Elastic Parity Logging for SSD RAID Arrays: Design, Analysis, and Implementation

Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, Yinlong Xu

Abstract—Parity-based RAID poses a design trade-off issue for large-scale SSD storage systems: it improves reliability against SSD failures through redundancy, yet its parity updates incur extra I/Os and garbage collection operations, thereby degrading the endurance and performance of SSDs. We propose EPLOG, a storage layer that reduces parity traffic to SSDs, so as to provide endurance, reliability, and performance guarantees for SSD RAID arrays. EPLOG mitigates parity update overhead via *elastic parity logging*, which redirects parity traffic to separate log devices (to improve endurance and reliability) and eliminates the need of pre-reading data in parity computations (to improve performance). We design EPLOG as a user-level implementation that is fully compatible with commodity hardware and general erasure coding schemes. We evaluate EPLOG through reliability analysis and trace-driven testbed experiments. Compared to the Linux software RAID implementation, our experimental results show that our EPLOG prototype reduces the total write traffic to SSDs, reduces the number of garbage collection operations, and increases the I/O throughput. In addition, EPLOG significantly improves the I/O performance over the original parity logging design, and incurs low metadata overhead.

Index Terms—SSD, RAID, parity logging

1 INTRODUCTION

Solid-state drives (SSDs) have seen wide adoption in desktops and even large-scale data centers [37], [44], [52]. Today's SSDs mainly build on NAND flash memory. An SSD is composed of multiple flash chips organized in blocks, each containing a fixed number (e.g., 64 to 128) of fixed-size pages of size on the order of KB each (e.g., 2KB, 4KB, or 8KB). Flash memory performs out-of-place writes: each write programs new data in a clean page and marks the page with old data as stale. Clean pages must be reset from stale pages through erase operations performed in units of blocks. To reclaim clean pages, SSDs implement garbage collection (GC), which chooses blocks to erase and relocates any page with data from a to-be-erased block to another block.

Despite the popularity, SSDs still face deployment issues, in terms of reliability, endurance, and performance. First, on the reliability side, bit errors are common in SSDs due to read disturb, write disturb, and data retention [9], [17], [18], [38], and the bit error rate of flash memory generally increases with the number of program/erase (P/E) cycles [17], [29]. Unfortunately, flash-level error correction codes (ECCs) only provide limited protection against bit errors [29], [58], especially in large-scale SSD storage systems. Second, on the endurance side, SSDs have limited lifespans. Each flash memory cell can only sustain a finite number of P/E cycles before wearing out [3], [18], [23]. The sustainable number of P/E cycles is typically 100K for a single-level cell (SLC) and 10K for a multi-level cell (MLC), and further drops to several hundred with a higher flash density [18]. Finally, on the performance side, small random writes are known to degrade the I/O performance of SSDs [10], [25], [39], since they not only aggravate internal fragmentation and trigger more GC operations (which also degrade SSD endurance), but also subvert internal parallelism across flash chips.

Parity-based RAID (Redundant Array of Inexpensive Disks) [48] provides a natural option to enhance the reliability of large-scale storage systems. Its idea is to divide data into groups of fixed-size units called *data chunks*, and each group of data chunks is encoded into redundant information called *parity chunks*. Each group of data and parity chunks, collectively called a *stripe*, provides fault tolerance against the loss of data/parity chunks of the same stripe. Recent studies examine the deployment of SSD RAID at the device level [6], [29], [31], [35], [46], [47], so as to protect against SSD failures.

However, deploying parity-based RAID in SSD storage systems requires special attention [22], [40]. In particular, small random writes are even more harmful to parity-based SSD RAID in both endurance and performance. To maintain stripe consistency, each write to a data chunk triggers updates to all parity chunks of the same stripe. Small writes in RAID imply partial-stripe writes [11], which first read existing data chunks, re-compute new parity chunks, and then write both new data and parity chunks. In the context of SSD RAID, parity updates not only incur extra I/Os (i.e., reads of existing data chunks and writes of parity chunks), but also aggravate GC overhead due to extra parity writes. Frequent parity updates inevitably undermine both endurance and performance of parity-based SSD RAID.

Therefore, parity-based RAID poses a design trade-off issue for large-scale SSD storage systems: it improves reliabil-

• *H. Chan and P. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. (emails: {hwchan,pclee}@cse.cuhk.edu.hk).*
• *Y. Li and Y. Xu are with the School of Computer Science and Technology, University of Science and Technology of China (emails: {ykli,ylxu}@ustc.edu.cn).*
• *An earlier conference version of the paper appeared in the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2016) [32]. In this journal version, we elaborate the parity commit and recovery procedures of EPLOG, extend EPLOG with a key-value store interface, and include substantially more evaluation results.*

ity against SSD failures; on the other hand, its parity updates degrade both endurance and performance. This motivates us to explore a new SSD RAID design that mitigates parity update overhead, so as to provide reliability, endurance, and performance guarantees simultaneously.

We propose EPLOG, an *elastic parity logging* design for SSD RAID arrays. EPLOG builds on parity logging [55] to redirect parity write traffic from SSDs to separate log devices. By reducing parity writes to SSDs, EPLOG slows down the flash wearing rate, and hence improves both reliability and endurance. It further extends the original parity logging design by allowing parity chunks to be computed based on the newly written data chunks only, where the data chunks may span within a partial stripe or across more than one stripe. Such an "elastic" parity construction eliminates the need of pre-reading old data for parity computation, so as to improve performance. To summarize, this paper makes the following contributions.

First, we design EPLOG as a user-level block device[1] that manages an SSD RAID array. Specifically, EPLOG uses hard-disk drives (HDDs) to temporarily log parity information, and regularly commits the latest parity updates to SSDs to mitigate the performance overhead due to HDDs. We show that EPLOG enhances existing flash-aware SSD RAID in different ways: (i) EPLOG is fully compatible with commodity configurations and does not rely on high-cost components such as non-volatile RAM (NVRAM); and (ii) EPLOG can readily support general erasure coding schemes for high fault tolerance.

Second, we implement an EPLOG prototype, and additionally build a key-value store interface atop EPLOG to demonstrate how EPLOG seamlessly supports upper-layer applications. Specifically, we show how our key-value store interface operates on key-value objects and metadata atop EPLOG. The source code of our EPLOG prototype and the key-value store interface is available for download at **http://adslab.cse.cuhk.edu.hk/software/eplog**.

Third, we conduct mathematical analysis on the system reliability in terms of mean-time-to-data-loss (MTTDL). We show that EPLOG improves the system reliability over the conventional RAID design when SSDs and HDDs have comparable failure rates [56] (see our digital supplementary file for details).

Finally, we conduct extensive trace-driven testbed experiments, and demonstrate the endurance and performance gains of EPLOG in mitigating parity update overhead. We compare EPLOG with the Linux software RAID implementation based on `mdadm` [43], which is commonly used for managing software RAID across multiple devices. We show that EPLOG reduces the total write traffic to SSDs by 45.6-54.9%, reduces the number of GC requests by 77.1-97.6%, and increases the I/O throughput by 30.1-119.2% under the (6+2)-RAID-6 setting. Also, EPLOG achieves higher throughput than the original parity logging design, limited metadata management overhead, and maintains performance gains in key-value operations.

The rest of the paper proceeds as follows. Section 2 states our design goals and motivates EPLOG design. Section 3 describes the design and implementation details of EPLOG. Section 4 presents the design of the key-value store interface atop EPLOG. Section 5 presents trace-driven evaluation results of our EPLOG prototype. Section 6 reviews related work, and finally Section 7 concludes the paper. In our digital supplementary file, we also present the caching design and the detailed analysis on the system reliability of EPLOG.

1. Here, a block refers to the read/write unit at the system level, and should not be confused with an SSD block at the flash level.

2 OVERVIEW

2.1 Goals

EPLOG aims for four design goals.

