Elastic Parity Logging for SSD RAID Arrays

Yongkun Li[1,2], Helen H. W. Chan[3], Patrick P. C. Lee[3], Yinlong Xu[1,4]
[1]School of Computer Science and Technology, University of Science and Technology of China
[2]Collaborative Innovation Center of High Performance Computing, National University of Defense Technology
[3]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[4]AnHui Province Key Laboratory of High Performance Computing
{*ykli,ylxu*}*@ustc.edu.cn,* {*hwchan,pclee*}*@cse.cuhk.edu.hk*

***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.**

I. Introduction

A. Background

Solid-state drives (SSDs) have seen wide adoption in desktops and even large-scale data centers [32], [38], [44]. 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 [6], [12], [13], [33], and the bit error rate of flash memory generally increases with the number of program/erase (P/E) cycles [12], [26]. Unfortunately, flash-level error correction codes (ECCs) only provide limited protection against bit errors [26], [49], 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 [2], [13], [19]. 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 [13]. Finally, on the performance side, small random writes are known to degrade the I/O performance of SSDs [7], [21], [34], since they not only aggravate internal fragmentation and trigger more GC operations (which also degrade the endurance of SSDs), but also subvert internal parallelism across flash chips.

Parity-based RAID (Redundant Array of Inexpensive Disks) [41] 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 chunks, such that any subset of a sufficient number of data/parity chunks of the same stripe can reconstruct the original data chunks. Recent studies examine the deployment of SSD RAID at the device level [3], [26], [28], [31], [39], [40], so as to protect against SSD failures.

However, deploying parity-based RAID in SSD storage systems requires special attention [18], [35]. 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 [8], 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 overheads due to extra parity writes. Frequent parity updates inevitably undermine both endurance and performance of parity-based SSD RAID, especially when we need a higher degree of fault tolerance (i.e., more parity updates).

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

B. Contributions

We propose EPLOG, an *elastic parity logging* design for SSD RAID arrays. EPLOG builds on parity logging [47] 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:

- We design and implement 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 (see Section VI) 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.
- 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 [48].
- We conduct extensive trace-driven testbed experiments, and demonstrate the endurance and performance gains of EPLOG in mitigating parity update overheads. We compare EPLOG with the Linux software RAID implementation based on `mdadm` [37], which is commonly used for managing software RAID across multiple devices. For example, in some settings, 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% even though it uses HDDs for parity logging. Finally, EPLOG shows higher throughput than the original parity logging design, and incurs low overhead in metadata management.

The rest of the paper proceeds as follows. In Section II, we state our design goals and motivate our new elastic parity logging design. In Section III, we describe the design and implementation details of EPLOG. In Section IV, we analyze the system reliability of EPLOG. In Section V, we present evaluation results on our EPLOG prototype through trace-driven testbed experiments. In Section VI, we review related work, and finally in Section VII, we conclude the paper.

II. OVERVIEW

In this section, we state the design goals of EPLOG. We also motivate how EPLOG mitigates parity update overhead through elastic parity logging.

A. Goals

EPLOG aims for four design goals.

- **General reliability:** EPLOG provides fault tolerance against SSD failures. In particular, it 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 VI).
- **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, thereby maintaining 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 SSD RAID designs (e.g., [10], [15], [26]).

EPLOG targets workloads that are dominated by small random writes, leading to frequent partial-stripe writes to RAID. Examples of such workloads include those in database applications [17], [27] and enterprise servers [20]. Note that real-world workloads often exhibit high locality both spatially and temporally [34], [43], [46], 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 [14], 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.

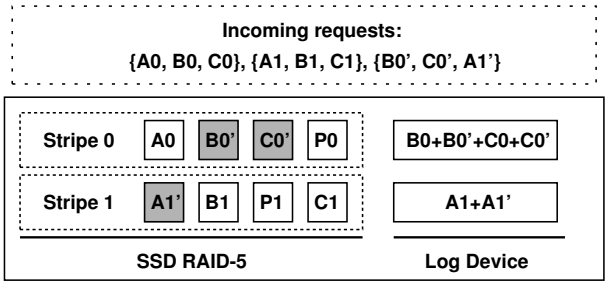
B. Elastic Parity Logging

Parity logging [47] 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.

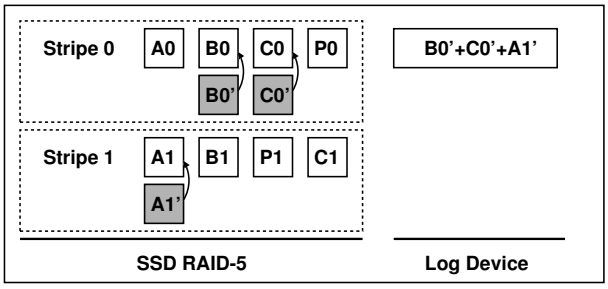
We first demonstrate how parity logging can improve endurance of an SSD RAID array by limiting parity traffic to SSDs. Our idea is to add separate *log devices* to keep track of parity information that we refer to as *log chunks*. To illustrate, Figure 1 shows an SSD RAID-5 array with three SSDs for data and one SSD for parity (i.e., the array can tolerate single SSD failure). In addition, we have one log device 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 updates data chunks `B0`, `C0`, and `A1` to `B0′`, `C0′`, and `A1′`, respectively. Figure 1(a) illustrates how the original parity logging works. 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 as described in Section I-A). It computes a log chunk by XOR-ing the old and new data chunks on a per-stripe basis. It then appends all log chunks to the log device.

