Elastic Parity Logging for SSD RAID Arrays: Design, Analysis, and Implementation (Supplementary File)

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

The following materials are supplementary to our main file.

1 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 [12]–[14], 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 1. We consider a stream

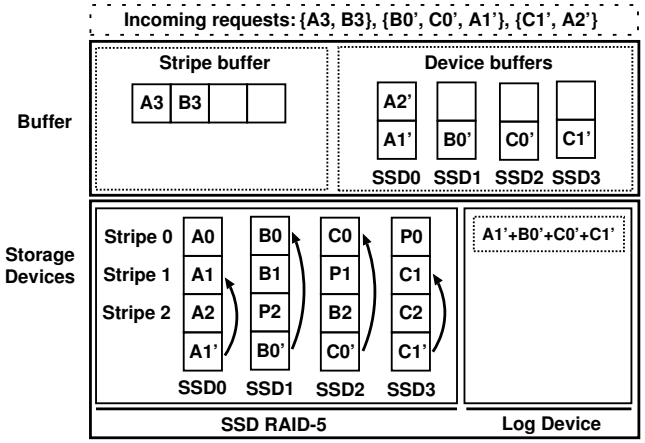


Fig. 1: Illustration of buffers in EPLog.

of write requests issued to an SSD RAID-5 array. Specifically, when the new write request {A3, B3} arrives, we add the data chunks to the stripe buffer. For the subsequent update requests {B0', C0', A1'} and {C1', 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 1.

2 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 also mitigates the failure rates of SSDs [5], [8]. On the other hand, EPLog adds log

- *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).*

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 [9]. Also, the correctness of MTTDL is debatable [4]. Nevertheless, our analysis only serves to provide reliability comparisons between EPLOG and conventional RAID, and by no means do we use the absolute values for accurate quantifications.

2.1 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[1]:

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

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.

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 2 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 $(n-1)\lambda_s + \lambda_h$.

1. The recent study [11] 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.

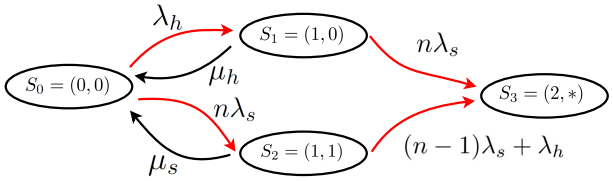


Fig. 2: State transition diagram of the Markov model for EPLOG's RAID-5.

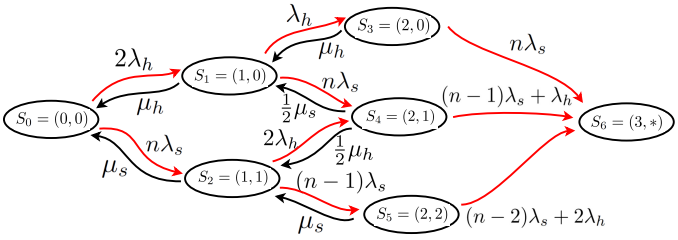


Fig. 3: State transition diagram of the Markov model for EPLOG's RAID-6.

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:

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

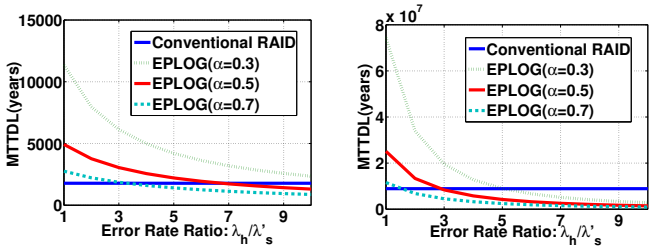
We can now derive the closed-form MTTDL of EPLOG's RAID-5 through standard approaches (e.g., by a Laplace transform) as follows:

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

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.

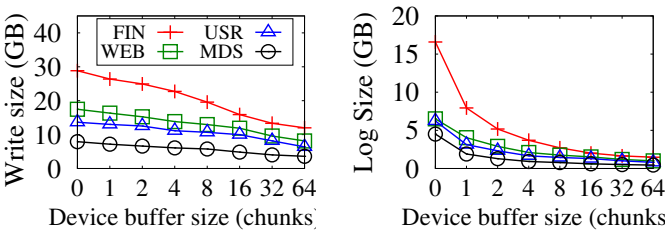
Figure 3 shows the state transition diagram of the Markov model for EPLOG's RAID-6. Let (i, j) denote a state as defined in the RAID-5 case. There are a total of six states, where $0 \leq j \leq i \leq 3$. In particular, the state $S_6 = (3,*)$ represents a data loss. One subtlety is that for the state 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.