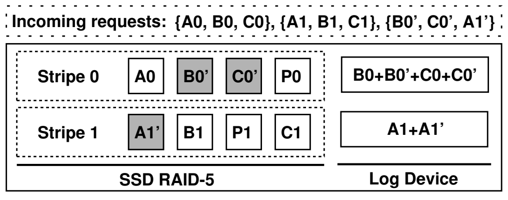
- **General reliability:** EPLOG can tolerate a general number of SSD failures through erasure coding. This differs from many existing SSD RAID designs that are specific for RAID-5 (see Section 6).
- **High endurance:** Since parity updates introduce extra writes to SSDs, EPLOG aims to reduce the parity traffic caused by small (or partial-stripe) writes to SSDs, thereby improving the endurance of SSD RAID.
- **High performance:** EPLOG eliminates the extra I/Os due to parity updates and maintains high read/write performance.
- **Low-cost deployment:** EPLOG is deployable using commodity hardware, and does not assume high-end components such as NVRAM as in previous SSD RAID designs (e.g., [16], [20], [29]).

EPLOG targets workloads that are dominated by small random writes, which lead to frequent partial-stripe writes to RAID. For example, such workloads can be found in database applications [21], [30] and enterprise servers [24]. Note that real-world workloads often exhibit high locality both spatially and temporally [39], [50], [54], such that recently updated chunks and their nearby chunks tend to be updated more frequently. It is thus possible to exploit caching to batch-process chunks in memory to boost both endurance and performance (by reducing write traffic to SSDs). On the other hand, modern storage systems also tend to force synchronous writes through `fsync`/`sync` operations [19], which make small random writes inevitable. Thus, our baseline design should address synchronous small random writes, but allows an optional caching feature for potential performance gains.

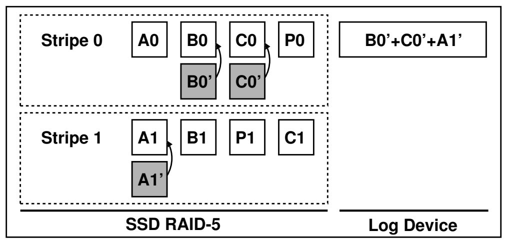
2.2 Parity Logging

Parity logging [55] has been a well-studied solution in traditional RAID to mitigate the parity update overhead. We first review the design of parity logging, and then motivate how we extend its design in the context of SSD RAID.

We demonstrate how parity logging can improve endurance of an SSD RAID array by limiting parity traffic to SSDs. The idea is to add separate *log devices* to keep track of parity information that we refer to as *log chunks*. To illustrate, Figure 1 shows an SSD RAID-5 array with three SSDs for data and one SSD for parity (i.e., the array can tolerate single SSD failure). In addition, one log device is used for storing log chunks. Suppose that a stream of write requests is issued to the array. The first two write requests, respectively with data chunks {`A0`, `B0`, `C0`} and {`A1`, `B1`, `C1`}, constitute two stripes. Also, the following write request



(a) Original parity logging



(b) Elastic parity logging

Fig. 1: Illustration of parity logging schemes in SSD RAID-5.

updates data chunks `B0`, `C0`, and `A1` to `B0'`, `C0'`, and `A1'`, respectively. Figure 1(a) illustrates how the original parity logging works. It first reads and buffers the old data chunks to update (i.e., `B0`, `C0`, and `A1`). It then computes a log chunk by XOR-ing the old and new data chunks on a per-stripe basis. Finally, it updates data chunks *in-place* at the system level above the SSDs (note that an SSD adopts out-of-place updates at the flash level), and appends all log chunks (i.e., `B0+B0'+C0+C0'` and `A1+A1'`) to the log device. Thus, updating the data chunks trigger three read I/Os followed by five write I/Os.

To recover any lost data chunk, parity logging first combines both the log chunks and the existing parity chunks in each stripe to form the up-to-date parity chunks, followed by decoding the lost data chunks. If either parity chunks or log chunks are lost, we directly regenerate them from existing data chunks.

2.3 Elastic Parity Logging

The original parity logging limits parity traffic to SSDs, thereby slowing down their wearing rates. Nevertheless, we identify two constraints of this design. First, it needs to pre-read old data to compute each log chunk, and hence incurs extra read requests. Second, the log chunks are computed on a per-stripe basis. This generates additional log chunks if a write request spans across stripes.

We build on the original parity logging and relax its constraints, and propose a new parity update scheme called *elastic parity logging*. Figure 1(b) illustrates its idea. Specifically, when the write request updates data chunks `B0`, `C0`, and `A1` to `B0'`, `C0'`, and `A1'`, respectively, we perform *out-of-place* updates at the system level, such that we directly write the new data chunks to the corresponding SSDs without overwriting the old data chunks. In other words, both the old and new versions of each data chunk are kept and accessible at the system level. In addition, we compute a log chunk by XOR-ing only the new data chunks to form `B0'+C0'+A1'`, and append it to the log device. Compared to the original parity logging, we now store only one log chunk instead of two. Note that the old versions of data chunks are needed to preserve fault tolerance. For example, if data chunk `A0` is lost, we can recover it from `B0`, `C0`, and `P0`, although both `B0` and `C0` are old versions. Overall, elastic parity logging updates the data chunks with zero read I/O and four write I/Os.

Unlike the original parity logging, elastic parity logging does not need to pre-read old data chunks. It also relaxes the constraint that the log chunks must be computed on a per-stripe basis; instead, a log chunk can be computed from the data chunks within part of a stripe or across more than one stripe (hence we call the parity logging scheme "elastic").

3 Design and Implementation

EPLog is designed as a user-level block device. It runs on top of an SSD RAID array composed of multiple SSDs, and additionally maintains separate log devices for elastic parity logging. In this work, we choose HDDs as log devices to achieve low-cost deployment; while SSDs can also be used as log devices, we do not see significant performance gains as shown in our experiments (see our digital supplementary file). In this section, we address the following design and implementation issues.

- How do we construct log chunks for a write request, such that we maintain reliability as in conventional RAID without using parity logging?
- How do we minimize the access overheads for log chunks in log devices, so as to maintain high performance?
- How do we manage metadata in a persistent manner in our EPLog implementation?

3.1 Architecture

EPLog stores data chunks in a set of SSDs (which we collectively call the *main array*) and log chunks in a set of HDD-based log devices. Accessing log chunks in HDD-based log devices is expensive. Thus, EPLog issues only sequential writes of log chunks to log devices. In addition, it regularly commits the latest parity updates in the main array in the background, such that the main array stores the latest versions of data chunks and the corresponding parity chunks. We call the whole operation *parity commit*. For example, referring to Figure 1(b), we update `P0` and `P1` to reflect the sets of latest data chunks {`A0`, `B0'`, `C0'`} and {`A1'`, `B1`, `C1`}, respectively. Thus, accessing data in degraded mode (i.e., when an SSD fails) can operate in the main array only (as in conventional SSD RAID without parity logging), and hence preserve performance.

EPLog realizes the above design via a modularized architecture, as shown in Figure 2. The *log module* schedules write requests and works with the *coding module* for parity computations. The data chunks are issued to the main array via the *SSD write module*, while the log chunks are issued to the log devices via the *log write module*. To tolerate the same number of device failures, we require the number of log devices in EPLog be equal to the number of tolerable device failures in the main array. For example, if the main array uses RAID-6 (which can tolerate two device failures), two log devices are needed. The *commit module* regularly performs parity commit to ensure that the data and parity chunks in the main array reflect the latest updates.

To further reduce parity traffic, we introduce two types of buffers in the log module, namely a *stripe buffer* and

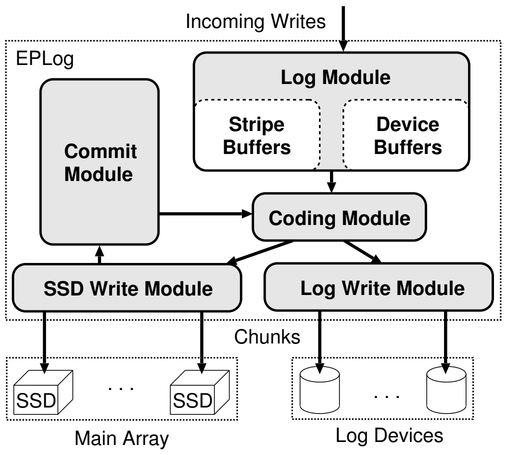


Fig. 2: EPLOG architecture.

multiple *device buffers*, to batch-process write requests in memory. The use of buffers is optional, and does not affect the correctness of our design. In the interest of space, we elaborate the caching design and evaluate its performance in the digital supplementary file.