The original parity logging limits parity traffic to SSDs, thereby slowing down their wearing rates. Nevertheless, we

[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.



(a) Original parity logging



(b) Elastic parity logging

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

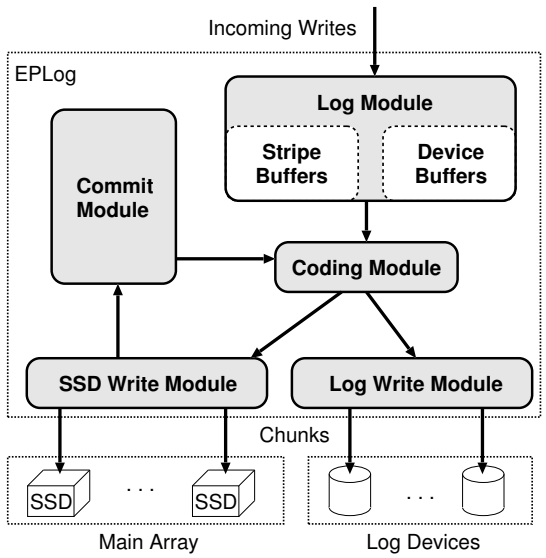


Fig. 2: EPLOG architecture.

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.

As opposed to 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 (and hence we call the parity logging scheme "elastic").

III. 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 (see Section II-B). In this work, we choose HDDs as log devices to achieve low-cost deployment. On the other hand, designing EPLOG faces different challenges, especially when we use HDDs as log devices. 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 further improve endurance and performance of EPLOG via caching, which is feasible for some applications (see Section II-A)?
- How do we manage metadata in a persistent manner in our EPLOG implementation?

A. 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 through 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 through the SSD write module, while the log chunks are issued to the log devices through 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 assumes RAID-6 (which can tolerate two device failures), two log devices are needed. We elaborate how EPLOG constructs log chunks in Section III-B.

The commit module regularly performs parity commit to ensure that the data and parity chunks in the main array reflect the latest updates. We elaborate the parity commit operation in Section III-C.

To further reduce parity traffic, we introduce two types of buffers in the log module, namely a *stripe buffer* and 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. We elaborate the caching design in Section III-D.

We carefully implement EPLOG to optimize its performance. In particular, our implementation ensures persistent metadata management. We elaborate the details in Section III-E.

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. Caching also helps to reduce the writes to SSDs and the amount of log chunks, so it can further mitigate parity commit overheads. We study these issues in our experiments (see Section V).

B. Write Processing

We first describe how EPLOG processes a single write request and constructs log chunks; in Section III-D, we extend the design for processing multiple write requests via caching. 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 III-A 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 (in the coding module) 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 Section IV, 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. 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 II-B 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.

C. 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.

In each parity commit operation, we identify the data stripes, via the metadata structure (see Section III-E), whose data chunks have been updated since the last parity commit. For each identified data stripe, we read the latest data chunks from SSDs, compute the corresponding parity chunks, and write back the updated parity chunks to SSDs in the main array. Finally, we update the metadata and release the space occupied by both the obsolete data chunks from the main array and log chunks from the log devices.

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 up-to-date data chunks, which will be used for computing parities, are kept in SSDs. This guarantees that the log devices can be always accessed in a sequential order when writing log chunks with the append-only policy (see Section III-B). In case of SSD failures, we scan and commit log chunks in batches, in which the batch size presents a trade-off between memory usage and parity commit overhead. We point out that parity commit introduces extra writes to the main array, yet we show that the write traffic remains limited compared to conventional RAID (see Section V). The reason is that we only need to perform parity commit on the latest 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 require special handling in SSD RAID arrays [18]. We pose the use of TRIM as future work.

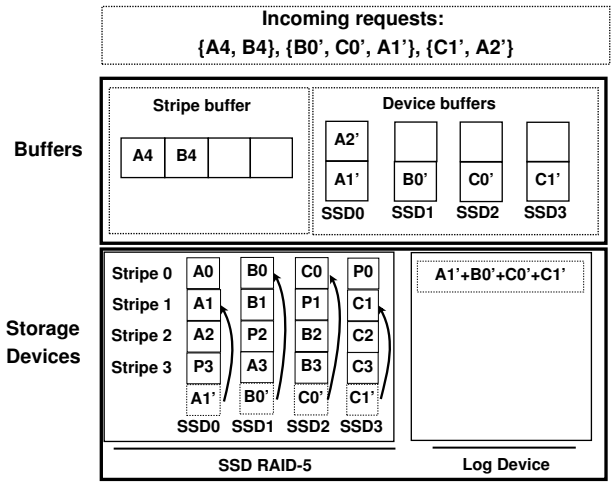


Fig. 3: Illustration of buffers in EPLOG.