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

Fig. 4: Reliability comparison between EPLOG and conventional RAID.



(a) Total size of write traffic to SSDs (b) Total size of log chunks

Fig. 5: Experiment S1: Impact of different device buffer sizes under (6+2)-RAID-6.

Conventional RAID: The derivations of the MTTDLs for conventional RAID-5 and RAID-6 are well-known in the literature (e.g., [2], [3]). For completeness, we write down the results, in terms of λ'_s (see Equation (1)) and μ_s.

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

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

2.2 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 [7], 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.

Figure 4 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 [15] (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 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.

3 Additional Experiments

We present additional experimental results on EPLOG's design. We refer readers to the main file about our evaluation setup.

Experiment S1 (Caching): We 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 5 shows the results for different traces under (6+2)-RAID-6. From Figure 5(a), the total size of write traffic to SSDs decreases as the device buffer size increases. For 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 5(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 S2 (I/O performance with `fio`): We measure EPLOG's basic I/O performance using the I/O benchmarking tool `fio` [1]. We use `fio` to generate workloads of user-level requests. We measure the performance by the number of requests issued per unit time (in request per second), which includes the overhead of writes to the HDD-based log devices.

Figure 6 shows the I/O performance of MD, PL, and EPLOG for sequential and random read/update workloads under (6+2)-RAID-6. We focus on 4KB-12KB requests, which introduce small partial stripe updates. EPLOG achieves higher throughput than MD and PL for both sequential and random updates, for example, by 455.8-573.1% and 374.1-422.7% for random updates, respectively. Also, MD, PL, and EPLOG have similar performance for sequential and random reads, as they all access the data chunks on SSDs directly.

Experiment S3 (Storage overhead): We examine the storage overhead of EPLOG under different parity commit cases. Table 1 shows the storage overhead across different traces

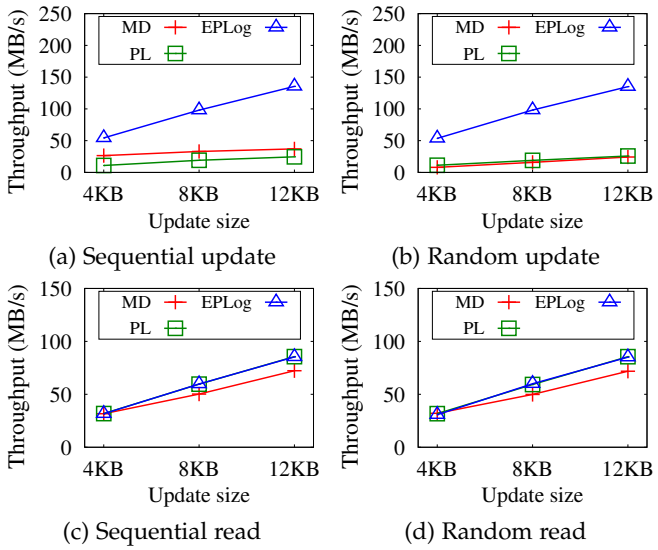


Fig. 6: Experiment S2: I/O performance measured by `fio` under 6+2)-RAID-6 in normal mode (without any SSD failure).

Cases	Stats.	FIN	WEB	USR	MDS
10 K	Min	0.03%	<0.01%	<0.01%	<0.01%
	Avg	0.50%	0.46%	0.54%	0.81%
	Max	2.46%	1.69%	7.17%	2.89%
1 K	Min	0.02%	<0.01%	<0.01%	<0.01%
	Avg	0.09%	0.08%	0.10%	0.15%
	Max	0.53%	0.59%	0.82%	1.19%

TABLE 1: Experiment S3: Storage overhead under different parity commit cases

under two parity commit cases. We measure the instantaneous additional storage usage every 1,000 requests before parity commit, and normalize it to the storage usage before trace replay (in which chunks are sequentially stored on SSDs). When we perform parity commit every 1,000 or 10,000 requests, the maximum storage overhead is 0.53-1.19% and 1.69-7.17%, respectively. The results show that EPLOG incurs limited storage overhead if we issue parity commit operations regularly.