Limitations: Before presenting the design of EPLOG, we discuss its design limitations. First, EPLOG requires additional storage footprints to keep log chunks, although we employ HDDs as log devices to limit the extra system cost. Second, EPLOG keeps multiple versions of data chunks during updates before parity commit, so we need to provision extra space in SSDs. Third, if a failure happens before parity commit, recovery performance may hurt due to the need of accessing log chunks, especially when we use HDDs as log devices. Finally, parity commit may create additional performance overhead. Our design rationale is that if we perform parity commit regularly on every fixed number of write requests in batch, we can limit parity commit overhead and the drawbacks as described above.

3.2 Write Processing

We describe how EPLOG processes a single write request and constructs log chunks. Note that read requests under no device failures are processed in the same way as in traditional RAID. Thus, we omit the read details.

EPLOG distinguishes (in the log module) write requests into two types. If the incoming write request is a new write and spans a full stripe in the main array, we directly write the data and parity chunks to the main array as in conventional RAID; otherwise, if the request is a new partial-stripe write or an update, then we write data chunks to the main array and the computed parity logs (i.e., log chunks) to log devices. The rationale is that both types of writes do not pre-read data chunks from the main array for parity computation. By issuing new full-stripe writes directly to the main array, we save the subsequent parity commit overhead.

Recall from Section 3.1 that EPLOG first stores data chunks in the main array and log chunks in log devices; after parity commit, it stores both data and parity chunks in the main array. For ease of presentation, we call a stripe that has data chunks stored in the main array and log chunks stored in the log devices a *log stripe*, and call a stripe that has both data and parity chunks stored in the main array a *data stripe*.

Stripe generation: We first explain how we generate a data stripe, followed by how we generate a log stripe. For a data stripe, EPLOG applies (in the coding module) k-of-n *erasure coding* (where $k < n$) to encode the k data chunks into additional $n-k$ parity chunks, such that any k out of n data and parity chunks can reconstruct the data chunks in the data stripe. We configure n to be the number of SSDs in the main array, and configure k such that $n-k$ is the tolerable number of device failures. For example, if we construct an SSD RAID-5 array, we set $n-k=1$; for an SSD RAID-6 array, we set $n-k=2$.

To generate a log stripe, we first require that the data chunks of a log stripe belong to different SSDs. To achieve this, we first identify the destined SSD for each data chunk included in a write request, and then group the data chunks written to different SSDs to form a log stripe. In particular, for a new partial-stripe write, since the data chunks can be written to any SSD, we combine them into a single log stripe and distribute them across SSDs. For an update request, since the destination of each data chunk included in the request is given, if multiple data chunks belong to the same SSD, then we separate them into different log stripes to ensure that each log stripe only contain at most one data chunk belonging to each SSD. We still use the example in Figure 1(b) to illustrate the idea. Since the data chunks `B0`, `C0`, and `A1` belong to different SSDs, we can combine the newly updated data chunks `B0'`, `C0'`, and `A1'` into a single log stripe. We generate only one log chunk `B0'+C0'+A1'` and write it to the log device.

Suppose now that a log stripe contains k' data chunks to be stored in k' different SSDs, where k' is less than or equal to the number of SSDs in the main array. EPLOG then applies k'-of-n' *erasure coding* to generate additional $n'-k'$ log chunks, such that $n'-k'=n-k$, or equivalently, $n'-k'$ equals the tolerable number of device failures. For example, referring to the example in Figure 1, we can group $k'=3$ data chunks {`B0'`, `C0'`, `A1'`} into a log stripe. We then set $n'=4$ and generate $n'-k'=1$ log chunk.

EPLOG can tolerate the same number of device failures (including SSD failures and log device failures) as we deploy conventional RAID directly in the main array. Note that data chunks in EPLOG are now protected by either the parity chunks in the main array or the log chunks in the log devices. Specifically, if a failed data chunk is not updated since the last parity commit, then it can be recovered from other data and parity chunks of the same data stripe in the main array. On the other hand, if a failed data chunk is updated before the next parity commit, then it can be recovered by the log chunks in log devices and other data chunks of the same log stripe. The same argument applies when either a parity chunk or a log chunk fails. In our digital supplementary file, we conduct mathematical analysis to investigate how EPLOG affects the system reliability.

Chunk writes: EPLOG writes both data and parity chunks of a data stripe, as well as the data chunks of a log stripe, to the main array via the SSD write module, while writing the log chunks of a log stripe to the log devices via the log write module. The two modules use different write policies.

Algorithm 1 Parity commit

```
1: Scan the metadata of all log stripes and identify latest versions of
   updated data chunks
2: Read latest versions of updated data chunks from the main array
3: if any data chunk is unavailable then
4:     Read alive data chunks and log chunks from the associated
       log stripe
5:     Recover the unavailable data chunk
6: end if
7: Identify data stripes associated with the updated data chunks
8: for each identified data stripe do
9:     Generate new parity chunks from the latest data chunks of
       the data stripe
10:    Write new parities to SSDs
11:    Update stripe metadata
12: end for
13: Free all old versions of data chunks
14: Free all obsolete log chunks
```

First, the SSD write module uses the *no-overwrite* policy. When it updates a data chunk in an SSD, it writes the new data chunk to a new logical address instead of overwriting the old one, and maintains a pointer to refer to the old data chunk. This makes both the old and new data chunks accessible after the update request. Since the parity chunks in the main array are not yet updated, keeping both the old and new versions of data chunks is necessary to preserve fault tolerance (see Section 2.3 for example). When the parity chunks in the main array are updated after parity commit, the old versions of the data chunks can be removed. On the other hand, the log write module uses the *append-only* policy, so as to ensure sequential writes of log chunks to the log devices and hence preserve performance.

3.3 Parity Commit

EPLOG regularly performs the parity commit operation to ensure all data and parity chunks of data stripes are based on the latest updates. It can trigger parity commit in one of the following scenarios: (i) the system is idle, (ii) there is no available space in any SSD and log device, (iii) an upper-layer application issues a parity commit, and (iv) after every fixed number of write requests.

Algorithm 1 shows the parity commit workflow in EPLOG. First, EPLOG scans the metadata of all log stripes and identifies the latest versions of updated data chunks (line 1). It then reads the latest versions of the updated data chunks from the SSDs of the main array (line 2). If any data chunk is unavailable, EPLOG reconstructs the data chunk by reading the alive data chunks and log chunks from the associated log stripe (lines 3-6). Then EPLOG identifies the data stripes that are associated with the updated data chunks (line 7). For each identified data stripe, EPLOG computes the corresponding parity chunks from the latest data chunks, writes them back to the main array, and updates metadata for stripe maintenance (lines 8-12). Finally, it releases the space occupied by both the old versions of data chunks from the main array and obsolete log chunks from the log devices (lines 13-14).

We emphasize that parity commit does not need to access any log chunks in the log devices in normal mode when there is no SSD failure. The reason is that all latest data chunks, which will be used for computing parities, are kept in SSDs. In case there exist SSD failures during parity commit, we can prefetch log chunks from the log devices in batches, so as to mitigate the access overhead to the log devices. Note that the parity commit operation remains unaffected even if any log device fails.

In addition, although parity commit introduces extra writes to the main array, the write traffic remains limited compared to conventional RAID according to our evaluation results (see Section 5.2). The reason is that we only need to perform parity commit on the latest versions of data chunks in the main array to construct the corresponding parity chunks, while a data chunk may have received multiple updates before parity commit.

We may explore the use of TRIM to explicitly remove the obsolete data chunks during parity commit and further remove GC overhead. On the other hand, the use of TRIM can be tricky and it requires special handling in SSD RAID arrays [22]; for example, we need to ensure the consistency among data and parity chunks of each stripe after removing a chunk from TRIM. Currently, our testbed does not support TRIM under the RAID configuration.