D. Caching

To further reduce parity traffic, EPLOG supports an optional caching feature to batch-process multiple write requests in memory. It includes two types of buffers in the log module: a *stripe buffer* and multiple *device buffers*, which process new writes and updates, respectively.

The stripe buffer is used to cache new writes, which are directed to the main array, so as to increase the chance of full-stripe writes when generating data stripes. We set the size of the stripe buffer to be multiples of data stripes. Specifically, when a new write request arrives, the data chunks contained in the write request are appended to the stripe buffer. If the stripe buffer is full, all cached data chunks are grouped together to generate full data stripes and written to the main array in batch.

In addition, there are multiple device buffers, each of which is associated with an SSD in the main array. Each device buffer is used to cache update requests. The rationale is that real-world workloads often exhibit high locality both spatially and temporally [34], [43], [46], such that recently updated chunks and their nearby chunks tend to be updated more frequently. Thus, the device buffers can potentially absorb multiple updates for the same data chunk, thereby reducing both data chunks and log chunks written to the main array and the log devices, respectively. Specifically, when an update request arrives, each of the data chunks in the request is cached in the corresponding device buffer, according to the destined SSDs of these data chunks. If the same data chunk is found in the device buffer, it is directly updated in place. When one of the device buffers is full, we extract one data chunk from the head of each non-empty device buffer to form a log stripe.

We further illustrate via an example how the buffers work in EPLOG, as shown in Figure 3. We consider a stream of write requests issued to an SSD RAID-5 array. Specifically, when the new write request $\{\texttt{A4}, \texttt{B4}\}$ arrives, we add the data chunks to the stripe buffer. For the subsequent update requests $\{\texttt{B0'}, \texttt{C0'}, \texttt{A1'}\}$ and $\{\texttt{C1'}, \texttt{A2'}\}$, we add them to the device

buffers. We add the data chunks A1′ and A2′ to the device buffer of SSD0, since both their original data chunks A1 and A2 belong to SSD0. Similarly, we add the data chunks B0′, C0′, C1′ to the device buffers of SSD1, SSD2, and SSD3, respectively. Suppose that the size of each device buffer is configured to hold at most two data chunks. Now the device buffer of SSD0 becomes full. Thus, we construct a log stripe using the set of data chunks {A1′, B0′, C0′, C1′}. Finally, we write the new data chunks A1′, B0′, C0′, and C1′ to the main array by using the no-overwrite policy, and append the generated log chunks to the log devices as shown in Figure 3.

E. 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. 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, EPLOG implements erasure coding based on Cauchy Reed-Solomon codes [4] using the Jerasure 2.0 library [42].

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 stale ones. 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 [24], 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 [37], 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 I/O performance, EPLOG uses multi-threading to read/write data via the devices in parallel.

IV. Reliability Analysis

In this section, we analyze the system reliability of EPLOG and compare it with that of conventional RAID (i.e., we deploy RAID directly in the main array without using any log device). At first glance, the impact of EPLOG on the system reliability is debatable. EPLOG reduces write traffic to the main array via elastic parity logging. This slows down the wearing of flash memory, and potentially decreases the failure rates of SSDs as well [12], [26]. On the other hand, EPLOG adds log devices, while still tolerating the same number of device failures. This degrades the system reliability.

We resolve this debate as follows. Specifically, we measure the system reliability of EPLOG and conventional RAID in terms of mean-time-to-data-loss (MTTDL) (i.e., the expected time until data loss happens) through a *simplified* setting. Suppose that a storage system (either EPLOG or conventional RAID) reaches a certain system state after processing some workload. We fix the current system state, which implies that the corresponding error and recovery rates are fixed. Then under the same system state, we analyze how much longer the storage system continues to survive without any data loss based on MTTDL. Note that our simplified reliability analysis does not consider the time-varying bit error rate of flash memory [29]. Also, the correctness of MTTDL remains a concern [11]. Nevertheless, our analysis only serves to provide reliability comparisons between EPLOG and conventional RAID, and by no means do we use the absolute values to provide accurate quantifications.

A. MTTDL Computation

We first define the notations. Let n be the number of SSDs in the main array. Given a fixed system state, let λ_s be the failure rate of an SSD in EPLOG, and λ_s' be the failure rate of an SSD in conventional RAID. Let μ_s be recovery rate of an SSD. Let λ_h and μ_h be the failure rate and recovery rate of an HDD for EPLOG, respectively.

Note that both λ_s and λ_s' generally increase with the number of P/E cycles performed, which depends on the amount of write traffic. For simplicity, we assume that the failure rate of an SSD increases proportionally with the amount of writes issued[2].

$$\lambda_s = \alpha \lambda_s', \tag{1}$$

where α denotes the ratio of the amount of writes issued to the main array in EPLOG to that in conventional RAID. Note that EPLOG keeps $\alpha < 1$ by reducing parity writes to SSDs. In practice, we can estimate α through measurements. For example, from our experiments (see Section V), we can estimate that $\alpha = 0.5$ according to the results in Figure 7.

