Improving RAID Performance Using an Endurable SSD Cache

Chu Li, Dan Feng, Yu Hua, Fang Wang*
Wuhan National Lab for Optoelectronics
School of Computer, Huazhong University of Science and Technology, Wuhan, China
Email: {lichu, dfeng, csyhua, wangfang}@hust.edu.cn

***Abstract*—Parity-based RAID storage systems have been widely deployed in production environments. However, they suffer from poor random write performance due to the parity update overhead, i.e., *small write problem*. With the increasing density and decreasing price, SSD-based caching offers promising opportunities for improving RAID storage I/O performance. However, as a cache device, frequent writes to SSD leads to being quickly worn out, which causes high costs and reliability problems. In this paper, we propose an efficient cache management scheme by Keeping Data and Deltas (KDD) in SSD. KDD dynamically partitions the cache space into *Data Zone* (DAZ) and *Delta Zone* (DEZ). DAZ stores data that are first admitted into SSD. On write hits, KDD writes the data to RAID storage without updating the parity blocks. Meanwhile, the deltas between old version of data and the currently accessed data are compactly stored in DEZ. In addition, KDD organizes the metadata partition on SSD as a circular log to make the cache persistent with low overhead. We evaluate the performance of KDD via both simulations and prototype implementations. Results show that KDD effectively reduces the small write penalty while significantly improving the lifetime of the SSD-based cache.**

I. Introduction

Redundant Array of Independent Disks (RAID) based storage architecture [1] is one of the most popular choices to provide reliable and high performance storage. Among different RAID levels, parity-based solutions like RAID-5 (and RAID-6) have been widely used in large scale data centers due to cost-effectiveness. They can tolerate one (and two) disk failure while providing comparable read performance with RAID-0. However, the write performance is severely degraded due to the parity update overhead, especially for workloads with lots of random small sized requests [2]. For example, each data update in RAID-5 needs two read and two write disk I/O operations, which is called the *small write problem* [2]. Many schemes have been proposed to eliminate the small write penalty of parity-based RAID systems. One of the ways is buffering parity/data blocks in Non-volatile RAM (NVRAM). With caching, small writes can be reduced to full stripe writes. However, the access time reduction they can provide is limited due to the poor locality at the disk I/O level [3], [4].

Flash-based Solid State Drives (SSDs) have gained popularity over the past few years. These devices have much lower latencies but higher prices than magnetic hard disk drives (HDDs). Therefore, SSDs are often deployed as a cache layer in front of HDD-based storage systems to achieve the performance of flash with the cost of disk [5]. Although SSD has some superior features such as high performance, low power, and non-volatility, it suffers from endurance issue since flash memory can only sustain a limited number of program/erase cycles. While the caching layer is more write intensive than its storage counterparts, typical data-center workloads can wear out the SSD cache device within months [6], [7].

Write-through and write-around caching policies are largely used in production environments to reap the benefits of the back-end RAID storage systems, such as high availability, and data reliability [8], [9]. However, they only reduce read latencies and fail to address the small write problem. Lee et al. [10] propose LeavO to reduce the parity update overhead while maintaining reliability against the SSD failure. LeavO delays parity update by storing redundant versions of data in SSD. However, LeavO wears out the SSD cache more quickly due to the requirement of storing metadata persistently on SSD. Moreover, RAID re-synchronization (i.e., reading data to re-generate parities) is required after SSD failures in LeavO, leading to a window of vulnerability, in which a disk failure causes permanent data loss. Consequently, it is critical to extend the lifetime of the SSD cache.

In this paper, we propose an efficient SSD-based caching solution to overcome the small write problem of RAID storage while taking the SSD's write endurance issue into account. The key idea of our scheme is Keeping Data and Deltas (KDD) in SSD. KDD dynamically partitions the cache space into Data Zone (DAZ) and Delta Zone (DEZ). DAZ stores data pages which are first admitted into SSD. When a write request arrives, if the data is found in cache, KDD stores the compressed XORs (delta) of the two versions of data in DEZ, and writes the data to RAID without updating the parity. The stale parity blocks will be updated in the background. Moreover, in order to minimize the overhead of cache metadata updates, KDD integrates the metadata of cached data and deltas, and maintains a fixed metadata partition on SSD which is managed as a circular persistent log.

We evaluate KDD using both a cache simulator and kernel-level implementations under synthetic and various real-world workloads. Trace-driven simulation results show that KDD significantly reduces cache write traffics compared to the write-through policy and LeavO, improving the SSD lifetime by up to $5.1\times$. Our evaluations with prototype implementations

*Corresponding author: Fang Wang (wangfang@hust.edu.cn).

show that KDD can effectively boost the I/O performance of the parity-based disk array.

The rest of this paper is organized as follows. In Section II, we present the background and motivation of this work. In Section III, we describe the detailed system design of our proposed KDD. Extensive performance evaluations are presented in Section IV. We discuss the related work in Section V and conclude this paper in Section VI.

II. Background and Motivation

A. Flash Memory based SSD

Current commercial SSDs use NAND flash memory as storage medium with numerous advantages, such as high speed random access, low power consumption, and shock resistance [3]. The flash memory can be classified into Single-Level-Cell (SLC), Multi-Level-Cell (MLC), and Triple-Level-Cell (TLC) flash. SLC flash stores one bit per cell, whereas MLC/TLC flash store two/three bits per cell. With the rapid increase of capacity and decrease of cost, SSD has been widely used in real-world storage systems. SSDs are constructed from an array of flash packages that are connected to a flash controller. Each package has one or more dies, which in turn have multiple planes; Each plane has many blocks, which are further divided into many pages. The flash controller improves SSD performance by exploiting multi-level parallelism in SSD.

Flash memory shows superior features as presented above, however, it suffers from write endurance problem. In flash memory, read and program (write) operations are performed in the unit of page with size ranging from 2KB to 16KB. Erase is performed in the unit of block which typically contains 64 to 128 pages. Flash memory does not support in-place update. Blocks can be only written after being erased and only a finite number of erasures are sustained before wearing out. Typically, SLC flash supports 100K erasures per block, while the MLC flash supports as low as 5,000 to 10,000 erasures per block. Furthermore, the write endurance is getting worse along with the flash density increasing and feature size scaling [11].

