Efficient caching on parity chunks in RAID-enabled SSDs

Zhigang Cai¹, Lihao Song¹, and Xiaoning Peng², a)

Abstract Caching the updated parity chunks in the RAID buffer can absorb update operations on parity chunks of data stripes in RAID-enable SSDs. Thus, the negative effects of write penalty can be minimized. However, caching integrated parity chunks fails to efficiently use the capacity-limited RAID buffer, as sometimes a major part of parity chunk are not modified comparing with the parity chunk saved in SSD blocks. This paper presents a patch-based cache management scheme on parity chunks, for improving cache use efficiency of RAID-enabled SSDs. Specifically, it only caches the modified portions of parity chunks in the buffer (called parity patches), which are corresponding to the updated parts of data chunks in the same stripe. Through a series of simulation tests on several disk traces, we illustrate our proposal can noticeably reduce the I/O latency by between 12.2% and 17.1% and the number of block erases by 20.8% on average, in contrast to state-of-the-art approaches. In brief, our proposed cache management scheme can work in RAID-enabled SSDs to achieve better I/O performance and extend the lifespan of SSDs.

Keywords: SSDs, RAID-5, cache management, parity, data patch

Classification: Circuits and modules for storage

1. Introduction

Solid state drivers (SSDs) store data electronically, which have the advantages of broad bandwidth, energy saving, and volumetric capacity [1, 2, 3]. In order to further reduce the storage cost per bit of SSDs, the feature size of NAND flash memory cells reaches the limit of the 10nm level. Then, the increase in bit error rates accompanying with the feature size shrinking is now becoming a major issue to be reckoned with in SSDs [4, 5].

Device-level redundancy is commonly the first front of dealing with storage failures [6]. To enhance SSD reliability, RAID (Redundant Array of Independent Disks) has been applied inside of SSD, as parallelism between channels or chips can offer supports to implement RAID into a single SSD [7, 8, 9]. As one of standard RAID levels, RAID-5 consists of block-level striping with distributed parity, and has advantages in load-balancing and I/O parallelism, so it is commonly used to enhance the reliability of SSD [10, 11, 12, 13].

Though enabling RAID-5 seems to increase SSD reliability, it doubles write operations as every write operation on the data chunk leads to another write on the parity chunk1 (termed as write penalty) [14, 15]. More importantly, each SSD block can only bear a limited number of erases, and the extra parity updates must cause unexpected erases that shortens the SSD endurance [15]. Regarding the issue of mitigating write penalty, a DRAM buffer is often equipped at the RAID (SSD) controller to absorb write operations for parity chunks that are the most frequently updates in RAID [12, 14, 16, 17]. Then, a part of parity updates can be completed in the buffer, to avoid relevant parity writes onto SSD blocks [18].

In other words, caching certain parity chunks of hot write stripes in the RAID buffer can reduce the number of write operations for parity updates, and the cached parities will not be flushed onto the SSD blocks until the buffer space is not enough. To cut down the overhead of reading data chunks to calculate parity chunks, Im et al. [17] introduced caching partial parity data that is the XOR result of only a very small part of data chunk is modified on the previous version of data. That is, a major part of the parity chunk of the relevant stripe generally keeps unchanged, which makes the RAID buffer less effective.

To better boost the use efficiency of the capacity-limited RAID buffer, and enhance I/O performance of RAID-5 enabled SSDs, this paper presents a patch-based data management scheme on the cached parity chunks to efficiently utilize the RAID buffer. Specially, the proposed scheme divides a parity chunk into multiple patch units, and only modified patch units are kept in the RAID buffer. Then, it is possible to directly calculate the parity chunks in the cache space, without reading relevant data chunks from SSDs. However, all these (parity) caching schemes buffer the data in the chunk unit even though only a very small part of data chunk is modified on the previous version of data. That is, a major part of the parity chunk of the relevant stripe generally keeps unchanged, which makes the RAID buffer less effective.