[2]The recent study [32] shows that the failure rate of an SSD does not monotonically increase as flash memory wears. However, as an SSD enters the wear-out period, which accounts for the majority of the SSD lifetime, the increasing trend actually holds and supports our assumption.

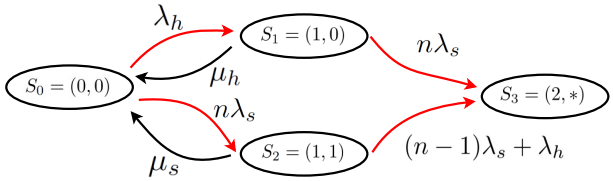


Fig. 4: State transition diagram of the Markov model for EPLOG's RAID-5.

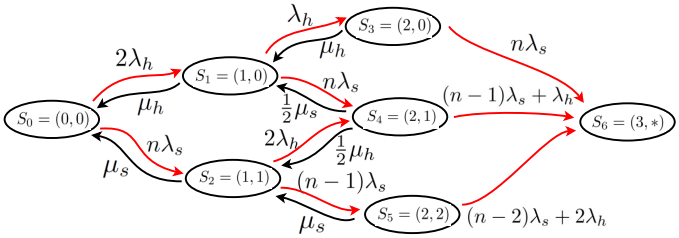


Fig. 5: State transition diagram of the Markov model for EPLOG's RAID-6.

EPLOG's RAID-5: We first consider EPLOG's RAID-5 design, which tolerates a single device failure. Recall that EPLOG adds one additional HDD as the log device. We now compute its MTTDL through a Markov model. Specifically, suppose that the storage system has a total of i device failures, j of which are SSDs. When $i \geq 2$, the storage system has a data loss, so we can focus on $0 \leq j \leq i \leq 2$. Let (i,j) denote a state. Thus, the storage system can be at one of the following states: $S_0 = (0,0)$, $S_1 = (1,0)$, $S_2 = (1,1)$, and $S_3 = (2,*)$ (note that S_3 can be $(2,1)$ or $(2,2)$, both of which imply a data loss).

Figure 4 shows the state transition diagram of the Markov model for EPLOG's RAID-5. Take $S_2 = (1,1)$ as an example, in which one SSD fails. If the failed SSD is recovered, S_2 transits to S_0, where the transition rate is μ_s. If one more device (either an SSD or the log device) fails, S_2 transits to S_3, where the total transition rate is $(n-1)\lambda_s + \lambda_h$.

We denote the system state at time t as $\boldsymbol{\pi}_t = (\pi_0(t), \pi_1(t), \pi_2(t), \pi_3(t))$, where $\pi_i(t)$ denotes the probability that EPLOG is at state S_i at time t. Let $\boldsymbol{\pi}(0) = (1,0,0,0)$, meaning that there is no device failure initially. Based on the Kolmogorov's forward equation, we have

$$\boldsymbol{\pi}'(t) = \boldsymbol{\pi}(t)\boldsymbol{Q}, \tag{2}$$

where $\boldsymbol{Q}$ denotes the transition rate matrix given by:

$$\boldsymbol{Q} = \begin{bmatrix} -(n\lambda_s+\lambda_h) & \lambda_h & n\lambda_s & 0 \\ \mu_h & -(\mu_h+n\lambda_s) & 0 & n\lambda_s \\ \mu_s & 0 & -(\mu_s+(n-1)\lambda_s+\lambda_h) & (n-1)\lambda_s+\lambda_h \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{3}$$

We can now derive the closed-form MTTDL of EPLOG's RAID-5 through standard approaches (e.g., by a Laplace transform) as follows:

$$\text{MTTDL} = \frac{\left[(2n-1)\lambda_s+\mu_s\right]+\left[2(\lambda_h+\mu_h)+\frac{(\lambda_h+\mu_h)(\lambda_h-\lambda_s+\mu_s)}{n\lambda_s}\right]}{\left[n(n-1)\lambda_s^2\right]+\left[n\lambda_s(2\lambda_h+\mu_h)+(\lambda_h+\mu_h)(\lambda_h-\lambda_s)+\lambda_h\mu_s\right]}. \tag{4}$$

EPLOG's RAID-6: We now consider EPLOG's RAID-6 design, which tolerates two device failures. Recall that EPLOG introduces two additional HDDs as log devices. We follow the same approach as in the RAID-5 case.

Figure 5 shows the state transition diagram of the Markov model for EPLOG's RAID-6. Let (i,j) denote a state as defined in the RAID-5 case. There are a total of six states, where $0 \leq j \leq i \leq 3$. In particular, the state $S_6 = (3,*)$ represents a data loss. One subtlety is that for the state S_4, which has one SSD failure and one HDD failure, we select a failed device for recovery via random tie-breaking. In this case, S_4 transits to S_1 and S_2 with rates $\frac{1}{2}\mu_s$ and $\frac{1}{2}\mu_h$, respectively.