B. Problems in Previous Schemes

SSDs have been largely adopted as write-through or write-around caches above disk-based RAID storage systems in production environments. These caching policies simplify the consistency model since all writes are stored in RAID before being acknowledged to applications. Thus, they help to deliver a recovery point objective (RPO) of zero (i.e., no data loss) under SSD failures [9], retaining the high availability and reliability of RAID. However, they cannot reduce the small write penalty since each write still causes parity update.

Based on write-through policy, LeavO [10] postpones parity update by storing both old and new data in SSD cache. However, the redundant versions of cached data in LeavO not only consume more cache space, but also need to be persistently stored in SSD. The former can decrease buffer hit ratios while the latter requires extra metadata I/Os to SSD. As a result, LeavO produces more SSD write traffics than the write-through policy. Both write-through and LeavO do not take the SSD write endurance into account, making the cache far less effective as discussed next.

The SSD caching layer is more write intensive than the primary storage because both writes in the workloads and reads that miss the cache can cause SSD write operations [12], [13]. Due to the limited endurance of flash, a typical data center workload can wear out an MLC SSD cache within months [7]. Replacing a worn-out SSD cache frequently not only increases costs but also hurts the reliability and availability of the storage system. Taking the RAID-5 storage with a LeavO-cache for example, once the SSD cache wears out, a single disk failure before the RAID is re-synchronized can cause data loss, and user requests will be adversely affected by the re-synchronization of RAID storage. This motivates us to propose an efficient cache scheme to extend the lifetime of the SSD while improving the storage I/O performance.

C. Content Locality

Previous studies indicate that strong content locality exists in real world applications [14], which means data in the system share similarity with each other. For example, research literatures have reported that in many data intensive applications, only 5% to 20% of bits inside a data block are changed on a write operation [14], [15]. More recently, file system workload analysis implies that many updates are less than 10% different from the previous contents of the block [16].

Based on these observations, KDD exploits the content locality of workloads to reduce the write traffics to SSD cache. Instead of maintaining redundant versions of data in cache as LeavO, KDD packs multiple deltas with respect to the old cached pages and stores them compactly in SSD cache. In this way, KDD is much more cache space-efficient than LeavO. More importantly, it has the potential to reduce cache write traffic than both write-through policy and the LeavO by deploying efficient cache management schemes. Although recent studies have proposed techniques to extend SSD's lifetime via delta compression [17], [18], they are designed in the Flash Translation Layer (FTL) inside SSD devices. We focus on using commodity SSD based cache in front of RAID storage, and face unique challenges presented by caching.

III. System Design of KDD

KDD is designed to reduce the writes to SSD cache while overcoming the small write problem of parity-based RAID storage. In this section, we first introduce an overview of the KDD and then describe the KDD design in detail. Finally, we discuss the recovery process from various failures in KDD.

A. Overview of KDD

Figure 1 shows the overview of our proposed KDD. A single SSD is used as a cache device for the disk-based RAID primary storage, which employs a parity-based configuration, such as RAID-5/6. SSD cache space is logically divided into Data Zone (DAZ) and Delta Zone (DEZ). When data of user requests are first admitted into SSD due to read/write misses, they are stored in DAZ. When write requests find cached data

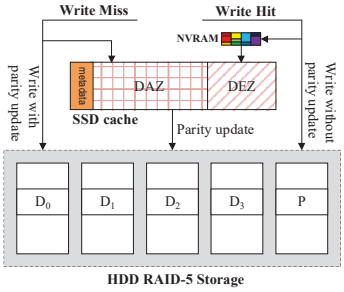


Figure 1: Overview of KDD

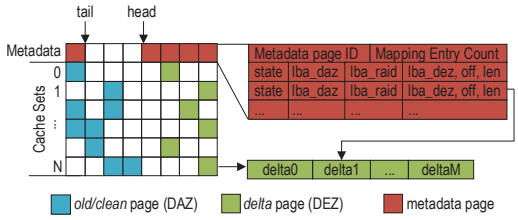


Figure 2: On-disk format of SSD cache

in DAZ, the deltas which are the compressed XORs of the current version of data and the old version are compactly written to DEZ pages. On a read hit request, KDD returns the data by combining the old version of data in DAZ and the latest associated delta. To make the cache persistent, mapping information for the cache is stored on a fixed partition in the beginning of the SSD.

Two interfaces are added between the SSD cache and RAID storage to reduce the parity update overhead, the *write-without-parity-update* operation and the *parity-update* operation [10]. On a write miss request, the data will be cached in DAZ and written to RAID storage with conventional parity update operations (i.e. read-modify-write or reconstruct-write). On a write hit request, KDD only updates the data in RAID storage without re-computing the new parity, thus reducing the overhead of parity update. The stale parity blocks in RAID will be updated in the background.

KDD aims to achieve reliability, availability, and cost-efficiency using the following goals as guidelines: (1) offer tolerance to power failures, disk failures, and SSD failures with a recovery point objective (RPO) of 0, (2) reduce the writes to SSD as many as possible, and (3) minimize the system overhead introduced due to KDD itself. Details of the data structures and algorithms to implement KDD towards these design goals are given in the following subsections.

B. Cache Space Management

KDD adopts the N-way set-associative method to organize the SSD cache. The cache space is divided into many cache sets, each containing a fixed number of pages. Basically, there are two ways to partition the cache space for DAZ and DEZ. One possibility is to use fixed partitions by maintaining the two zones in separate cache sets. However, it is hard to determine the appropriate size of these zones. On the one hand, DEZ should have enough pages to maintain the deltas efficiently. On the other hand, the size of DEZ should be small to cache more unique pages in DAZ for higher cache hit ratios. We deploy an alternative approach in KDD, where the pages of the two zones are dynamically allocated and mixed in each cache set, thus adapting to different workload environments. In particular, DAZ pages are located in cache sets via hash functions, while DEZ pages are evenly distributed across the cache sets. To exploit the spatial locality of workloads, DAZ pages in the same parity stripe are mapped to the same cache set, and thus they can be reclaimed together during cache cleaning which will be presented later.