Experiment S4 (Checkpoint overhead of metadata): We evaluate the overhead of the metadata checkpoint operations. We consider the scenario where metadata is generated after a large number of random writes. We use IOzone [6] 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 checkpoint after all stripe updates. We evaluate the metadata checkpoint overhead by comparing the cases with and without checkpoint operations.

Table 2 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.

	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 2: Experiment S4: Total sizes of write traffic to SSDs with/without metadata checkpoint operations.

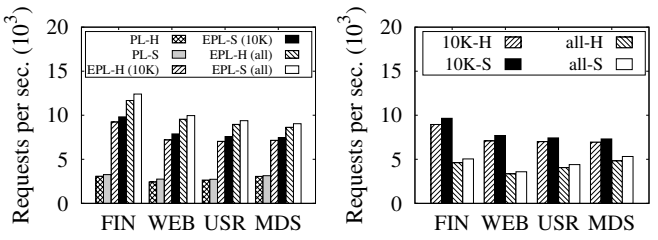
	FIN	WEB	USR	MDS
(i) before replay (MB)	15.92	27.42	15.69	11.86
(ii) 1K (MB)	16.25	28.09	16.25	12.44
(ii) 10K (MB)	17.71	29.71	20.28	13.70

TABLE 3: Experiment S5: Memory overhead of metadata.

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 overhead in metadata management.

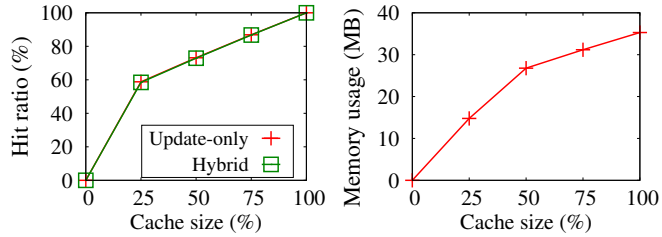
Experiment S5 (Memory overhead of metadata): We evaluate the overhead of maintaining up-to-date metadata in memory under real-world traces. We measure the maximum sizes of memory required to store the metadata (i) before trace replay, (ii) during trace replay with parity commit every 1,000 requests, and (iii) during trace replay with parity commit every 10,000 requests, as shown in Table 3. The memory overhead is 0.37-0.63% of the working set size before trace replay. When we perform parity commit every 1,000 requests and 10,000 requests, the overhead is 0.38-0.65% and 0.40-0.81% of the working set size, respectively. Before trace replay, the memory overhead is proportional to the number of data stripes required to cover the working sets of traces. Note that although the `MDS` trace has a larger working set size than the `USR` trace, it requires fewer data stripes to cover its working set (after we compact the traces) and hence incurs lower memory overhead. During trace replay, the memory overhead increases with the amount of update traffic between parity commit operations. Hence, we observe higher memory overhead when parity commit is performed less frequently (e.g., every 10,000 requests). Nevertheless, the memory overhead remains limited.

Experiment S6 (Using SSDs as log devices): We evaluate EPLOG when it uses SSDs as log devices instead of HDDs, and study whether SSD-based log devices can improve I/O performance over HDD-based ones. Figure 7 shows the I/O throughput results across different traces under (6+2)-RAID-6 when parity commit is performed every 10,000 requests and at the end of trace replay, and we consider both normal mode when there is no SSD failure (Figure 7(a)) and degraded mode when there is a double-SSD failure (Figure 7(b)). For comparison, we also evaluate PL using SSDs as log devices in normal mode (Figure 7(a)). We see that using SSD-based log devices improves I/O throughput as they allow faster parity logging during writes as well as parity commit. However, we do not see significant performance gains of SSD-based log devices over HDD-based ones; specifically, EPLOG's throughput increases by 1.85-



(a) EPLOG (EPL) and PL under no SSD failure (b) EPLOG under double-SSD failure

Fig. 7: Experiment S6: I/O performance of EPLOG with SSD-based (denoted by suffix "-S") and HDD-based (denoted by suffix "-H") log devices under (6+2)-RAID-6.



(a) Hit ratio (b) Actual memory usage

Fig. 8: Experiment S7: Effectiveness of object metadata cache in the key-value store.