1.1. Data Chunk and Patch Unit

1 A (data/parity) chunk is normally referred to a page in RAID-enabled SSDs. Namely, the size of chunk is equal to that of a page.
The rest of the paper is organized as follows: Section 2. depicts the related work and motivations. The approach of patch-based cache management on parity chunks is presented in Section 3. Section 4. describes the evaluation experiments and relevant discussions. Finally, the paper is concluded in Section 5.

2. Related work and motivation

2.1 Background
Due to the nature of RAID-5, whenever a data chunk is updated, the old parity chunk of the stripe is need to be read for computing a new parity chunk, and the new parity chunk have to be flushed into the SSD device as well. To relieve the negative effects of this nature (i.e. write penalty), the RAID buffer is used to reduce the quantity of writes forwarded to underlying storage devices in RAID systems [19, 20, 21, 22].

Through absorbing certain update requests on both data and parity chunks, flushing operations on SSD blocks will be significantly cut down. Similarly, Pan et al. [18] caches (hot) update chunks in the buffer, to absorb write requests onto parity chunks and avoid flushing parity data to SSD blocks, to remit the negative effects of write penalty.

On the other side, considering parity chunks are the most frequently modified data in the RAID storage, the RAID buffer is used to cache the parity data chunks until certain specific conditions are activated RAID-5 [17, 23]. Then, such delay parity update scheme can definitely cut down the parity write times. Moreover, Im et al. [10, 24, 25] presented a partial parity technique to minimize the number of read operations required to calculate a parity, by exploiting the implicit redundant data of flash memory. Based on the partial parity update approach, Chung et al. [12] proposed to use a separate data cache for holding frequently read data to speed up calculation of parity. Kim et al. [8, 26] introduced a RAID scheme that allows flexible stripe sizes and parity placement to minimize parity updates, but it needs more SSD pages for holding parity data.

2.2 Motivation
As discussed, the RAID buffer holds the frequently modified parity chunks, to absorb parity update requests to the relevant stripes [12]. In order to exploit how many contents in the buffered parity chunks keep unchanged since the last modification, we have recorded the percentage of unchanged portions while running the benchmarks. Figure 1 shows the results of changed/unchanged data ratio in buffered parity chunks after running the selected traces with a limited capacity of RAID buffer (i.e. 128K, 256K, and 512K in our tests). The details about the selected traces and experimental methodology can be found in Section 4.

As seen, a major part of data in the buffered parity chunks (i.e. 54.0% on average) are not new data, that can also be found in the SSD blocks. Such observations motivate us to propose an efficient buffer management approach in the RAID buffer having valuable but limited capacity, to absorb more update operations onto parity chunks by only buffering their modified parts.

3. Patch-based caching on parity

3.1 Architecture overview
In the process of dealing with a write requests on the data chunk in RAID-5 enabled SSDs, another new page will be retrieved to hold the new data chunk. Then, the parity chunk in the same stripe is supposed to be updated consequently. Figure 2 shows a comparison of high-level architecture overview on conventional and patch-based cache management on parity chunks in a RAID-5 enabled SSD. In the illustration, we assume only two portions of the data chunk in the stripe are needed to be updated.

Figure 2(a) demonstrates the work-flow of conventional caching management on parity chunks. As seen, it needs to buffer the new parity chunk of Stripe 0 when one of its data chunk (i.e. D0) is updated. The fact is, however, the parts of a and b in D0 are new contents, and only the corresponding parts of parity chunk of P0 are needed to be cached in the SSD buffer, as the remainder contents of the parity chunk can be retrieved from the underneath SSD page.

In order to efficiently use the SSD buffer (i.e. the RAID buffer) for better performance improvements, we propose patch-based caching management to only keep the modified parts of the parity chunk in the SSD buffer, as illustrated in Figure 2(b). That is to say, we cache the new contents of the parity chunk, which are corresponding to the updated portions of the data chunk in the same stripe.

3.2 Patch-based caching on parity chunks
The parity chunk will be accordingly modified if one of data chunks in the same stripe is updated. We make use of XOR compression to identify the modified portions by comparing the new data chunk with the dirty data chunk. That indicates the corresponding parts of the parity chunk in the same stripe are supposed to be updated.