The state of each page in the cache sets can be represented by *free, clean, old,* and *delta*. Among these pages, *clean* and *old* pages are contained in DAZ, while *delta* pages are contained in DEZ. Initially, all pages in an empty SSD cache are in *free* state. When allocating a cache page for DAZ, KDD finds a *free* page in the corresponding cache set, and the data will be cached in the DAZ page with its state converted from *free* to *clean*. When a write request hits on a *clean* page in DAZ, the page state will be changed to *old* and the delta is stored in a small *staging buffer* which is managed in a FIFO manner. When the buffer is full, multiple deltas are compacted into one page and committed to a DEZ page with its state altered to *delta*. DEZ pages are not updated in-place but allocated on demand, similar to the log-structured writes in LFS [19] and DCD [20]. The difference is that DEZ pages are not required to be consecutive in SSD cache. When allocating a page for DEZ, KDD always chooses a free page from the cache set which has the least number of DEZ pages. When the SSD cache is full, *clean* pages are evicted while *old* and *delta* pages are only reclaimed in a cleaning thread triggered by several system events (see Section III-D).

To make the SSD cache persistent, the cache metadata, including the mapping information for cached data and deltas, also need to be persistently stored on SSD. However, issuing metadata updates upon each cache insertion/eviction can cause a large number of cache writes [21]. To address this problem, KDD organizes the metadada partition on SSD as a *circular persistent log*. Two counters are maintained to indicate the head and the tail of the log space. New mapping entries are first accumulated in a *metadata buffer*. When there are enough entries in the buffer to fill a page, they are written to the tail of the log, and the tail counter is incremented. KDD reclaims metadata pages from the head of the log to ensure that the log usage is under a certain threshold. On recovery from power failure or restart, KDD can rebuild the mapping information by scanning the log. To avoid data loss against power failures, the staging buffer, the metadata buffer, and the two counters (head/tail) are all stored in the NVRAM (e.g., battery-backed RAM) which is commonly used in storage arrays. The SSD cache organization is shown in Figure 2.

C. Data Structures

KDD maintains an in-memory *primary map* and NVRAM buffers for efficient cache management. The *primary map*

maintains all cache page states and allows translating the storage block addresses of I/O requests to the block addresses of the cached data (and deltas) on flash. It is implemented as a simple array indexed by the logical block addresses (LBAs) of all pages in each cache set. NVRAM buffers are used to accumulate newly updates and commit to flash in the unit of page. Newly generated deltas are first stored in the staging buffer and the old deltas will be invalidated. The metadata buffer is a mapping table maintaining the latest mapping entries for DAZ pages. Several important fields of mapping entries in the primary map and the metadata buffer are described as follows:

- *state* indicates the cache page state, i.e. *free, clean, old,* and *delta*.
- *lba_daz* represents the offset of the cached data page on SSD.
- *lba_raid* represents the offset of the corresponding data page in the RAID storage.
- *lba_dez,off,len* indicates the location of the associated delta in a DEZ page on SSD. When the deltas are still buffered in the NVRAM, the values of these variables are assigned -1.
- *valid count* indicates the number of valid deltas in the DEZ page. The value will be decremented when a delta in the page is invalidated, and the DEZ page cannot be freed until the *valid count* reaches zero.

As shown in Figure 3, mapping entries in the primary map have different required fields for different kinds of pages. For example, the tuple *(lba_dez, off, len)* is only maintained for *old* pages, while *valid count* is only maintained for *delta* pages. Once a new *clean* page is allocated, the corresponding mapping entry in the primary map is updated and also inserted to the metadata buffer. On a write hit request, only the primary map is updated and the changed mapping entries are not inserted to metadata buffer until the deltas residing in the staging buffer are committed to DEZ. When a DAZ page is reclaimed due to cache eviction or cache cleaning, a mapping entry with its *state* field marked as *free* is also stored in the metadata buffer to reflect this change. Allocation and deallocation of DEZ pages are not recorded in the metadata buffer because the mapping information is embedded in mapping entries of *old* pages. It should be noted that write coalescing can be used for the NVRAM buffers. For example, an entry in the metadata buffer can be overwritten by a new entry having the same *lba_daz* value, and only the newest version of delta for one DAZ page is maintained in the staging buffer.

The mapping table in the metadata buffer is always committed to a new metadata page on flash pointed by the tail counter. The persistent log can run out of space, thus requiring garbage collection. KDD uses an *oldest first* policy, i.e. the metadata page pointed by the head counter is chosen as the candidate page for cleaning. Valid mapping entries in the candidate page are reinserted to the metadata buffer, and those entries will be finally committed to the tail of the log. As an optimization, KDD maintains a list in memory for each

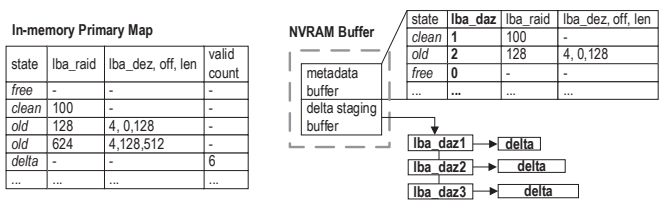


Figure 3: Key data structures in KDD

metadata page, and all mapping entries in the *primary map* are linked to the corresponding list at runtime. Thus, KDD only needs to scan the in-memory list to copy the valid mapping entries instead of reading the metadata page on flash for garbage collection. Garbage collection will cause extra write traffics, and the overhead depends on the total size of the metadata partition and the characteristic of the workloads [19]. Generally, configuring the persistent log with more metadata pages can reduce the cleaning cost at the expense of crash recovery performance. Fortunately, the metadata I/Os are a very small fraction of cache write traffics even with a small metadata partition size, as shown in Section IV-A3.

The primary map can easily fit into memory on modern storage servers. Assuming a 4KB page size for both SSD and RAID, for every mapping entry, KDD stores 4 bytes for each LBA (supporting capacity of 16TB), 1 bytes for the state, and 3 bytes for the *(off, len)* tuple, plus 12 additional bytes for pointers (encoded as integers) which are used for maintaining LRU lists and metadata page lists (not shown in Figure 3). Thus, the ratio of the *primary map*'s memory overhead to SSD size is about 0.59% ($= 24/4096$), or 6MB per GB.

D. Flushing Policy

KDD postpones the parity updates by keeping *old* pages and associated deltas in SSD, resulting in many stale parity blocks on RAID storage. New parity blocks are generated and flushed to RAID storage in a background cleaning thread, which is triggered by several system events. For example, when the total size of the *old/delta* pages exceeds a certain threshold, or the system has been idle for a certain period, the cleaning thread will be woken up for parity updating and cache page reclaiming.