9.45% in normal mode and by 5.1-10.1% in degraded mode, while PL's throughput increases by 2.5-13.5% in normal mode. The reason behind this small performance gain is that the difference between the random write throughput of SSDs in the main array and the sequential write throughput of HDD-based log devices is small (see Preliminary benchmarks in Section 5.1 of the main file). Note that EPLOG still outperforms PL by at least 137.8-200.7%, as PL reads the original data chunks for encoding log chunks, while EPLOG does not.

Experiment S7 (Object metadata cache in the key-value store): We further study the trade-off of the object metadata cache in our key-value store implementation between the hit ratio and the actual memory usage. Figure 8(a) shows the hit ratio. Both update-only and hybrid workloads have almost identical hit ratios, which reach 60% even by caching 25% of all key-value pairs. Figure 8(b) shows the actual memory usage of the cache (recall that the cache only stores the keys and metadata of key-value pairs, and the value size does not affect the actual memory usage). We see that the increase in actual memory usage drops as the size of the object metadata cache increases. The reason is that the ARC algorithm [10] also keeps track of the recently evicted items. As the size of the object metadata cache increases to 100%, no items are evicted. Thus, the ARC algorithm can save the memory for storing the history of the recently evicted items.

Experiment S8 (Write traffic per SSD): We now evaluate the per-SSD write traffic in order to examine the skewness of write traffic across SSDs in EPLOG. We report the average, maximum, minimum, and standard deviation of per-SSD write sizes of MD, EPLOG without parity commit, and EPLOG with parity commit every 10,000 requests. We do not consider PL, whose write size without parity commit is identical to that of EPLOG without parity commit.

Traces		Avg	Max	Min	SD
FIN	MD	8.14	8.42	7.83	0.18
	EPLOG (nil)	3.61	4.11	3.10	0.29
	EPLOG (10K)	4.23	4.60	3.96	0.20
WEB	MD	4.02	5.27	3.21	0.72
	EPLOG (nil)	2.18	2.88	1.68	0.35
	EPLOG (10K)	2.50	3.14	2.11	0.33
USR	MD	3.39	4.23	2.81	0.51
	EPLOG (nil)	1.70	2.23	1.19	0.33
	EPLOG (10K)	1.95	2.46	1.24	0.29
MDS	MD	2.19	2.67	1.43	0.41
	EPLOG (nil)	0.98	1.29	0.75	0.16
	EPLOG (10K)	1.16	1.52	0.90	0.17

TABLE 4: Experiment S8: Average (Avg), maximum (Max), minimum (Min), and standard deviation (SD) of per-SSD write sizes (in units of GB) across different traces under (6+2)-RAID-6.

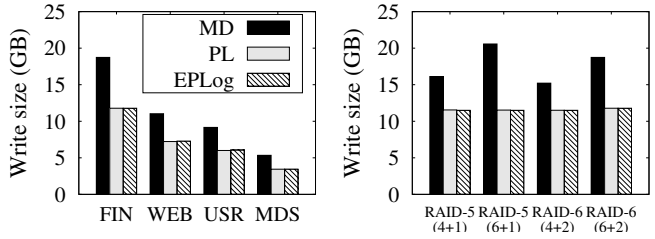
RAID settings		Avg	Max	Min	SD
(4+1)-RAID-5	MD	9.60	9.80	9.36	0.16
	EPLOG (nil)	5.77	6.15	5.08	0.40
	EPLOG (10K)	6.30	6.56	5.84	0.25
(6+1)-RAID-5	MD	6.71	7.07	6.38	0.23
	EPLOG (nil)	4.12	4.71	3.58	0.39
	EPLOG (10K)	4.40	4.92	4.06	0.27
(4+2)-RAID-6	MD	11.19	11.31	10.95	0.14
	EPLOG (nil)	4.81	5.73	4.20	0.52
	EPLOG (10K)	5.80	6.53	5.44	0.37
(6+2)-RAID-6	MD	8.14	8.42	7.83	0.18
	EPLOG (nil)	3.61	4.11	3.10	0.29
	EPLOG (10K)	4.23	4.60	3.96	0.20

TABLE 5: Experiment S8: Average (Avg), maximum (Max), minimum (Min), and standard deviation (SD) of per-SSD write sizes (in units of GB) across different RAID settings under the FIN trace.