We do not present the closed-form solution for the MTTDL of EPLOG's RAID-6 due to its complexity, but we can compute the MTTDL through numerical methods. We can further extend our analysis for the tolerance against a general number of device failures, and obtain the MTTDL through numerical methods.

Conventional RAID: The derivations of the MTTDLs for conventional RAID-5 and RAID-6 are well-known in the literature (e.g., [8], [9]). For completeness, we write down the results, in terms of λ'_s (see Equation (1)) and μ_s.

$$\text{MTTDL for RAID-5} = \frac{\mu_s + (2n-1)\lambda'_s}{n(n-1)(\lambda'_s)^2}, \tag{5}$$

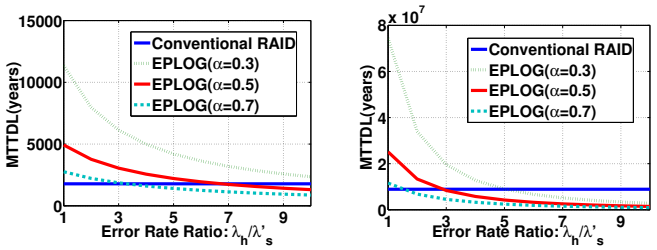
$$\text{MTTDL for RAID-6} = \frac{\mu_s^2+2(n-1)\lambda'_s\mu_s+(3n^2-6n+2)(\lambda'_s)^2}{n(n-1)(n-2)(\lambda'_s)^3}. \tag{6}$$

B. Results

To better illustrate whether EPLOG really improves the system reliability, we now compare the MTTDL of EPLOG and that of conventional RAID via numerical analysis. We first configure the parameters for conventional RAID. Suppose that the main array contains $n = 10$ SSDs. For the failure rate λ'_s, we note that it is challenging to maintain a minimum SSD lifetime of 3-5 years in a write-intensive environment [25], we set the average failure rate as $1/\lambda'_s = 4$ years (i.e., $\lambda'_s = 0.25$). For the recovery rate μ_s, suppose that the capacity of each SSD is around 400GB, and the I/O throughput for sequential writes is around 100MB/s, the average time to recover one device (i.e., rewrite all data) as around $1/\mu_s = 10^{-4}$ year (i.e., $\mu_s = 10^4$).

We now configure the parameters for EPLOG. We vary the failure rate of an HDD λ_h from λ'_s to $10\lambda'_s$, and still set $\mu_h = 10^4$. We set λ_s by considering three values of α, including 0.3, 0.5, and 0.7. Note that $\alpha = 0.5$ can be justified from our trace-driven evaluations (see Figure 7).

Figure 6 shows the MTTDL results versus the ratio λ_h/λ'_s for RAID-5 and RAID-6 (note that the MTTDL for conventional RAID is fixed since no HDD is used). It is reported that SSDs and HDDs have comparable failure rates [48] (i.e., $\lambda_h \approx \lambda'_s$). In this case, EPLOG achieves higher system reliability. For example, if $\lambda_h = \lambda'_s$ and $\alpha = 0.5$, EPLOG



(a) RAID-5 (b) RAID-6

Fig. 6: Reliability comparison between EPLOG and conventional RAID.

achieves 2.8× MTTDL compared to conventional RAID for both RAID-5 and RAID-6. The system reliability of EPLOG heavily depends on the failure rate of the log devices. As the failure rate λ_h increases, the system reliability of EPLOG drops dramatically, and the drop rate is even more significant when more HDDs are used (e.g., in RAID-6). In particular, when $\alpha = 0.5$, EPLOG maintains higher system reliability provided that λ_h is less than $6\lambda_s'$ and $2\lambda_s'$ for RAID-5 and RAID-6, respectively.

V. 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`. To summarize, our experiments have the following key findings: (i) EPLOG improves the endurance of SSD RAID by reducing both the write traffic to SSDs and the number of GC requests, (ii) EPLOG achieves potential gains with small-sized caching, (iii) EPLOG has limited parity commit overhead, (iv) EPLOG achieves higher I/O throughput than baseline approaches, and (v) EPLOG has limited metadata management overhead.

A. 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.

We compare EPLOG with two baseline parity update schemes. The first one is the Linux software RAID implementation based on `mdadm` (denoted by MD) [37], which implements conventional RAID and writes parity traffic to SSDs directly. The second one is the original parity logging (denoted by PL) [47], which performs parity updates at the stripe level (see Figure 1(a)). We implement PL based on our EPLOG prototype (see Section III-E) 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.

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

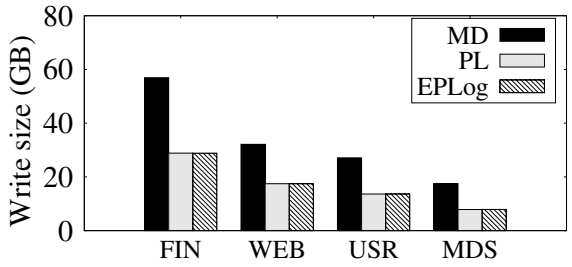
Traces: We consider four real-world I/O traces:

- `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 [36]. They describe the workloads of enterprise servers of three volumes, namely `web0`, `usr0`, and `mds0`, respectively, over a one-week period.