When updating one parity block, KDD finds all cached data pages and associated deltas of the parity stripe, and recalculates the new parity using either read-modify-write or reconstruct-write mode. Reconstruct-write is only used when all data blocks within the stripe are residing in SSD, in which case KDD computes the new parity block by performing XOR operations on all the data blocks and decompressed deltas of the parity stripe. Otherwise, KDD uses read-modify-write for parity updating by reading the stale parity from RAID and XORing it with all decompressed deltas of this parity stripe. In this way, KDD reduces the I/Os to RAID storage by exploiting the spatial locality of workloads.

There are two schemes for reclaiming the *old/delta* pages after updating the parity blocks. The first one is to combine the

old page with associated deltas to generate the latest version of data, and then write them to DAZ pages marked as *clean*. In this way, another write hit on this page can benefit from the parity delay scheme. However, the benefit may be marginal at the expense of more cache writes, because the victim pages are commonly cold. Therefore, we choose the second scheme for the sake of simplicity, which simply reclaims the *old* pages and invalidates the deltas during cache cleaning.

E. Failure Handling

KDD provides high reliability and availability by tolerating both power failures and device failures.

1) Power Failures: In case of a power failure, the head and tail counters of the metadata persistent log are reconstructed from NVRAM. The *primary map* is rebuilt by reading all metadata pages and "replay" the mapping entries from the head to the tail. Then the mapping entries in the NVRAM metadata buffer are copied back to the *primary map*. Finally, mapping entries corresponding to the deltas residing in the NVRAM staging buffer are also updated.

2) Device Failures: Disk failures can occur in either the SSD or HDDs. While KDD delays the parity update for RAID storage, a number of stripes in RAID have stale parity blocks. However, on an SSD failure, RAID storage can be re-synchronized through reconstruct-write because data blocks are always dispatched to RAID. If a HDD fails, KDD first updates all parity blocks using the *parity_update* interface and then triggers the rebuilding process at the RAID layer.

IV. Evaluation

In this section, we conduct extensive simulations and prototype evaluations to assess the effectiveness of the proposed KDD. We first show the hit ratios and SSD write traffics of different algorithms in Section IV-A using trace-driven simulations. Then we compare the average response time of KDD with existing approaches through prototype implementations in Section IV-B.

A. Simulation Experiment

1) Cache Simulator: We developed an SSD cache simulator to evaluate the effectiveness of our proposed KDD with respect to the SSD write traffic, and the hit ratios compared to previous approaches, including the write-through (WT), write-around (WA), and the LeavO. Write-back caching is not evaluated because it cannot prevent data loss under SSD failures. Each algorithm is implemented as a separate module in the simulator. The simulator first converts raw traces into a uniform format and then processes trace requests one by one according to the timestamp of each request. After each test, detailed statistics such as hit ratios and cache write traffics are reported. NVRAM buffers are also included in the simulator and managed as described in Section III. The simulator can be configured with many parameters, such as cache size, page size, cache associativity, NVRAM buffer size, etc. In our tests, the page size of the cache and the two NVRAM buffers are all set to 4KB. For fair comparisons, the NVRAM buffer is employed in all of the algorithms.

TABLE I: Characteristics of I/O workload traces

Workload	Unique Pages(x1,000)			Requests(x1,000)		Read Ratio
	Total	Read	Write	Read	Write	
Fin1	993	331	966	1,339	5,628	0.19
Fin2	405	271	212	3,562	917	0.80
Hm0	609	488	428	2,880	5,992	0.33
Web0	1,913	1,884	182	4,575	3,186	0.59

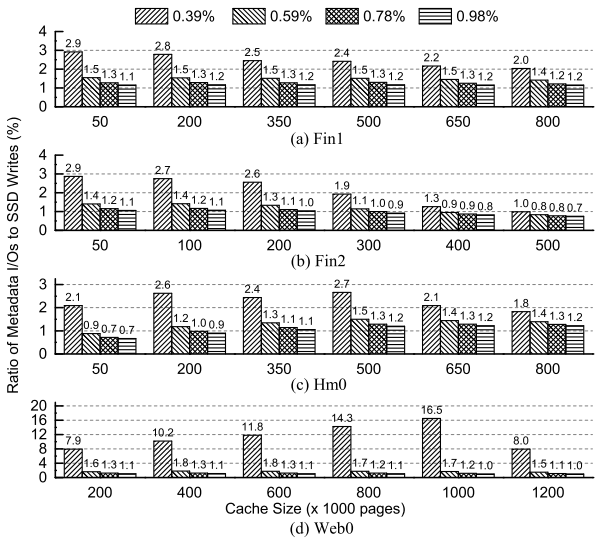
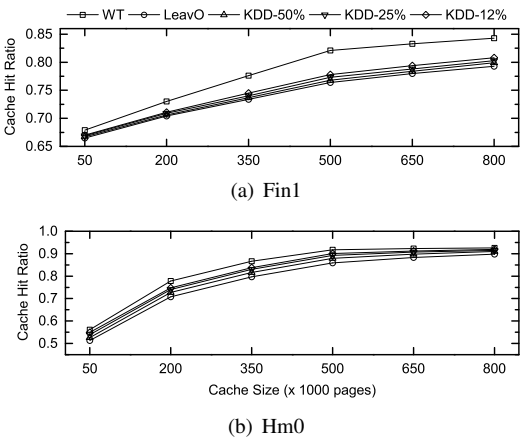


Figure 4: Effects of the metadata partition size on metadata I/Os under various workloads and cache sizes

2) Workloads: We choose several block-level traces obtained from the Storage Performance Council (SPC) [22] and Microsoft Cambridge Server (MCS) [23]. The two financial traces (Fin1 and Fin2) are collected from OLTP applications running at two large financial institutions. The MCS traces are collected from 36 volumes containing 179 disks on 13 servers for one week in a data center. The access patterns of these traces are representative of many enterprise data centers. Among the MCS traces of different servers, we only show the results of the first volume of Hm and Web (denoted as Hm0 and Web0) due to lack of space. The characteristics of these workloads are summarized in Table I (with 4KB page size). Among these traces, Fin1 and Hm0 are write dominant, while Fin2 and Web0 are read dominant. The content locality of a workload is application specific [14], [15], [24] and can vary in different applications and periods. To evaluate KDD, we assume the delta compression ratio values follow Gaussian distribution [14], [17] with an average equaling 50%, 25%, and 12% (denoted by KDD-x%), representing workloads with low, medium, and high level of content locality, respectively.