Table 4 shows the results across different traces under (6+2)-RAID-6, and Table 5 shows the results across four RAID settings under the FIN trace. EPLOG achieves much lower per-SSD write sizes; its maximum per-SSD write size is less than the minimum per-SSD write size in MD. Depending on the workloads, EPLOG achieves lower standard deviations in per-SSD write sizes than MD under the WEB, USR, and MDS traces, while having a higher standard deviation than MD under the FIN trace. Thus, the skewness of write traffic across SSDs is workload-dependent, and we do not see strong evidence that EPLOG aggravates the skewness of write traffic across SSDs over MD.

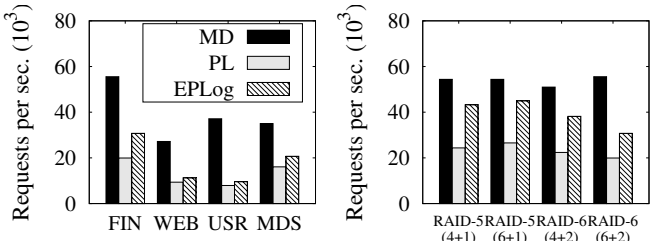
Experiment S9 (Buffered I/O): We thus far focus on I/O in O_DIRECT mode. We now examine the I/O performance of MD, PL, and EPLOG under buffered I/O, i.e., with O_DIRECT mode disabled. Since MD allocates a write cache of size 1MB for each SSD by default, we allocate the same amount of cache to PL and device buffer to EPLOG for fair comparisons. We flush the modified buffer cache every 1,000 requests via fsync().

Figures 9(a) and 9(b) show the total write sizes to SSDs across different traces under (6+2)-RAID-6 as well as across four different RAID settings under the FIN trace, respectively. EPLOG reduces 33.8-37.1% of the write traffic over MD across traces under (6+2)-RAID-6, while the write



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

Fig. 9: Experiment S9: Total size of write traffic to SSDs.



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

Fig. 10: Experiment S9: I/O performance comparisons under real-world traces

traffic of EPLOG is also 24.4-28.4% and 37.1-44.0% lower than MD for RAID-5 and RAID-6, respectively. As expected, the reduction in total write size is lower than that with O_DIRECT mode enabled, due to the absorption of writes by cache.

Figures 10(a) and 10(b) show the I/O performance across different traces under (6+2)-RAID-6 as well as across four different RAID settings under the FIN trace, respectively. We see that MD achieves the highest I/O performance. The throughput of EPLOG is 44.7-74.2% lower than MD, but 20.5-54.1% higher than PL across traces under (6+2)-RAID-6. Similarly, the throughput of EPLOG is 17.3-20.4% and 25.1-44.7% lower than MD, but 69.5-77.6% and 54.1-70.6% higher than PL for RAID-5 and RAID-6, respectively. To understand the performance gap, we measure the raw device performance with fsync() issued every 1,000 requests under buffered I/O. We observe that the random write performance of SSDs (164MB/s) is notably higher than the sequential write performance of HDD-based log devices (107MB/s). This contributes to the performance difference between EPLOG and MD. Nevertheless, EPLOG still improves the throughput of PL by eliminating reads to SSDs during updates. As we argue in the main paper (see Section 2.1), EPLOG targets workloads with small random writes, especially when modern storage systems issue synchronous writes.

REFERENCES

[1] J. Axboe. Flexible I/O Tester. https://github.com/axboe/fio, 2005.
[2] 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.
[3] J. Elerath and M. Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *Proc. of IEEE/IFIP DSN*, 2007.
[4] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean Time to Meaningless: MTTDL, Markov Models, and Storage System Reliability. In *Proc. of USENIX HotStorage*, 2010.
[5] 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.
[6] IOzone. IOzone Filesystem Benchmark. http://www.iozone.org.
[7] S. Lee, T. Kim, K. Kim, and J. Kim. Lifetime Management of Flash-based SSDs Using Recovery-aware Dynamic Throttling. In *Proc. of USENIX FAST*, 2012.
[8] 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.
[9] Y. Li, P. P. C. Lee, and J. C. S. Lui. Stochastic Analysis on RAID Reliability for Solid-State Drives. In *Proc. of IEEE SRDS*, 2013.
[10] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of USENIX FAST*, 2003.
[11] 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.
[12] 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.
[13] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *USENIX Winter 1993 Technical Conference*, 1993.
[14] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proc. of USENIX FAST*, 2010.
[15] K. Thomas. Solid State Drives No Better Than Others, Survey Says. www.pcworld.com/article/213442.