3.4 Recovery

In the presence of SSD failures, EPLOG recovers lost data and parity chunks using erasure coding. EPLOG first performs parity commit to ensure that the data stripes in the main array have the latest data and parity chunks. Then for each data stripe, EPLOG recovers the lost chunks via erasure coding in a new SSD.

To improve the recovery performance, we design optimization techniques that favor parallelism and large I/Os to mitigate recovery overhead. First, EPLOG issues reads and writes across multiple SSDs in parallel via multi-threading. In addition, EPLOG performs recovery of multiple data stripes in batches. Specifically, it performs metadata scans to identify a batch of data stripes to be recovered, where the batch size (i.e., the number of data stripes) is configurable. It reorders and merges any read/write operations to consecutive chunks for each SSD into a single large read/write operation. Finally, it issues (large) reads/writes for the available chunks and recovers the lost chunks via erasure coding.

During the ongoing metadata scans, EPLOG can also prefetch chunks into an in-memory cache pool. Thus, it can directly read any chunks from the cache pool instead of from the main array to further improve recovery performance.

3.5 Implementation Details

We build EPLOG as a user-level block device that is compatible with commodity hardware configurations. We implement the EPLOG prototype in C++ on Linux (with around 7,300 lines of codes). It exports the basic block device interface, which operates on logical addresses on underlying physical devices, as a client API to allow upper-layer applications to access the storage devices. For parity computations, it implements erasure coding based on Cauchy Reed-Solomon codes [7] using the Jerasure 2.0 library [49]. Note that EPLOG issues writes and reads to raw devices via system calls `pwrite` and `pread`, respectively, without using any specific block device driver.

EPLOG is designed to provide persistent metadata management, and it supports two metadata checkpoint operations: *full checkpoint* and *incremental checkpoint*. The full checkpoint flushes all metadata, while the incremental checkpoint flushes any modified metadata since the last full/incremental checkpoint. Both checkpoint operations can be triggered regularly in the background, or by the upper-layer applications.

EPLOG maintains a flat namespace and comprises two types of metadata: *data stripe metadata* and *log stripe metadata*. The data stripe metadata describes the mapping of each data stripe to data chunks, including both the latest and old versions of data chunks. It includes the stripe ID and chunk locations. The log stripe metadata describes the mapping of each log stripe to data chunks, referenced by data stripes, and log chunks on the log devices. It contains stripe ID, number of chunks, and a list of chunk locations.

EPLOG provides persistent metadata storage on SSDs. It creates a separate metadata volume from the main array to keep the metadata checkpoints. The metadata volume comprises three areas: *super block area*, *full checkpoint area*, and *incremental checkpoint area*. The superblock area is located at the front of the metadata partition, and keeps the essential information of the metadata layout. The full checkpoint area follows the super block area, and keeps the full checkpoints. It has two sub-areas [28], which hold the latest and previous full checkpoints. The intuition is to write the full checkpoints alternately to one of the sub-areas, so as to ensure that there always exists a consistent copy of the full checkpoint and hence survive any unexpected system failure during the checkpoint operation. The incremental checkpoint area follows the full checkpoint area. It stores all incremental checkpoints in append-only mode.

To create the metadata volume, we first create two partitions in each SSD in the main array, one for data and another for metadata. We then mount a RAID-10 volume on the metadata partitions of all SSDs using `mdadm` [43], and EPLOG directly accesses the metadata on the volume. In addition, EPLOG directly accesses the data partitions of SSDs and the log devices as raw block devices in JBOD mode. To maintain high I/O performance, EPLOG uses multi-threading to read/write data via the devices in parallel.

EPLOG caches all up-to-date metadata in memory to enable both fast metadata lookups during normal operations and fast metadata scans during parity commit and recovery. It maintains a hash map that indexes both data stripe metadata and log stripe metadata by their stripe IDs. If EPLOG does not store any old version of data chunks (e.g., before receiving any update or right after parity commit), then the cache size is proportional to the number of data stripes that are currently stored. When EPLOG receives updates before issuing a parity commit operation, the cache size further increases with the amount of accumulated update traffic due to the creation of log stripes and the mappings of the old versions of data chunks in data stripe metadata.

4 CASE STUDY: KEY-VALUE STORAGE

To show that EPLOG can seamlessly support upper-layer applications, we design and implement a simple key-value store interface atop the block device interface of EPLOG.

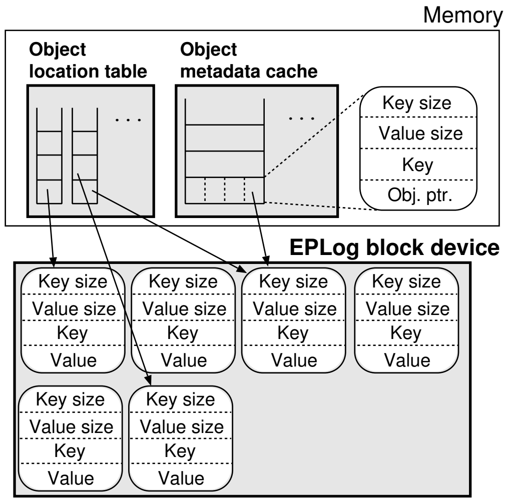


Fig. 3: Key-value store architecture atop EPLOG.

Note that many flash-based key-value stores have been proposed in the literature (e.g., [13], [14], [33], [51]). We do not claim the novelty of the key-value store itself; instead our goal is to demonstrate that the key-value store can easily access data as *key-value pairs* in storage via EPLOG and manage the key-value pairs with limited memory overhead.

Each key-value pair comprises a *key*, which acts as a unique identifier, and a *value*, which contains the actual data content, and *metadata*, which contains the attributes of the key-value pair. Currently, we use 5 bytes of metadata for each key-value pair to represent the key size in 1 byte and the value size in 4 bytes, meaning that the key size and the value size are at most 255 bytes and 2^{32}-1 bytes (almost 4GB), respectively.

Figure 3 shows the architecture of the key-value store atop EPLOG. The key-value store maintains an in-memory *object location table* to store the logical addresses (currently of size 8 byte each) of key-value pairs in EPLOG. We use *cuckoo hashing* [45] to realize the object location table to allow constant-time lookups and updates with high space utilization (up to 90% [15]). Cuckoo hashing maps each inserted key to two possible buckets. If both buckets are not free, it relocates an existing key to make room for the inserted key. Each bucket holds the (8-byte) logical address of the mapped key, and we use logical address to identify the corresponding key to resolve hash collisions. Note that FlashStore [14] also leverages cuckoo hashing (with a different key relocation approach from the original one) to index key-value pairs.

To mitigate the overhead of accessing keys via EPLOG, we implement an optional *object metadata cache* in the key-value store to cache the key, the key and value sizes, and the logical address of a key-value pair that is recently accessed. Like the object location table, we implement the object metadata cache as a cuckoo hashing table. We also implement the cache replacement policy using the ARC algorithm [36], which captures both recency and frequency; note that designing the best cache replacement policies is beyond the scope of this work.

Our key-value store currently supports three key-value operations, namely: (i) `SET`, which inserts new key-value pairs, (ii) `GET`, which retrieves the values of existing key-value pairs, and (iii) `UPDATE`, which updates the values of existing key-value pairs.

Our key-value store implementation itself contains around 500 lines of code. We use the `libcuckoo` library [2] to implement cuckoo hashing. Note that the integration of our key-value store with EPLOG does not change the internals of EPLOG.

5 EXPERIMENTS

We evaluate EPLOG via trace-driven testbed experiments, and compare its endurance and performance with those of the original parity logging and conventional RAID implemented by Linux software RAID based on `mdadm`. Our main findings are: (i) EPLOG improves the endurance of SSD RAID by reducing both the write traffic to SSDs and the number of GC requests, (ii) EPLOG has limited parity commit overhead, (iii) EPLOG achieves higher I/O throughput than baseline approaches, and (iv) our key-value store implementation atop EPLOG preserves the performance gains over baseline approaches.