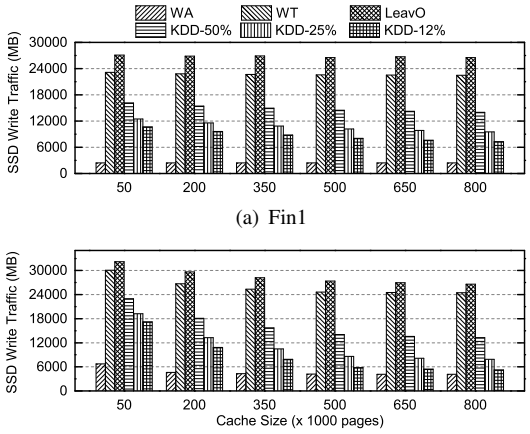
3) Simulation Results: To run the evaluations, KDD has to determine an appropriate configuration for the metadata partition size. A big metadata partition can adversely affect both recovery performance and cache hit ratios. However, if it is too small, significant metadata I/Os will be caused due to the cleaning cost as mentioned earlier. Consequently, we first conduct experiments to explore the effects of the partition size on the number of metadata I/Os. Due to the



(a) Fin1

(b) Hm0

Figure 5: Cache hit ratios of different approaches under the write-dominant traces



(a) Fin1

(b) Hm0

Figure 6: SSD write traffics under the write-dominant traces

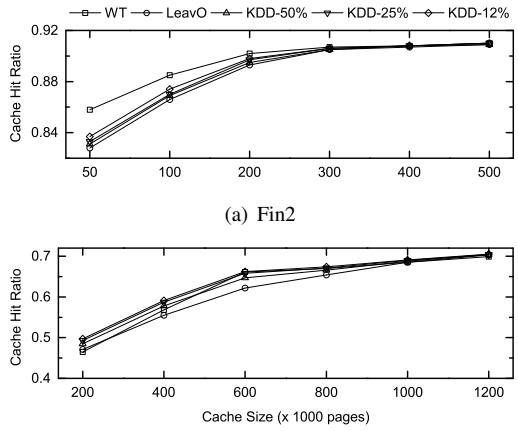
space limitation, we only report the results assuming the workloads have medium content locality, and similar trends are observed under other circumstances. Figure 4 shows the ratios of metadata I/Os to the total cache write traffics under various workloads and cache sizes. The size of the metadata partition is configured with 0.39%-0.98% of the SSD size. As shown in the figure, when 0.59% of the SSD capacity is reserved for metadata partition, the proportions of metadata I/Os in total cache write traffics are less than 1.55%, 1.42%, 1.51%, and 1.79% under the four workloads respectively. Therefore, we choose this configuration in the following experiments.

Figure 5 shows the cache hit ratios of different approaches for the write-dominant traces. Both LeavO and KDD have lower cache hit ratios than that of WT because they consume some cache space for caching new versions of cached data and *delta* pages, respectively. While WT only keeps one version of each cached page, more unique pages can be maintained in SSD cache, resulting in better hit ratios. However, KDD stores the small-sized deltas compactly in SSD, therefore, it convincingly outperforms LeavO for different levels of content locality. In addition, the stronger content locality the workloads have, the higher hit ratios KDD can achieve. Note that we omit to show the hit ratios of WA in the figure because all writes bypass the cache in write-around caching.

One goal of KDD is to extend SSD lifetime by reducing cache write traffics. Figure 6 compares the SSD write traffics for each approach under the write-dominant traces. Among the caching policies except WA, the write traffics to SSD consist of read-fill operations on read misses and cache write operations on all write requests. In addition, cache metadata updates are required in LeavO and KDD. WA allocates only on read misses, and shows the least number of cache writes under these write-dominant traces. KDD incurs much less cache write traffics than all tested caching policies except WA. Specifically, compared to WT under the Fin1 workload, the amount of data written to SSD is decreased by up to 37.6%, 57.6%, and 67.6% for KDD-50%, KDD-25%, and KDD-12% respectively. Under the Hm0 workload, KDD reduces the SSD write traffics by up to 45.7%, 67.7%, and 78.6% compared to WT under different levels of content locality. The reason for this large reduction lies in KDD's efficient management of metadata partition and the space-efficient DEZ which packs multiple updates into one *delta* page. The results also reveal that the performance gain of KDD is proportional to the content locality. LeavO shows the largest number of SSD writes due to lower hit ratios and extra metadata updates. Compared to LeavO, KDD reduces the SSD write traffics by up to 47.2%-72.6% and 50.1%-80.4% under two traces respectively, thus extending the lifetime of SSD by up to $5.1\times$.

Figure 7 shows the hit ratios of WT, LeavO, and our scheme under the read-dominant workloads. LeavO shows the smallest hit ratios in most cases due to the inefficiency of consuming extra cache space. Under the Fin2 workload, the hit ratios of KDD lie between those of WT and LeavO, but the performance gap among LeavO, KDD, and LeavO becomes narrower as the cache size increases. Under the Web0 workload, KDD even outperforms WT when the cache size is small. The reason behind this abnormality lies in the characteristics of Web0 and the different cache replacement schemes in WT and KDD. On the one hand, after a detailed analysis of the Web0 workload, we found its temporal locality of write requests is much higher than that of read requests. On the other hand, while WT only has clean pages in cache, the cached pages are evicted strictly in LRU order. However, KDD maintains *old/delta* (can be regarded as 'dirty') pages, which are not evicted immediately but in the cleaning thread triggered by several events. Thus, KDD has the potential to keep more 'dirty' pages in cache than WT. Combining the two factors, KDD generates more hit rates in such cases.

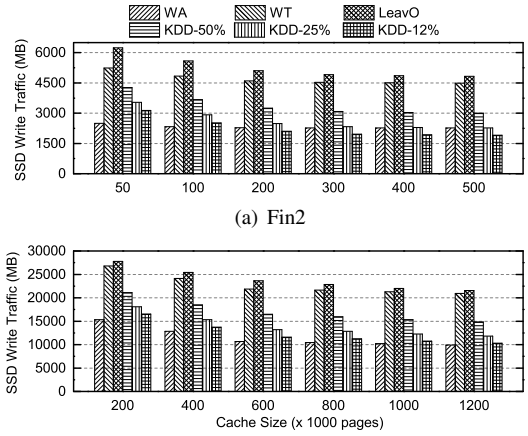
Figure 8 compares the SSD write traffics when the read-dominant workloads are used. Compared to WT, KDD can reduce the write traffics by up to 57.5% and 50.6% for Fin2 and Web0 respectively. Compared to LeavO, the write traffics are decreased by up to 60.5% and 52.1% for the two workloads. The improvement under the read-dominant workloads is smaller than that under write-dominant workloads especially



(a) Fin2

(b) Web0

Figure 7: Cache hit ratios of different approaches under the read-dominant traces



(a) Fin2

(b) Web0

Figure 8: SSD write traffics under the read-dominant traces

when the cache size is small. Because under read-dominant traces, a large number of SSD write traffics are caused by read-fill operations (i.e. cache writes on read misses), which cannot be reduced in KDD. Another observation is that the write traffics gap between WA and KDD becomes narrower under read-dominant workloads. For Fin2 under large cache sizes (no less than 200 x 1000 pages), KDD-12% even has less cache writes than WA.