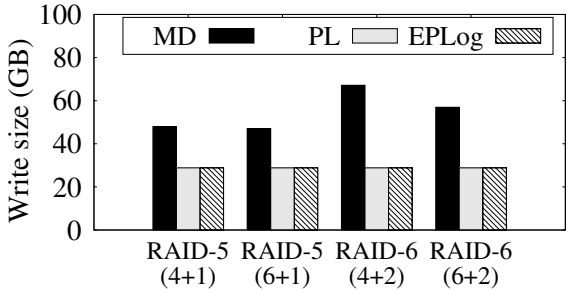
Note that each of the original traces spans a very large address space, yet only a small proportion of the addresses are actually accessed. To fit the traces into our testbed, which has a limited storage capacity, we compact each trace by skipping the addresses that are not accessed. Specifically, we 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 replaying each trace, we first sequentially write to all remaining segments (after our compaction) 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.

Table I 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.

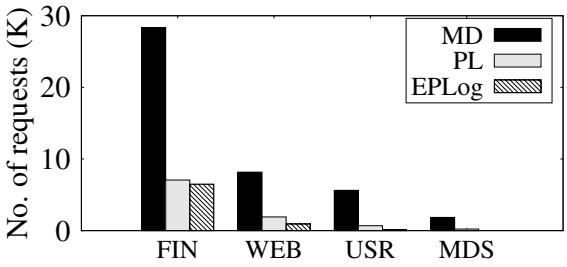


(a) Different traces under (6+2)-RAID-6

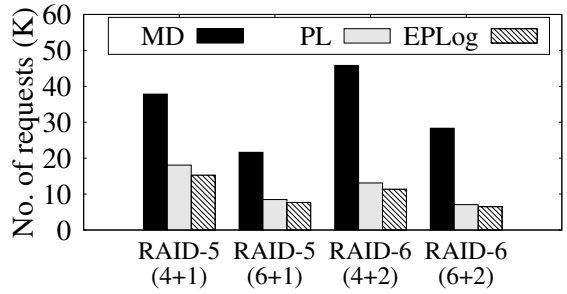


(b) Different RAID settings under FIN

Fig. 7: Experiment 1: Total size of write traffic to SSDs.



(a) Different traces under (6+2)-RAID-6



(b) Different RAID settings under FIN

Fig. 8: Experiment 2: GC overhead, measured in the average number of GC requests to each SSD. 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.

B. 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 7(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 5). Figure 7(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.

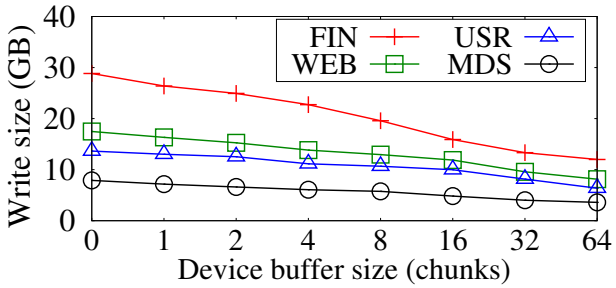
Experiment 2 (GC overhead): We study the endurance in terms of GC overhead, which we measure by the average number of GC requests to each SSD. Since SSD controllers do not expose GC information, we resort to trace-driven simulations using Microsoft's SSD simulator [2] that builds on Disksim [5]. For 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 workloads 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 the results across all SSDs in the array.

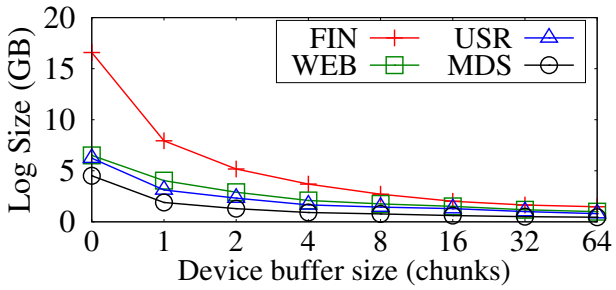
Figure 8 plots the total number of GC requests per SSD, averaged over all SSDs. Figure 8(a) shows the results across different traces under (6+2)-RAID-6. EPLOG significantly reduces the number of GC requests over MD, for example, by 77.1% under the FIN trace. This implies EPLOG significantly improves 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. Also, Figure 8(b) shows that EPLOG reduces 59.6-77.1% of GC requests over MD across different RAID settings under the FIN trace.

Experiment 3 (Impact of caching): We now evaluate the impact of caching of EPLOG. Since we focus on updates, we do not consider the effect of the stripe buffer (which is designed for new writes). Instead, we evaluate the impact of the device buffers. We vary the size of device buffer of each SSD from zero to 64 chunks. We measure both the total size of write traffic to SSDs and the total size of log chunks in the log devices.