In our digital supplementary file, we also show that EPLOG achieves potential gains with small-sized caching and has limited overhead on both the storage and metadata management. We evaluate EPLOG on different aspects: (i) the performance of EPLOG using SSDs as log devices, (ii) the trade-off of the object metadata cache in the key-value store implementation, (iii) the individual SSD write sizes, and (iv) the basic I/O performance of EPLOG.

5.1 Setup

Testbed: We conduct our experiments on a machine running Linux Ubuntu 14.04 LTS with kernel 3.13. The machine has a quad-core 3.4GHz Intel Xeon E3-1240v2, 32GB RAM, multiple Plextor M5 Pro 128GB SSDs as the main array, and multiple Seagate ST1000DM003 7200RPM 1TB SATA HDDs as the log devices. It interconnects all SSDs and HDDs via an LSI SAS 9201-16i host bus adapter. Also, we attach an extra SSD to the motherboard as the OS drive. Note that SATA SSDs are still commonly found in modern data centers [42], to which our testbed setup applies.

We compare EPLOG with two baseline parity update schemes. The first one is the Linux software RAID implementation based on `mdadm` (denoted by MD) [43], which implements conventional RAID and writes parity traffic to SSDs directly. The second one is the original parity logging (denoted by PL) [55], which performs parity updates at the stripe level (see Figure 1(a)). We implement PL based on our EPLOG prototype for fair comparisons.

We focus on RAID-5 and RAID-6, which tolerate one and two device failures, respectively. We consider four settings: (4+1)-RAID-5 (i.e., five SSDs), (6+1)-RAID-5 (i.e., seven SSDs), (4+2)-RAID-6 (i.e., six SSDs), and (6+2)-RAID-6 (i.e., eight SSDs). For PL and EPLOG, we allocate one and two additional HDDs as log devices for RAID-5 and RAID-6, respectively. In all schemes, we set the chunk size as 4KB. We use the `O_DIRECT` mode to bypass the internal cache. For PL and EPLOG, we disable caching, parity commit, and metadata checkpointing (i.e., the metadata structure remains in memory), except when we evaluate these features.

	Plextor SSD	Seagate HDD
Sequential write (MB/s)	93.4	91.6
Random write (MB/s)	93.1	1.1
Sequential read (MB/s)	99.23	94.2
Random read (MB/s)	32.8	0.6

TABLE 1: Device performance under 4KB direct I/Os.

	No. of writes	**Avg. write size (KB)**	**Random write (%)**	**WSS (GB)**
FIN	4,110,563	7.19	76.17	3.67
WEB	1,431,628	12.50	77.62	7.26
USR	1,363,855	10.05	76.19	2.44
MDS	1,069,421	7.22	82.99	3.09

TABLE 2: Trace statistics: total number of writes, average write size, ratio of random writes, and working set size.

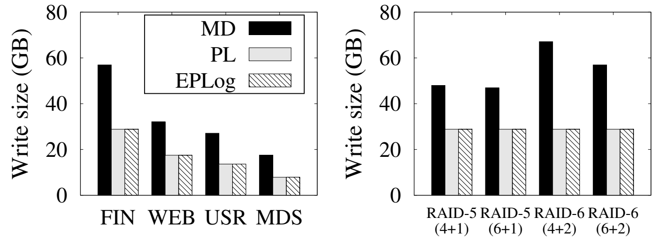
Preliminary benchmarks: Before we start our evaluation, we first measure the raw device performance of the SSDs used in the main array and the HDDs used as log devices in our testbed. We issue 4KB requests, with a total of 2GB data, in `O_DIRECT` mode using `fio` [5] to each type of devices and measure its performance. Table 1 shows the throughput of the two types of devices under both sequential and random I/Os. Although HDDs have very low random read/write performance, its sequential write performance is only slightly lower than the random write performance of SSDs by 1.5 MB/s (or 1.6%). Thus, we expect that the HDD-based log devices will not be the major bottleneck in small-write dominant workloads targeted by EPLOG (see Section 2.1). Note that the log-structured design of SSDs can transform random writes into sequential ones, so random writes can outperform random reads in SSDs [4, Chapter "Flash-based SSDs"].

Traces: We consider four real-world I/O traces to replay in our experiments:

- `FIN`: It is an I/O trace collected by the Storage Performance Council [1]. The trace captures the workloads of a financial OLTP application over a 12-hour period. We choose the write-dominant trace file `Financial1.spc` out of the two available traces.
- `WEB`, `USR`, and `MDS`: They are three I/O traces collected by Microsoft Research Cambridge [41]. They describe the workloads of enterprise servers of three volumes, `web0`, `usr0`, and `mds0`, respectively, over a one-week period.

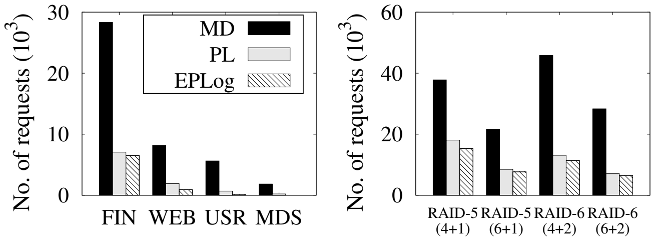
Note that each of the original traces spans a very large address space, yet only a small proportion of the addresses are actually accessed. To fit the traces into our testbed, which has a limited storage capacity, we compact each trace by skipping the addresses that are not accessed. Specifically, we divide the whole logical address space of each trace into 1MB segments. We then skip any segment that is not accessed, and also shift the offsets of the requests in the following accessed segments accordingly. We keep the same request order, so as to preserve workload locality.

Before the trace replay for each trace, we first sequentially write to all remaining segments (after we compact the traces) to fully occupy the working set. Each write request in a trace will be treated as an update. In addition, we round up the size of each write request to the nearest multiple of the chunk size. By making all write requests as updates, we can stress-test the impact of parity updates.



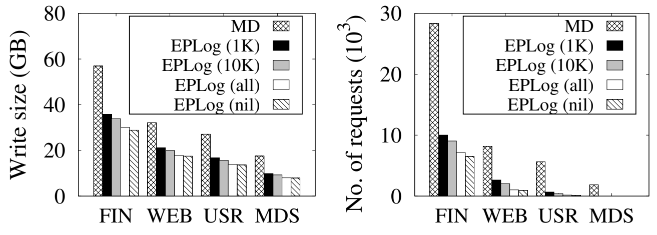
(a) Different traces under (6+2)-RAID-6 (b) Different RAID settings under FIN

Fig. 4: Experiment 1: Total size of write traffic to SSDs.



(a) Different traces under (6+2)-RAID-6 (b) Different RAID settings under FIN

Fig. 5: Experiment 2: GC overhead, measured in the average number of GC requests to each SSD.



(a) Total write size to SSDs (b) GC overhead

Fig. 6: Experiment 3: Total size of write traffic to SSDs and GC overhead under different parity commit cases and (6+2)-RAID-6 setting.

Table 2 summarizes the write statistics of the four traces, after we round up the sizes of all write requests. It shows a few key properties. First, their average write sizes are generally small (7-13KB). Second, if we examine the access pattern, we see that all traces have a high proportion of random write requests. Here, by a random write request, we mean that a write request whose starting offset differs from the ending offset of the last write request by at least 64KB. Finally, if we examine the working set size (i.e., the size of unique data accessed throughout the trace duration), all traces have small working set sizes.

5.2 Results

Experiment 1 (Write traffic to SSDs): We first show the effectiveness of EPLOG in reducing write traffic to SSDs due to parity updates, given that our traces are dominated by small random writes. Figure 4(a) shows the total size of write traffic to SSDs across different traces under (6+2)-RAID-6. Overall, EPLOG achieves a 45.6-54.9% reduction in write size when compared to MD. Both PL and EPLOG have the same results, since they write the same amount of data updates to SSDs (see Figure 1), while redirecting parity traffic to log devices. We emphasize that even though EPLOG has the same write traffic to SSDs with PL, it achieves much higher I/O throughput than PL due to elastic parity logging (see Experiment 4). Figure 4(b) shows the reduction of write traffic to SSDs across four different RAID settings under the FIN trace (which has the most write requests among all traces). EPLOG reduces 38.6-39.9% and 49.3-57.0% of write traffic over MD for RAID-5 and RAID-6, respectively. Note that RAID-6 shows more significant reduction of write traffic than RAID-5.