B. Implementation and Measurement

1) Experiment Platform: We have implemented the KDD prototype by modifying both the Linux Software RAID and the EnhanceIO [25] modules with 64-bit Linux kernel version 3.13. For delta comression/decompression in KDD, we use the lzo library [26] due to its superior performance. All our experiments were conducted on a platform of server-class hardware with an 8-core Intel Xenon Processor, 16GB RAM, 15 7,200 RPM drives of 1TB each, and a single 120GB SSD. We use 1GB flash device as the cache. The SSD cache operates on 4KB pages and uses the LRU replacement algorithm for all caching policies. The backing store was configured as a 5-disk

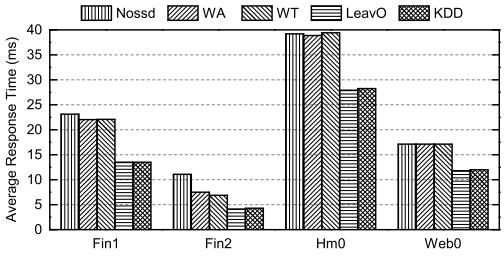


Figure 9: Average response time of different approaches under various workloads

RAID-5 with a default 64KB chunk size. We disable all the drive look-ahead and drive volatile cache using *hdparm*.

We evaluate the performance of KDD using both an open-loop model and a closed-loop model. In open-loop model, I/Os are issued according to the request time. Whereas in closed-loop model, requests are generated back to back with a limited request queue (i.e. equal to the number of request threads). For the open-loop evaluations, we use RAIDmeter [27] to replay each workload for 30 minutes and measure the average response time. For the closed-loop model, we use FIO 2.2.10 benchmark tool [28] to generate synthetic workloads with the zipf distribution which is common among many real-world workloads. A medium level of content locality with the average delta compression rate equaling 25% is chosen in the following evaluations. We run each experiment three times and report average numbers.

2) Results of trace-replay: KDD significantly reduces the write traffics to SSD cache compared to LeavO, as shown in the previous trace-driven simulations. However, KDD requires extra SSD read I/Os and delta compression/decompression on write/read hit requests. In this first experiment, we evaluate the performance of KDD in terms of average response time under various workloads against other approaches. Figure 9 shows the average response time of each approach under the selected traces. WA and WT outperform Nossd (RAID without an SSD cache) only for Fin2 which has a large proportion of read accesses. As shown in Figure 9, KDD outperforms both WT and WA under all workloads because small write penalty is effectively reduced in KDD. Compared to Nossd, KDD reduce the average response time by 41.7%, 61.2%, 28.0%, and 30.1% for Fin1, Fin2, Hm0, and Web0, respectively.

KDD shows the similar performance compared to LeavO, which means KDD effectively hides the overhead of the delta processing. The reasons can be explained as follows. For read hit request on an *old* page, KDD can read the data and delta concurrently, which is quite efficient due to the parallelism inside SSD. And it takes only tens of microseconds to decompress the delta and combine it with the data. For write hit requests in both KDD and LeavO, the data will always be dispatched to the RAID storage. This usually takes milliseconds and KDD has enough time to generate the deltas.

3) Results of FIO Benchmark: We further assess the effectiveness of KDD using the FIO benchmark with Zipfian write pattern of α=1.0001 (e.g. [29]). The benchmark reads/writes a total of 4GB data with 4KB block size. The number of

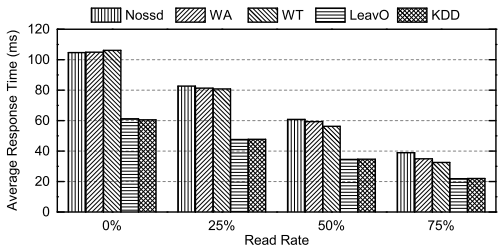


Figure 10: Average response time of different approaches under FIO benchmark

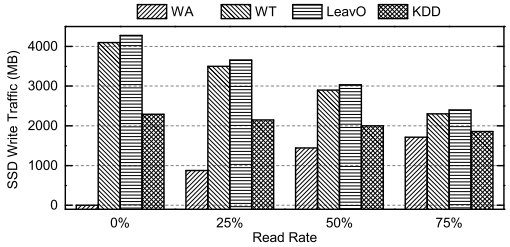


Figure 11: SSD write traffics under FIO benchmark

threads is set to 16 to bound the I/O latencies to about 100ms. The working set size for this workload is 1.6GB, larger than the SSD cache size. We run the tests with different read rates to explore the performance of KDD compared to other approaches under different environments. The read rate ranges from 0% to 75%. Evaluations with 100% read rate workload are omitted because in that case both LeavO and KDD will degrade to WT.

Figure 10 compares the performance of different approaches in terms of average response time. Again, KDD shows comparable performance compared to LeavO, which indicates that even under workloads with multiple concurrent I/O streams, the delta processing of KDD will not be a performance bottleneck. The reason is that both CPU speed and the SSD random access are much faster than the HDD-based RAID storage I/Os. As shown in Figure 10, KDD reduces the average response time by 42.1%-43.3% and 42.8%-32.3% than Nossd and WT respectively. WT and WA slightly outperform Nossd only under synthetic workloads with high read rates, because the small write is much slower than read operations in parity-based disk array.

Figure 11 shows the SSD write traffics of WA, WT, LeavO, and KDD under synthetic workloads with different read rates. We can observe that WA has the least number of cache writes as validated in our simulation evaluations. As the read rate increases, WA produces more cache writes getting close to KDD. Compared to WT and LeavO, KDD effectively reduces the amount of data written to SSD. Specifically, compared to WT, KDD reduces the write traffics by 44.0%, 38.6%, 31.0%, and 19.4%. Compared to LeavO, KDD reduces the write traffics by 46.4%, 41.3%, 34.0%, and 22.6% for the synthetic workloads respectively. Again, this indicates our KDD is more efficient under write-dominant workloads.

