA-RAID under different dedup ratio from 0% to 75% and Diff-RAID with a large chunk size. When there are no duplicate data, 0%, DA-RAID works like Diff-RAID so the age difference of SSDs is not created. As duplicate data increases, the difference of written pages among SSDs also increases from 25% to 75%, reaching target ratio. According to [18], the average dedup ratio is around 30% in production systems. Therefore, we expect that a deduplication ratio would not be a limiting factor for DA-RAID to be used in real-world applications.

V. Conclusion

In this paper, we proposed a lifetime management technique, DA-RAID, for SSD-based RAIDs that can meet the required age differences among SSDs so that multiple SSD failures can be avoided. By using deduplication as a means to convert full stripe writes to partial stripe writes, DA-RAID can satisfy inter-SSD lifetime fluctuations required by Diff-RAID while avoiding the write amplification problem of the existing approach. Our evaluation results show that DA-RAID can successfully sustain the age differences among SSDs for various workloads including ones for which Diff-RAID performs poorly. Furthermore, DA-RAID achieves the skewed age distribution among SSDs without sacrificing the SSD lifetime. DA-RAID improves the lifetime of SSDs by 32% over a naive solution of using a large chunk size in Diff-RAID.

VI. Acknowledgments

This research was supported by National Research Foundation of Korea (NRF) grant funded by Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260) and the Next-Generation Information Computing Development Program through the NRF funded by the MSIP (NRF-2015M3C4A7065645). The ICT at Seoul National University and IDEC provided research facilities for this study.

References

[1] 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.

[2] I. Mir, and A. McEwan, "A Reliability Enhancement Mechanism for High-assurance MLC Flash-based Storage Systems," in *Proceedings on 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011.

[3] Y. Zhang, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Warped Mirrors for Flash," in *Proceedings on 29th Symposium on Mass Storage Systems and Technologies*, 2013

[4] J. Hsieh, and M. Liu, "Configurable Reliability Framework for SSD-RAID," in *Proceedings on Non-Volatile Memory Systems and Applications Symposium*, 2014

[5] 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.

[6] "The Apache Cassandra Project," http://cassandra.apache.org.

[7] "Mongo DB," http://www.mongodb.org.

[8] "RocksDB: A persistent key-value store for fast storage environments," http://rocksdb.org.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," in*Acta Informatica* vol. 33, no. 4, pp. 351-385, 1996.

[10] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proc. 13th USENIX Conference on File and Storage Technologies*, 2015.

[11] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree Filesystem," in *ACM Transactions on Storage*, vol. 9, no. 3, 2013.

[12] "Apache Hadoop Distributed File System," http://hadoop.apache.org.

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

[14] 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.

[15] 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.

[16] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. 1st ACM Symposium on Cloud Computing*, 2010.

[17] "IOzone Filesystem Benchmark," http://www.iozone.org.

[18] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-Aware inline data deduplication for primary storage," in *Proc. 10th USENIX Conference on File and Storage Technologies*, 2012.