Figure 9 shows the results for different traces under (6+2)-RAID-6. From Figure 9(a), the total size of write traffic to SSDs decreases as the device buffer size increases. For

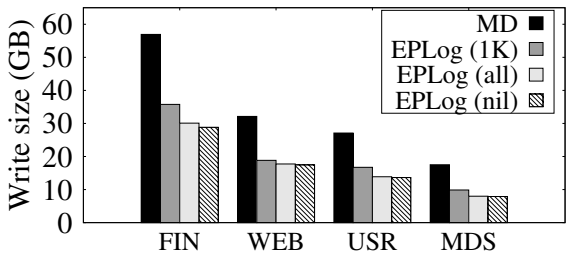


(a) Total size of write traffic to SSDs

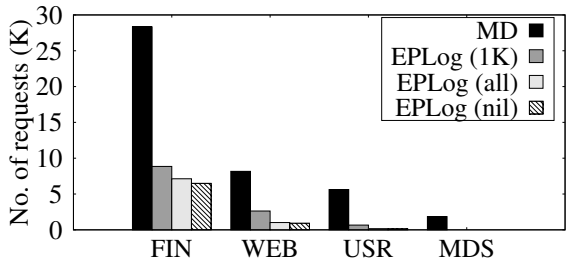


(b) Total size of log chunks

Fig. 9: Experiment 3: Impact of different device buffer sizes under (6+2)-RAID-6.



(a) Total size of write traffic to SSDs



(b) GC overhead

Fig. 10: Experiment 4: Total size of write traffic to SSDs and GC overhead under different parity commit cases and (6+2)-RAID-6 setting.

example, when the device buffer size reaches 64 chunks (i.e., 256KB per device), the write size drops by 53.3-58.4%. From Figure 9(b), the total size of log chunks drops even more significantly. For example, when the device buffer size reaches 64 chunks, the total size of log chunks decreases by 84.7-91.1%. Note that the total cache size of EPLOG is very small. For example, if we set the device buffer size per SSD as 64 chunks of size 4KB each, we only need 2MB. This implies that a small-sized cache can effectively absorb the data updates, and hence reduce both the write traffic to SSDs and the storage of log chunks.

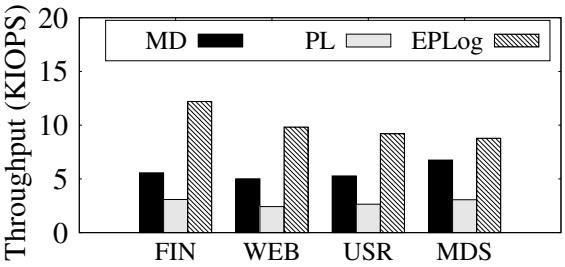
Experiment 4 (Parity commit overhead): Parity commit introduces additional writes (see Section III-C). We study the impact of parity commit. In particular, we consider three cases of parity commit: (i) without any parity commit, (ii) commit only at the end of the entire trace, and (iii) commit every 1,000 write requests. We also include the results of MD from Experiment 1 for comparison.

Figure 10 shows the parity commit overhead for different traces under (6+2)-RAID-6. Figure 10(a) first 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% and 24.9% when we perform parity commit at the end of a trace 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 10(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% and 67.8-88.2% less than that of MD when we perform parity commit at the end of a trace and every 1,000 write requests, respectively. The results show that the parity commit overhead remains limited if we perform parity commit in groups of writes.

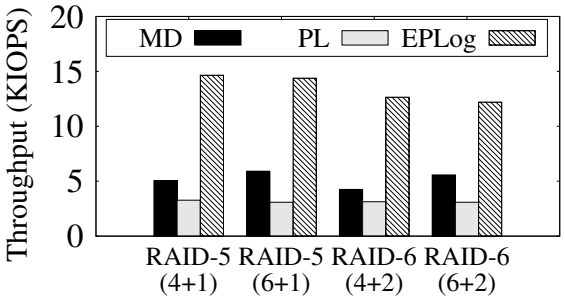
Experiment 5 (I/O performance): The previous experiments examine the write size and number of GC requests. We now examine the I/O throughput of EPLOG, measured as the number of user-level requests issued to the SSDs divided by the total time (in units of KIOPS); note that the total time also includes the overheads of writes to log devices. We replay each trace as fast as possible to obtain the maximum possible performance.

Figure 11 shows the throughput results. Figure 11(a), EPLOG outperforms MD by 30.1-119.2% and PL by 186.9-305.5% across different traces under (6+2)-RAID-6. Figure 11(b), EPLOG outperforms MD by 119.2-197.3% and PL by 295.7-366.1% across different RAID settings under the FIN trace. Both MD and PL read data before updating or logging parity on the update path. MD achieves higher throughput than PL, as MD directly updates parities on SSDs, while PL logs parity updates to HDD-based log devices for endurance. 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, and the reduction also leads to throughput gains.