In summary, WT and WA fail to accelerate the I/O performance of parity-based storage systems unless the workloads are much read intensive. WA has much less cache write traffics, making it more cost-effective than WT for SSD-based cache. Compared to WA/WT, KDD can notably reduce the I/O latencies under all tested representative workloads in data centers and synthetic workloads. Both LeavO and KDD can boost the I/O performance by reducing the small write penalty, however, only KDD takes the SSD's write endurance into account. Compared to LeavO, KDD achieves similar I/O performance but significantly reduces the cache write traffics. The comparison of different caching policies for SSD-based RAID cache is summarized in Table II.

TABLE II: Comparison of different caching policies

	WT	**WA**	**LeavO**	**KDD**
I/O Latency	High	High	Low	**Low**
SSD endurance	Bad	Good	Bad	**Good**

V. Related Work

A. Overcoming Small Write Problem

Parity Logging [2] reduces the cost of small write request by accumulating the *parity update images* (the XOR result of the old and new data) in a log disk and updating the out-of-date parities with large sequential accesses when the log disk is full. AFRAID [30] only writes the new data without waiting for the parity to be updated for certain periods. The stripes which have stale parity blocks are marked un-redundant in NVRAM, and the pending parity updates are processed in idle periods. However, AFRAID cannot always tolerate one disk failure like RAID5.

Many studies use NVRAM for data and/or parity caching to reduce the write overhead [31]–[33]. Using buffer caching schemes can not only reduce the write traffics to RAID, but also help to construct full-stripe writes. However, the NVRAM buffer size is often quite small for power and cost efficiency, limiting the effects of reducing small write penalty. Flash memory is much cheaper than NVRAM and produces the possibility to address this problem, e.g. the LeavO [10] and our KDD. However, flash suffers from write endurance issue, which is also addressed in KDD.

Other schemes eliminate the small write penalty by changing the standard RAID data organization. For example, Hot-mirroring [34] and AutoRAID [35] deploy a two-level storage hierarchy with RAID1 storing active (hot) data and RAID5 storing inactive (cold) data. Thus, they can combine the performance benefits of mirroring and the storage efficiency of RAID-5. Dynamic striping [36] reorganizes the parity stripes via either a LFS based method or Virtual Striping. It redirects updates to new stripes and always dispatches full-stripe writes to RAID storage, eliminating the small write penalty.

B. Flash-based SSD Caching

SSDs have been largely adopted as cache devices to boost the disk-based storage systems. Mercury [8] deploys flash based cache in the hypervisor's virtual I/O stack to support a variety of protocols. It uses write-through caching to avoid compromising reliability, availability, and data management

features of the shared storage system. Policies based on write-back caching have been proposed for better performance while insuring data consistency at the network storage. For example, Koller et al. [9] propose consistent write-back policies by trading off data staleness for performance and data consistency. Qin et al. [37] propose two write-back based policies to provide reliability guarantees relying on applications which issue write barriers for data persistency.

Although write-back based policies can offer critical performance benefits, they suffer from data loss under cache device failures. Many studies employ RAID techniques to improve the reliability of the caching layer. Arteaga et al. [21] propose a cache-optimized RAID technique for better cache utilization and cost-efficiency, where clean and dirty pages are stored in a RAID-0 and a RAID-1 fashion, respectively. Oh et al. [38] propose SRC (SSD RAID as a Cache) which adopts RAID technique and log-structured approach in the cache layer. Unlike these approaches, our KDD provides reliable flash caching with a single SSD, which is more cost-effective.

C. Enhancing lifetime of SSD Cache

Deduplication and compression techniques have been applied to SSD cache layer. Flaz [39] deploys transparent compression in SSD cache to increase the effective cache size, thus increasing the cache's cost-effectiveness. CacheDedup [13] integrates deduplication and caching to improve the endurance and the effective capacity of SSD cache. It also proposes duplacation-aware cache replacement algorithms D-LRU and D-ARC to further improve the cache hit ratios and reduce cache write traffics. Nitro [40] extends SSD lifetime by combining deduplication and compression and using large replacement units. Our solution is different from these approaches in that KDD reduces cache write traffics by exploiting content locality of the workloads.

Various cache replacement schemes have been proposed to reduce SSD cache writes. Approaches such as SieveStore [41] and LARC [42] propose selective cache allocation policies to only capture the most popular blocks in SSD cache. In this way, they can significantly reduce the allocation writes and avoid the problem of cache pollution. Chai et al. [43] propose Write-Efficient Caching (WEC) to improve the SSD cache's write durability. WEC first identifies write-efficient data that can produce many hits for writing one block. Then it keeps the write-efficient data in SSD cache long enough via pull-mode caching. These approaches are complementary to our KDD in that they can be deployed in KDD to further reduce the amount of writes to SSD.

All the above approaches and our KDD are proposed for off-the-shelf SSDs, however, there are also other hardware/software co-designed flash based caching solutions to improve flash cache endurance. Kgil et al. [44] propose a two-level disk cache composed of a small DRAM and a SLC/MLC dual mode flash. The flash-based cache is split into separate read and write regions to reduce the garbage collection overhead. A programmable flash memory controller is employed to improve flash endurance by changing error code strength or cell density dynamically. OP-FCL [45] balances the read/write cache and the over-provisioned space for efficient use of the flash memory cache. Yang et al. [12] present cache admission/eviction policies and garbage collection policies to reduce erase count of the flash based cache. DuraCache [7] prolongs SSD lifetime by transforming data errors in cache into cache misses and dynamically increasing ECC strength.

An orthogonal family of approaches is those FTL optimizations propsed to enhance the endurance of SSDs at the device level. Examples include changing program and erase voltage [46], reducing the number of flash writes by using deduplication [24] and delta compression [17], [18], reducing block erasures through flash page reuse [47], [48], etc. These FTL designs can also be incorporated into the firmware of SSD cache devices to extend their lifetime.