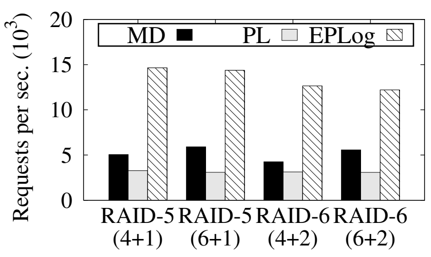
Experiment 2 (GC overhead): We study the endurance in terms of GC overhead, which we measure by the average number of GC requests to each SSD. Since SSD controllers do not expose GC information, we resort to trace-driven simulations using Microsoft's SSD simulator [3] that builds on Disksim [8]. For the simulator, we configure each SSD with 20GB raw capacity and 16,384 blocks with 64 4KB pages each (i.e., 256KB per block). Also, based on the default simulator settings, each SSD over-provisions 15% of blocks for GC (i.e., the effective capacity of each SSD is 17GB) and triggers GC when the number of clean blocks drops below 5%. We use the default greedy algorithm in the simulator and disable the wear-leveling block migration.

We replay the traces and use the `blktrace` utility to capture block-level I/O requests for each SSD in the background. Then we feed the block I/O requests into the simulator. We measure the total number of GC requests per SSD, and take an average over all SSDs in the main array.

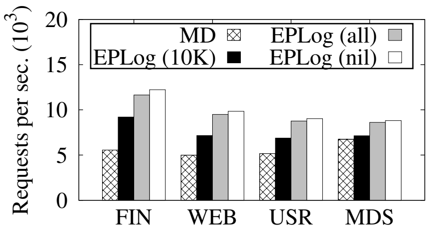
Figure 5 plots the total number of GC requests per SSD, averaged over all SSDs. Figure 5(a) shows the results across different traces under (6+2)-RAID-6. EPLOG significantly reduces the number of GC requests over MD by 77.1-97.6%. This implies EPLOG significantly improves endurance. We also note that EPLOG reduces at least 8.1% of GC requests over PL in all traces. The reason is that EPLOG updates data chunks by using the no-overwrite updating policy, which reserves part of the logical address space for data updates. Thus, EPLOG introduces higher sequentiality for writes to SSDs. Note that EPLOG triggers no GC under MDS and (6+2)-RAID-6, since it reduces the amount of writes to each SSD and does not cause the number of clean blocks to drop below the threshold. Also, Figure 5(b) shows that EPLOG reduces 59.6-77.1% of GC requests over MD across different RAID settings under the FIN trace.

Experiment 3 (Parity commit overhead): Parity commit introduces additional writes (see Section 3.3). We study the impact of parity commit on endurance. We consider the following cases of parity commit: (i) without any parity commit (labeled as "nil"), (ii) commit only at the end of trace replay (labeled as "all"), and (iii) commit every 1,000 or 10,000 write requests (labeled as "1K" and "10K", respectively). We also include the results of MD from Experiment 1 for comparison.

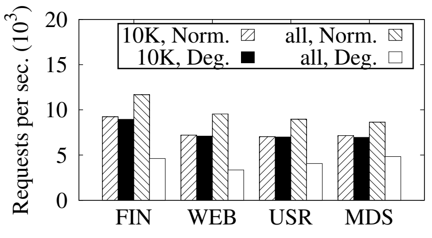
Figure 6 shows the parity commit overhead for different traces under (6+2)-RAID-6. Figure 6(a) shows the total size of write traffic to SSDs (as in Experiment 1). Compared to the case without any parity commit, the write size increases by up to 4.3%, 17.4%, and 24.9% when we perform par-



(a) Different RAID settings under `FIN`



(b) Different traces under (6+2)-RAID-6 without SSD failure



(c) Different traces under (6+2)-RAID-6 with double-SSD failure

Fig. 7: Experiment 4: I/O performance comparisons under real-world traces.

ity commit at the end of trace replay, every 10,000 write requests, and every 1,000 write requests, respectively. The write size with parity commit is still less than MD (e.g., by over 40% in some cases). Figure 6(b) plots the average number of GC requests to each SSD (as in Experiment 2). The number of GC requests of EPLOG is 74.8-97.1%, 68.1-93.0%, and 67.8-88.2% less than that of MD when we perform parity commit at the end of trace replay, every 10,000 write requests, and every 1,000 write requests, respectively. The results show that the parity commit overhead in write size remains limited if we perform parity commit in groups of writes.

Experiment 4 (I/O performance): We examine the I/O performance of EPLOG via trace replay of the real-world I/O traces in Table 2. We replay each trace as fast as possible to obtain the maximum possible performance. We measure the performance by the number of requests issued to MD, PL, and EPLOG per unit time (in requests per second). Note that in PL and EPLOG, the measured performance includes the overhead of writes to the HDD-based log devices.

Figure 7(a) shows the throughput across different RAID settings under the `FIN` trace without parity commit and without any SSD failure. EPLOG outperforms MD and PL by 119.2-197.3% and 295.7-366.1%, respectively. Both MD and PL read data before updating or logging parity on the update path. MD is faster than PL as MD directly updates parities on SSDs, while PL logs parity updates to HDD-based log devices for endurance. While the performance gap between random writes on SSDs and sequential write on HDDs is small (see Preliminary benchmarks in Section 5.1), PL writes log chunks to log devices for multiple stripes under cross-stripe updates, which increases the performance overhead of logging in PL. EPLOG eliminates pre-reads of existing data in log chunk computation, thereby increasing the I/O throughput. In addition, EPLOG reduces the total size of log chunks by 8-15% compared to PL (not shown in the figure) due to elastic parity logging, which also contributes to the throughput gains.

Figure 7(b) shows the throughput across different traces under (6+2)-RAID-6 when parity commit is enabled, while there is no SSD failure. First, if there is no parity commit (labeled as "nil"), EPLOG's throughput is higher than MD's by 30.1-119.2% and PL's by 186.9-305.5% across different traces. If we perform parity commit at the end of the trace (labeled as "all"), EPLOG's throughput only slightly decreases by at most 4.79%, compared to that without parity commit. Even if we commit every 10,000 write requests (labeled as "10K"),

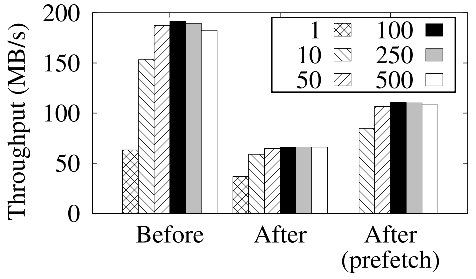


Fig. 8: Experiment 5: Recovery throughput of EPLOG under `FIN` and (6+2)-RAID-6 for different batch sizes. Recovery is performed (i) before trace replay, (ii) after trace replay, and (iii) after trace replay with chunk prefetching.

EPLOG's throughput drops by 23.8% on average, but it is still higher than MD's by 37.2% on average across traces.

We now compare the parity commit overhead in normal mode (i.e., no SSD failure) and in degraded mode (i.e., in the presence of SSD failures). Figure 7(c) shows the throughput of EPLOG across different traces under (6+2)-RAID-6 when it performs parity commit in normal and degraded modes. To activate degraded mode, we bring down two SSDs (i.e., a double-SSD failure) at the end of trace replay but before the last parity commit operation is performed. Thus, our results reflect the impact of a single degraded parity commit operation on the overall performance. We observe from the figure that the performance drop from normal mode to degraded mode depends on the parity commit frequency. When EPLOG performs parity commit every 10,000 requests, its throughput drops by up to 3.03% only from normal mode (labeled as "10K, Norm.") to degraded mode (labeled as "10K, Deg."). On the other hand, when EPLOG performs parity commit at the end of trace replay, its throughput drops by up to 64.8% (for the `WEB` trace) from normal mode (labeled as "all, Norm.") to degraded mode (labeled as "all, Deg.").