Figure 3 shows the proposed bitmask-based mechanism for locating which parts of parity chunk should be changed. We first compare the dirty data chunk saved on the SSD block with the new data chunk in the same stripe received by the RAID controller to yield a series of data units (e.g. 512 byte per unit in our tests by default). Since we expect that many parts of the diff chunk are filled with zeros, we then generate a bitmask to describe which units hold non-zero values. Then, we call these non-zero parts as parity patches.
Comparison of conventional caching (a) and the proposed patch-based caching (b) on parity chunks in a chip-level RAID-5 SSD, after Chunk D0 of Stripe 0 is modified with new parts a and b.

Furthermore, to merge parity patches in the case of having duplicate updates to the same data stripe, our proposal will search the bitmask for checking the offset of patches and making sure overlaps exist or not. If there is an overlap of parity patches, it will be directly ejected from the cache as the overlap part of parity is changed back to that of the parity chunk saved on the SSD block.

We emphasize that the motivation of caching parity patches in the RAID buffer is to better make use of the limited buffer space for holding more parity patches of existing data stripes, compared with buffering entire parity chunks. Therefore, more write stripe updates can be satisfied by temporarily saving the modified contents in the buffer, and I/O performance can be consequently enhanced. By referring back to Figure 2(b), the data stripe of $S0$ has two parity patches in the RAID buffer, meanwhile the stripe of $S5$ has only one but a larger size of parity patch.

3.3 Data structure and implementation

To manage the buffered parity patches and speedup patch merging, we maintain a doubly linked list for recording the information on the cached parity chunks. In the linked list, each node corresponds to a parity chunk, and contains the information on all patches of the parity chunk. Figure 4 illustrates the data structure for recording patches of the buffered parity chunk.

Because of limited capacity of the RAID buffer, we employ the Least Recently Used algorithm (LRU) to replace the buffered parity patches. That is, if the buffer space is not enough, the patches of the parity chunks which have not been accessed recently will be ejected from the buffer.

4. Experiments and evaluation

This section first describes the experimental setup. Then, evaluation results and relevant discussions are presented, to show the feasibility and applicability of our proposal. At last, we make a brief summary about the findings obtained from the evaluation tests.

4.1 Experimental setups

Considering the SSD controller has limited computation power and memory capacity, we conducted tests on a local ARM-based machine. This machine has an ARM Cortex A7 Dual-Core with 800MHz, 128MB of memory and 32-bit Linux (ver 3.1). We have made use of a widely used SSD simulator of SSDsim (ver 2.1), to conduct trace-driven tests [27]. Note that SSDsim can only model the performance of SSDs, it cannot store and restore real data for each request. Table I presents our settings of SSDsim in experiments. Since the most of file systems have standardized on 512B blocks to divide physical disk units [28], we have conducted certain sensitive analysis on the patch unit size of 512B to 2048B, and found the patch unit size of 512B can perform the best in a major part of cases. Thus, we set the size of patch unit as 512B in our evaluation.

We employed 6 commonly used disk traces, the ads trace comes from Microsoft Production Server [29]. The other five block I/O traces are recently collected from a part of an enterprise VDI (Virtual Desktop Infrastructure) [30]. To be specific, they are 2016021615-LUN0 (labeled as lun0), 2016021619-LUN1 (lun1), 2016021619-LUN2 (lun2), 2016021615-LUN3 (lun3), and 2016021615-LUN4 (lun4). Table II reports the details on the selected traces. Specially, the indicator of Frequent R means the ra-
In addition to the proposed scheme (labeled as Pattern), which caches the modified portions of parity chunks in the RAID buffer, we selected the two comparison counterparts in our evaluation:

- **FRA** [23], which stands for Flash-aware Redundancy Array and caches parity updates in the RAID buffer. That is, the parity updates are not included in the critical path of read and write operations, and then the I/O response time can be reduced. In our tests, we set it as the baseline scheme.