VI. Conclusion

In this paper, we propose KDD to overcome the small write problem of parity-based RAID storage systems while extending the lifetime of the SSD cache device. KDD reduces cache write traffics by exploiting the strong content locality existing in many data intensive workloads. Cache space is dynamically partitioned for the data and delta zones adapting to different kinds of workloads. In addition, KDD maintains the cache metadata as a persistent log in SSD to reduce the metadata I/Os. To assess the effectiveness of KDD, we conduct extensive trace-driven and benchmark-driven experiments via both simulator and prototype implementations. Simulation results show that KDD significantly reduces the SSD traffics for all workloads. And KDD is more efficient for write-intensive workloads with strong content locality. Evaluations with prototype implementations show that KDD effectively reduces the small write penalty while reducing cache writes.

Acknowledgment

This work was supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301; 863 Project No. 2013AA013203, No. 2015AA015301, No. 2015AA016701; NSFC No. 61173043, No.61303046, No. 61472153; This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," *SIGMOD Rec.*, vol. 17, no. 3, pp. 109–116, Jun. 1988.

[2] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging overcoming the small write problem in redundant disk arrays," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 64–75, May 1993.

[3] Q. Yang and J. Ren, "I-cash: Intelligently coupled array of ssd and hdd," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 278–289.

[4] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 505–519, Jun. 2004.

[5] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12, pp. 267–280.

[6] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," *Trans. Storage*, vol. 6, no. 2, pp. 4:1–4:22, Jul. 2010.

[7] R.-S. Liu, C.-L. Yang, C.-H. Li, and G.-Y. Chen, "Duracache: a durable ssd cache using mlc nand flash," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, pp. 166:1–166:6.

[8] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–12.

[9] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proceedings of the 11th USENIX conference on File and Storage Technologies*, ser. FAST'13.

[10] E. Lee, Y. Oh, and D. Lee, "Ssd caching to overcome small write problem of disk-based raid in enterprise environments," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, pp. 2047–2053.

[11] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12, pp. 2–2.

[12] J. Yang, N. Plasson, G. Gillis, and N. Talagala, "Hec: improving endurance of high performance flash-based cache devices," in *Proceedings of the 6th International Systems and Storage Conference*, ser. SYSTOR '13, pp. 10:1–10:11.

[13] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: In-line deduplication for flash caching," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 301–314.

[14] Q. Yang, W. Xiao, and J. Ren, "Trap-array: A disk array architecture providing timely recovery to any point-in-time," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 289–301, May 2006.

[15] I. Morrey, C.B. and D. Grunwald, "Peabody: the time travelling disk," in *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 241–253.

[16] E. Lee, J.-e. Jang, and H. Bahn, "Dtfs: Exploiting the similarity of data versions to design a write-efficient file system in phase-change memory," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14, pp. 1535–1540.

[17] G. Wu and X. He, "Delta-ftl: Improving ssd lifetime via exploiting content locality," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, pp. 253–266.

[18] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, "Reducing solid-state storage device write stress through opportunistic in-place delta compression," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 111–124.

[19] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[20] Y. Hu and Q. Yang, "Dcddisk caching disk: a new approach for boosting i/o performance," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA '96, pp. 169–178.

[21] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014, pp. 7:1–7:11.

[22] Oltp trace from umass trace repository. http://git.kernel.dk/?p=fio.git.

[23] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.

[24] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11, pp. 6–6.

[25] "Enhanceio," *https://github.com/stec-inc/EnhanceIO*.

[26] M. Oberhumer, "Lzo real-time data compression library," *User manual for LZO version 0.28, URL: http://www. infosys. tuwien. ac. at/Staff/lux/marco/lzo. html (February 1997)*, 2005.

[27] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song, "Pro: A popularity-based multi-threaded reconstruction optimization for raid-structured storage systems," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07, pp. 32–32.

[28] J. Axboe, "Fio - flexible i/o tester synthetic benchmark," http://git.kernel.dk/?p=fio.git.

[29] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "Dfs: A file system for virtualized flash storage," *Trans. Storage*, vol. 6, no. 3, pp. 14:1–14:25, Sep. 2010.

[30] S. Savage and J. Wilkes, "Afraid: A frequently redundant array of independent disks," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '96, pp. 3–3.

[31] S. Mishra and P. Mohapatra, "Performance study of raid-5 disk arrays with data and parity cache," in *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, vol. 1, pp. 222–229 vol.1.

[32] S. Im and D. Shin, "Flash-aware raid techniques for dependable and high-performance flash memory ssd," *Computers, IEEE Transactions on*, vol. 60, no. 1, pp. 80–92, 2011.

[33] C.-C. Chung and H.-H. Hsu, "Partial parity cache and data cache management method to improve the performance of an ssd-based raid," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.

[34] K. Mogi and M. Kitsuregawa, "Hot mirroring: A method of hiding parity update penalty and degradation during rebuilds for raid5," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96, pp. 183–194.

[35] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The hp autoraid hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, Feb. 1996.

[36] K. Mogi and M. Kitsuregawa, "Dynamic parity stripe reorganizations for raid5 disk arrays," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pp. 17–26.

[37] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 451–462.

[38] Y. Oh, E. Lee, C. Hyun, J. Choi, D. Lee, and S. H. Noh, "Enabling cost-effective flash based caching with an array of commodity ssds," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15, pp. 63–74.

[39] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Transparent online storage compression at the block-level," *Trans. Storage*, vol. 8, no. 2, pp. 5:1–5:33, May 2012.

[40] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized ssd cache for primary storage," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, pp. 501–512.

[41] T. Pritchett and M. Thottethodi, "Sievestore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, pp. 163–174.

[42] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng, "Improving flash-based disk cache with lazy adaptive replacement," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pp. 1–10.

[43] Y. Chai, Z. Du, X. Qin, and D. Bader, "Wec: Improving durability of ssd cache drives by caching write-efficient data," *Computers, IEEE Transactions on*, vol. 64, no. 11, pp. 3304–3316, Nov 2015.

[44] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, pp. 327–338.

[45] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, ser. FAST'12, pp. 25–25.

[46] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of nand flash-based storage systems using dynamic program and erase scaling," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14, pp. 61–74.

[47] F. Margaglia and A. Brinkmann, "Improving mlc flash performance and endurance with extended p/e cycles," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–12.

[48] F. Margaglia, G. Yadgar, E. Yaakobi, Y. Li, A. Schuster, and A. Brinkmann, "The devil is in the details: Implementing flash page reuse with wom codes," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 95–109.