The results in Figure 7(c) illustrate the trade-off between normal and degraded performance at different parity commit frequencies. If we perform parity commit more frequently, EPLOG has a higher throughput drop in normal mode, but has a lower throughput drop in degraded mode as it needs to access fewer log chunks from the log devices. The results also show the trade-off between endurance and performance, as lowering the parity commit frequency improves endurance (see Experiment 3).

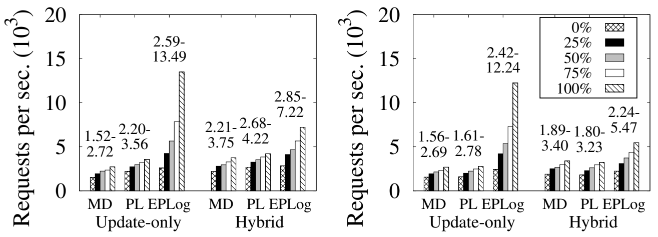
Experiment 5 (Recovery performance): We evaluate the recovery performance of EPLOG under failures. We focus on the (6+2)-RAID-6 setting under the FIN trace. We consider two cases of recovery: (i) before trace replay (i.e., chunks are sequentially stored on SSDs) and (ii) after trace replay (i.e., the up-to-date chunks are stored on SSDs). We assume that parity commit has been performed in both cases, and then we erase the data of two SSDs to simulate a double-SSD failure. We recover the lost data on the same two SSDs and measure the *recovery throughput* as the amount of recovered data over the total recovery time.

We also study the performance gains of our optimization techniques (see Section 3.4). We study the impact of batched recovery versus the batch size (i.e., the number of data stripes to be recovered in each batched). We also consider the impact of chunk prefetching during metadata scans.

Figure 8 shows the recovery performance of EPLOG. Before trace replay, the recovery throughput is 60.3MB/s and 182.9MB/s when EPLOG recovers lost data on a per-stripe basis (i.e., batch size is one) and in batches of 100 data stripes, respectively. This shows that batched recovery improves recovery performance. Note that batching 100 data stripes only incurs around 3MB of memory for buffering (i.e., 4KB chunk size $\times$ 8 chunks per stripe $\times$ 100). The performance becomes stable when the batch size is 100 or above. After trace replay, the recovery throughput is 35.0MB/s and 62.9MB/s when EPLOG recovers lost data on a per-stripe basis and in batches of 100 stripes, respectively. Since EPLOG adopts the no-overwrite policy for data updates, the latest data chunks are no longer sequentially stored and EPLOG issues random reads to alive chunks for recovery. Nevertheless, chunk prefetching can improve the recovery performance for the after-trace-replay case by at least 43.9%.

Experiment 6 (Key-value store performance): We evaluate our key-value store interface atop EPLOG. We also integrate our key-value store interface with PL and MD, and evaluate them for comparisons. We use YCSB [12] to generate two types of key-value store workloads: (i) *update-only*, which comprises UPDATE requests only, and (ii) *hybrid*, which comprises 50% of UPDATE requests and 50% of GET requests. Before running each workload, we first load 1 million key-value pairs via SET into storage. For each workload, we then issue 3 million requests over the 1 million key-value pairs, such that the access pattern follows a heavy-tailed Zipf distribution with the shape parameter 0.99. For each key-value pair, we fix its key size as 24 bytes. Also, by varying the value size, we configure the total size of a key-value pair (including the metadata, key, and value) as 4KB and 8KB, both of which align with the 4KB chunk size. To evaluate the impact of the object metadata cache, we configure its size as 0%, 25%, 50%, 75%, and 100% of the total number of key-value pairs. We focus on the (6+2)-RAID-6 setting, and disable parity commit. Figure 9 shows the throughput results, and we make the following observations.

First, EPLOG still achieves higher throughput than MD and PL, conforming to the results in Experiment 4 when they run without the key-value store. For example, for 4KB key-value pairs, EPLOG outperforms MD and PL by 70.7-395.5% and 17.6-279.3%, respectively, in the update-only workload, and by 28.8-92.2% and 6.4-71.2%, respectively, in the hybrid workload.



(a) 4KB key-value pairs (b) 8KB key-value pairs

Fig. 9: Experiment 6: Key-value store performance under (6+2)-RAID-6 and different cache settings.

In most cases, EPLOG achieves higher throughput in the update-only workload than in the hybrid workload, by up to 86.9% and up to 123.8% for 4KB and 8KB key-value pairs, respectively. In EPLOG, the update-only workload (with UPDATE requests only) contains only random writes without pre-reading old chunks for parity updates, while the hybrid workload contains a mix of random reads and random writes and the random write performance is better than the random read performance (see Preliminary benchmarks in Section 5.1). On the other hand, both PL and MD achieve higher throughput in the hybrid workload than in the update-only workload. Recall that they both pre-read chunks for parity updates in UPDATE requests. Thus, the overhead of UPDATE is higher than that of GET, making the update-only workload more expensive.

In some cases, PL achieves higher throughput than MD, for example, by 30.6-45.2% and 12.3-21.1% in the update-only and hybrid workloads for the 4KB key-value pairs, respectively. Such results are opposite to the ones in Experiment 4 (see Figure 7(a)). One possible reason is that our workloads for 4KB key-value pairs here do not contain cross-stripe requests, as opposed to the FIN trace in Experiment 4. This limits the parity logging overhead in PL, making PL faster than MD.

As expected, increasing the size of the object metadata cache significantly improves the throughput, as the overhead of accessing the keys in storage for resolving hash collisions decreases. The performance gain with the object metadata cache in particular is more significant in EPLOG. For example, when the object metadata cache size increases from 0% to 100%, EPLOG's throughput increases by 420.9% and 153.4% in the update-only and hybrid workloads for 4KB key-value pairs, respectively.

6 RELATED WORK

Researchers have proposed various techniques for enhancing the performance and endurance of a single SSD, such as disk-based write caching [54], read/write separation via redundancy [53], and flash-aware file systems (e.g., [27], [28], [34], [39]). EPLOG targets an SSD RAID array and is currently implemented as a user-level block device. It can also incorporate advanced techniques of existing flash-aware designs, such as hot/cold data grouping [28], [39] and efficient metadata management [27], [34], for further performance and endurance improvements.

Flash-aware RAID designs have been proposed either at the chip level [16], [20], [26] or at the device level [6], [29], [31], [35], [46], [47]. For example, Greenan *et al.* [16]

keep outstanding parity updates in NVRAM and defer them until a full stripe of data is available. FRA [31] also defers parity updates, but keeps outstanding parity updates in DRAM, which is susceptible to data loss. Balakrishnan *et al.* [6] propose to unevenly distribute parities among SSDs to avoid correlated failures. Lee *et al.* [29] and Im *et al.* [20] propose the partial parity idea, which generates parity chunks from partial stripes and maintains the parity chunks in NVRAM. HPDA [35] builds an SSD-HDD hybrid architecture which keeps all parities in HDDs and uses the HDDs as write buffers. Kim *et al.* [26] propose an elastic striping method that encodes the newly written data to form new data stripes and writes the data and parity chunks directly to SSDs without NVRAM. Pan *et al.* [47] propose a diagonal coding scheme to address the system-level wear-leveling problem in SSD RAID, and the same research group [46] extends the elastic striping method by Kim *et al.* [26] with a hotness-aware design. LDM [57] stores both data and parities in SSDs in a RAID-5 setting, while it buffers writes in mirrored HDDs as in HPDA [35].

EPLOG relaxes the constraints of parity construction in which parity can be associated with a partial stripe, following the same rationale as previous work [20], [26], [29], [46]. Compared to previous work, EPLOG keeps log chunks with elastic parity logging using commodity HDDs rather than NVRAM as in [20], [29]. Also, instead of directly writing parity chunks to SSDs [26], [46], EPLOG keeps log chunks in log devices to limit parity write traffic to SSDs, especially when synchronous writes are needed (see Section 2.1). While HPDA [35] also uses HDDs to keep parities as in EPLOG, it always keeps all parities in HDDs and treats HDDs as a write buffer, but does not explain how parities in HDDs are generated and stored. In contrast, EPLOG ensures sequential writes of log chunks to HDD-based log devices and regularly performs parity commit in SSDs (note that parity commit does not need to access log devices in normal mode). In addition, EPLOG employs an elastic logging policy, which does not need to pre-read old data chunks and also relaxes the constraint of per-stripe basis in computing parity logs, so as to reduce the amount of logs and fully utilize device-level parallelism among SSDs. We point out that EPLOG targets general RAID schemes that tolerate a general number of failures, as opposed to single fault tolerance as assumed in most existing approaches discussed above.