- **Partial parity caching** (labeled as **PPC**) [12], which might be the most related work on RAID buffer management. Besides caching some frequently updated parity chunks in the RAID buffer to remit side-effects of write penalty, it also caches certain data chunks to speedup calculating parity.

### 4.2 Results and discussions

#### 4.2.1 Average I/O latency

Because the average I/O latency depends on the size of requests in the traces, it varies greatly when replaying different I/O traces. Therefore, we collect the normalized I/O response performance in this section, and Figure 5 shows the results of read time, write time and the total I/O time. As seen, the parity buffer can reduce the write latency greatly, though it does significantly not contribute to read latency reduction.

In contrast to the comparison counterpart schemes, the proposed **Patch** method can reduce both read and write time to a certain extent in a major part of traces. Specifically, in contrast to the baseline scheme and **PPC**, **Patch** can cut down the overall I/O time by 17.1% and 12.2% respectively. This fact verifies patch-based data management offers caching more parity chunks in the RAID buffer, which greatly contribute to the reduction in I/O latency. Besides, we can understand that the **Patch** approach can achieve a similar I/O improvement while the cache size is becoming large, that indicates our proposal has well scalability.

#### 4.2.2 Block erases

Absorbing parity update requests in the RAID buffer can cut down the I/O response time and extends the lifetime of SSDs, through reducing the number of block erases.

In contrast to the comparison counterpart schemes, the proposed **Patch** method can reduce both read and write time to a certain extent in a major part of traces. Specifically, in contrast to the baseline scheme and **PPC**, **Patch** can cut down the overall I/O time by 17.1% and 12.2% respectively. This fact verifies patch-based data management offers caching more parity chunks in the RAID buffer, which greatly contribute to the reduction in I/O latency. Besides, we can understand that the **Patch** approach can achieve a similar I/O improvement while the cache size is becoming large, that indicates our proposal has well scalability.

---

**Table I** Experimental settings of SSDsim

| Parameters        | Values | Parameters        | Values |
|-------------------|--------|-------------------|--------|
| Channel Size      | 8      | Read latency     | 0.075ms|
| Chip Size         | 4      | Write latency    | 2ms    |
| Plane Size        | 2      | Erase latency    | 15ms   |
| Block per plane   | 2048   | Cache rd/wr      | 0.001ms|
| Page per block    | 128    | GC Threshold     | 30%    |
| Page Size         | 8KB    | RAID Buffer      | 128/256/512KB|
| FTL Scheme        | Page   | RAID Level       | 5 (Chip-level)|
| Wear-leveling     | Static |                  |        |

**Table II** Trace specifications (ordered by write ratio)

| Traces | Req #  | Wr Ratio | Wr Size | Frequent R (Wr) |
|--------|--------|----------|---------|-----------------|
| lun0   | 1073405| 73.1%    | 7.6KB   | 12.6%(92.4%)    |
| lun1   | 534529 | 66.6%    | 9.3KB   | 12.4%(88.4%)    |
| lun3   | 821226 | 52.9%    | 9.0KB   | 16.6%(87.4%)    |
| lun4   | 527385 | 55.4%    | 9.8KB   | 14.6%(84.4%)    |
| lun2   | 1644153| 23.1%    | 12.0KB  | 12.2%(74.5%)    |
| ads    | 1532120| 9.5%     | 7.0KB   | 26.1%(2.3%)     |

Fig. 5 The normalized I/O response time.
It is not abnormal that *Patch* brings about more than 47.7% erase reduction after running *ads* with a 512K RAID buffer. Because *ads* is a read intensive workload, and has a small footprint, the absolute erase number is relative small. More exactly, *FRA*, *PPC* and *Patch* respectively result in merely 625, 540 and 245 block erases after running the benchmark of *ads*.

### 4.2.3 Cache hits

The metric of *cache hit* is defined as the number of the parity updates fulfilled in the RAID buffer, and Figure 7 shows the results of this metric. As seen, *Patch* greatly outperforms other two schemes by up to 79.3%, since it can cache the parity data for more data stripes.