Experiment 6 (Metadata management overheads): We now evaluate the overheads of the metadata checkpoint operations (see Section III-E). We consider the scenario where metadata is generated after a large number of random writes. We use IOzone [16] to first create continuous stripes covering a 8GB area on SSD RAID using sequential writes, and then issue uniform random updates of size 4KB each across all stripes. We then measure the total size of write traffic to SSDs under three cases: (i) full checkpoint after stripe creation, (ii) incremental checkpoint after all stripe updates, and (iii) full



(a) Different traces under (6+2)-RAID-6



(b) Different RAID settings under FIN

Fig. 11: Experiment 5: I/O performance.

	Setting	EPLOG
(i) Stripe creation	**w/o chkpt. (GB)**	10.922
	full chkpt. (GB)	10.961 (+0.36%)
(ii) Stripe update	**w/o chkpt. (GB)**	8.147
	incr. chkpt. (GB)	8.294 (+1.81%)
(iii) Stripe update	**w/o chkpt. (GB)**	8.147
	full chkpt. (GB)	8.331 (+2.25%)

TABLE II: Experiment 6: Total sizes of write traffic to SSDs with/without metadata checkpoint operations.

checkpoint after all stripe updates. We evaluate the metadata checkpoint overhead by comparing the cases with and without checkpoint operations.

Table II shows the results. Note that stripe creation issues new full-stripe writes, so EPLOG writes them to SSDs. The total write size is around 11GB, including parity writes. Later in stripe updates, EPLOG redirects parities to the log devices, and the total write size drops to around 8GB. Overall, the metadata checkpoint overhead in write size is at most 2.25%. The incremental checkpoint operation only writes dirty metadata after updates, and its overhead is less than that of the full checkpoint operation. The results show that EPLOG incurs low overheads in metadata management.

VI. Related Work

Researchers have proposed various techniques for enhancing the performance and endurance of a single SSD, such as disk-based write caching [46], read/write separation via redundancy [45], and flash-aware file systems (e.g., [23], [24], [30], [34]). 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 [24], [34] and efficient metadata management [23], [30], for further performance and endurance improvements.

Flash-aware RAID designs have been proposed either at the chip level [10], [15], [22] or at the device level [3], [26], [28], [31], [39], [40]. For example, Greenan *et al.* [10] keep outstanding parity updates in NVRAM and defer them until a full stripe of data is available. FRA [28] also defers parity updates, but keeps outstanding parity updates in DRAM, which is susceptible to data loss. Balakrishnan *et al.* [3] propose to unevenly distribute parities among SSDs to avoid correlated failures. Lee *et al.* [26] and Im *et al.* [15] propose the partial parity idea, which generates parity chunks from partial stripes and maintains the parity chunks in NVRAM. HPDA [31] builds an SSD-HDD hybrid architecture which keeps all parities in HDDs and uses the HDDs as write buffers. Kim *et al.* [22] 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.* [40] propose a diagonal coding scheme to address the system-level wear-leveling problem in SSD RAID, and the same research group [39] extends the elastic striping method by Kim *et al.* [22] with a hotness-aware design.

EPLOG relaxes the constraints of parity construction in which parity can be associated with a partial stripe, following the same rationale as previous work [15], [22], [26], [39]. Compared to previous work, EPLOG keeps log chunks with elastic parity logging using commodity HDDs rather than NVRAM as in [15], [26]. Also, instead of directly writing parity chunks to SSDs [22], [39], EPLOG keeps log chunks in log devices to limit parity write traffic to SSDs, especially when synchronous writes are needed (see Section II-A). While HPDA [31] 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.

VII. Conclusions

We present EPLOG, a user-level block device that mitigates parity update overhead in SSD RAID arrays through elastic parity logging. Its idea is to encode new data chunks to form log chunks and append 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 on commodity hardware, and evaluate EPLOG through reliability analysis and testbed experiments. We show that EPLOG improves reliability, endurance, and

performance. The source code of EPLOG is available at **http://ansrlab.cse.cuhk.edu.hk/software/eplog**.

ACKNOWLEDGMENTS

This work was supported in part by National Nature Science Foundation of China (61303048 and 61379038), Anhui Provincial Natural Science Foundation (1508085SQF214), CCF-Tencent Open Research Fund, the University Grants Committee of Hong Kong (AoE/E-02/08), and Research Committee of CUHK.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] J. Elerath and M. Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *IEEE/IFIP DSN*, June 2007.

[10] 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.

[11] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean Time to Meaningless: MTTDL, Markov Models, and Storage System Reliability. In *Proceedings of the 2nd USENIX HotStorage*, 2010.

[12] 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.

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

[14] 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.

[15] 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.

[16] IOzone. IOzone Filesystem Benchmark. http://www.iozone.org.

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

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

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

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

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

[22] 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.

[23] 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.

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

[25] S. Lee, T. Kim, K. Kim, and J. Kim. Lifetime Management of Flash-based SSDs Using Recovery-aware Dynamic Throttling. In *Proceedings of the 10th USENIX FAST*, 2012.

[26] 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.

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

[28] 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.

[29] Y. Li, P. P. C. Lee, and J. C. S. Lui. Stochastic Analysis on RAID Reliability for Solid-State Drives. In *IEEE SRDS*, 2013.

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

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

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

[44] 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.

[45] 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.

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

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

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

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