7 Conclusions

We present EPLOG, a user-level block device that mitigates parity update overhead in SSD RAID arrays through elastic parity logging. It encodes new data chunks to form log chunks and appends the log chunks into separate log devices, while the data chunks may span in a partial stripe or across more than one stripe. We carefully build our EPLOG prototype and its integration with a key-value store interface on commodity hardware. Our evaluation shows that EPLOG improves reliability, endurance, and performance.

Acknowledgments

The work was supported in part by National Nature Science Foundation of China (61772484), and Anhui Provincial Natural Science Foundation (1508085SQF214).

References

[1] Storage Performance Council. http://traces.cs.umass.edu/index.php/Storage/Storage, 2002.

[2] libcuckoo. https://github.com/efficient/libcuckoo, 2013.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC*, 2008.

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[5] J. Axboe. Flexible I/O Tester. https://github.com/axboe/fio, 2005.

[6] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential RAID: Rethinking RAID for SSD Reliability. *ACM Trans. on Storage*, 6(2):4:1–4:22, Jul 2010.

[7] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug. 1995.

[8] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual, 2008.

[9] Y. Cai, E. Haratsch, O. Mutlu, and K. Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *Prof. of DATE*, 2012.

[10] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proc. of ACM SIGMETRICS*, 2009.

[11] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[13] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. of USENIX ATC*, 2010.

[14] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. of VLDB Endowment*, 3(1-2):1414–1425, Sept. 2010.

[15] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*, 2013.

[16] K. Greenan, D. D. E. Long, E. L. Miller, T. Schwarz, and A. Wildani. Building Flexible, Fault-Tolerant Flash-based Storage Systems. In *Proc. of USENIX HotDep*, 2009.

[17] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proc. of IEEE/ACM MICRO*, 2009.

[18] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *Proc. of USENIX FAST*, 2012.

[19] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proc. of ACM SOSP*, 2011.

[20] S. Im and D. Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. *IEEE Trans. on Computers*, 60(1):80–92, Jan 2011.

[21] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC*, 2013.

[22] N. Jeremic, G. Mühl, A. Busse, and J. Richling. The Pitfalls of Deploying Solid-state Drive RAIDs. In *Proc. of ACM SYSTOR*, 2011.

[23] M. Jung and M. Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proc. of ACM SIGMETRICS*, 2013.

[24] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *Proc. of IEEE IISWC*, 2008.

[25] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proc. of USENIX FAST*, 2008.

[26] J. Kim, J. Lee, J. Choi, D. Lee, and S. Noh. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. In *Proc. of IEEE/IFIP DSN*, 2013.

[27] J. Kim, H. Shim, S.-Y. Park, S. Maeng, and J.-S. Kim. FlashLight: A Lightweight Flash File System for Embedded Systems. *ACM Trans. on Embedded Computing Systems*, 11S(1):18:1–18:23, June 2012.

[28] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST*, 2015.

[29] S. Lee, B. Lee, K. Koh, and H. Bahn. A Lifespan-aware Reliability Scheme for RAID-based Flash Storage. In *Proc. of ACM SAC*, 2011.

[30] S.-W. Lee and B. Moon. Design of Flash-based DBMS: An In-page Logging Approach. In *Proc. of ACM SIGMOD*, 2007.

[31] Y. Lee, S. Jung, and Y. H. Song. FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In *Proc. of IEEE/ACM CODES+ISSS*, 2009.

[32] Y. Li, H. H. W. Chan, P. P. C. Lee, and Y. Xu. Elastic Parity Logging for SSD RAID Arrays. In *Proc. of IEEE/IFIP DSN*, 2016.

[33] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. on Storage*, 13(1):5:1–5:28, Mar. 2017.

[34] Y. Lu, J. Shu, and W. Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proc. of USENIX FAST*, 2014.

[35] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, and L. Zeng. HPDA: A Hybrid Parity-based Disk Array for Enhanced Performance and Reliability. *ACM Trans. on Storage*, 8(1):4:1–4:20, Feb 2012.

[36] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of USENIX FAST*, 2003.

[37] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proc. of ACM SIGMETRICS*, 2015.

[38] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit Error Rate in NAND Flash Memories. In *Proc. of IEEE IRPS*, 2008.

[39] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proc. of USENIX FAST*, 2010.

[40] S. Moon and A. L. N. Reddy. Don't Let RAID Raid the Lifetime of Your SSD Array. In *Proc. of USENIX HotStorage*, 2013.

[41] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM Trans. on Storage*, 4(3):10:1–10:23, Nov. 2008.

[42] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid. Ssd failures in datacenters: What? when? and why? In *Proc. of ACM SYSTOR*, 2016.

[43] J. Ostergaard and E. Bueso. The Software-RAID HOWTO. http://tldp.org/HOWTO/html_single/Software-RAID-HOWTO.

[44] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proc. of ACM ASPLOS*, 2014.

[45] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[46] Y. Pan, Y. Li, Y. Xu, and Z. Li. Grouping-Based Elastic Striping with Hotness Awareness for Improving SSD RAID Performance. In *Proc. of IEEE/IFIP DSN*, 2015.

[47] Y. Pan, Y. Li, Y. Xu, and W. Zhang. DCS5: Diagonal Coding Scheme for Enhancing the Endurance of SSD-Based RAID-5 Systems. In *Proc. of IEEE NAS*, 2014.

[48] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of ACM SIGMOD*, 1988.

[49] J. S. Plank and K. M. Greenan. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications. Technical Report UT-EECS-14-721, University of Tennessee, EECS Department, Jan 2014.

[50] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *USENIX Winter 1993 Technical Conference*, 1993.

[51] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didacache: A deep integration of device and application for flash based key-value caching. In *Proc. of USENIX FAST*, 2017.

[52] R. S. Sinkovits, P. Cicotti, S. Strande, M. Tatineni, P. Rodriguez, N. Wolter, and N. Balac. Data Intensive Analysis on the Gordon High Performance Data and Compute System. In *Proc. of ACM KDD*, 2011.

[53] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proc. of USENIX ATC*, 2014.

[54] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proc. of USENIX FAST*, 2010.

[55] D. Stodolsky, G. Gibson, and M. Holland. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of ISCA*, 1993.

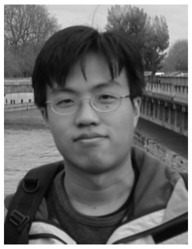
[56] K. Thomas. Solid State Drives No Better Than Others, Survey Says. www.pcworld.com/article/213442.

[57] S. Wu, B. Mao, X. Chen, and H. Jiang. LDM: Log Disk Mirroring with Improved Performance and Reliability for SSD-Based Disk Arrays. *ACM Trans. on Storage*, 12(4):22:1–22:21, May 2016.

[58] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the Robustness of SSDs under Power Fault. In *Proc. of USENIX FAST*, 2013.

Helen H. W. Chan received the B.Eng. degree in Computer Engineering from The Chinese University of Hong Kong in 2013. She is now pursuing her Ph.D. degree in Computer Science and Engineering at The Chinese University of Hong Kong. Her research interests include storage reliability and flash-based storage.

Yongkun Li is currently an associate professor in School of Computer Science and Technology, University of Science and Technology of China. He received the B.Eng. degree in Computer Science from University of Science and Technology of China in 2008, and the Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong in 2012. His research mainly focuses on data-intensive computing systems, with emphasis on file systems and memory systems.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security.

Yinlong Xu received the Bachelor's degree in mathematics from Peking University in 1983, and the master and PhD degrees in computer science from University of Science and Technology of China (USTC) in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology at USTC. His research interests include network coding, wireless network, combinatorial optimization, design and analysis of parallel algorithm, parallel programming tools, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences in 2006.