Another interesting fact shown in the figure is about *PPC* performs the worst in this metric, but it can still yield better results in the metric of I/O latency. This is because *PPC* allocates a part of cache for holding the frequently read data, which may benefit to servicing relevant read requests and computing parity of relevant stripes without reading them from SSD blocks.

### 4.2.4 Overhead analysis

The main space overhead of the proposed approach is to hold the data structure that reflects offset and size information on parity patches. Then, we argue that the space overhead is neglectable. Moreover, performing XOR computations to generate the parity patches may bring about side-effects on I/O by delaying the processing on incoming I/O requests.

### 4.3 Data recovery analysis

This section only discusses the recovery process of our proposed method, for fighting with the chunk/parity data loss on a SSD chip. (1) To restore the lost *parity* chunks, it expects reading the active data chunks in the same stripe to rebuild the lost (obsolete) parity chunks. Then, the parity patches in the cache will be used to form the latest parity chunks. (2) To restore the lost *data* chunks, the cached parity patches is first XORed with the dirty parity chunks, to generate the latest parity chunks. After that, the other active data chunks and the latest parity chunks in the same data stripes are used to recover the lost data chunks.

*Patch* keeps modified parts of parity chunks in the buffer, so it may require more time to restore the lost data for the purpose of ensuring data consistency. Figure 8 shows the time required for restoring the lost data on *Chip 0*. Compared with *FRA* and *PPC*, *Patch* requires more 7.9% and 11.2% time to recover the data, as it must carry out XOR computations to merge the patches and the obsolete parity chunks for guaranteeing data consistency.

| Table III | Computation overhead of *Patch* (second) |
|-----------|------------------------------------------|
| Cache size | lun0 | lun1 | lun3 | lun4 | lun2 | ads |
| 128K       | 14.29 | 5.19 | 6.28 | 4.34 | 7.28 | 1.91 |
| 256K       | 14.29 | 5.19 | 6.28 | 4.34 | 7.27 | 1.91 |
| 512K       | 14.29 | 5.19 | 6.28 | 4.34 | 7.28 | 1.91 |
4.4 Summary
Through comparing the related work and the proposed patch-based buffer management scheme on parity data in RAID-5 enabled SSDs, we emphasize the following two key observations. First, caching modified parts of parity chunks can make the capacity-limited RAID buffer to hold more the latest parity data, and then reduce the I/O response time. Second, absorbing more parity updates can reduce the number of flush operations onto SSD blocks, and then cut down the number of block erases for better lifetime of SSDs.

5. Conclusion
This paper has proposed, implemented, and evaluated a patch-based cache management scheme in RAID-5 enabled SSDs. Instead of caching the whole parity chunks in the RAID buffer, our approach keeps only the updated parts of parity chunks in the buffer. That is, we make use of XOR to compute the old parity chunk and the new data chunk to form the modified parts of parity, and only cache them in the RAID buffer. Thus, the RAID buffer can absorb more parity updates while the relevant data chunks are modified.

Through a series of emulation experiments based on several realistic disk traces, we show that the proposed scheme of patch-based caching can noticeably cut down the overall I/O response time by between 12.2% and 17.1%. Our measurements also demonstrate the proposed approach can decrease the number of erases by 20.8% on average, in contrast to other comparison counterparts.

Acknowledgments
This work was partially supported by “National Natural Science Foundation of China (No. 61872299)”, and “National Key Technologies R&D Program of China (No. 2018AAA0102102)”. In particular, we thank our shepherd, Dr. Jianwei Liao.

References
[1] J. Liao, et al.: “Adaptive wear-leveling in flash-based memory,” IEEE Comput. Archit. Lett. 14 (2015) 1 (DOI: 10.1109/lca.2014.2329871).
[2] C. Matsui, et al.: “Design of hybrid SSDs with storage class memory and NAND flash memory,” Proc. IEEE 105 (2017) 1812 (DOI: 10.1109/jproc.2017.2716958).
[3] Y. Arakawa, et al.: “Silicon photonics for next generation system integration platform,” IEEE Commun. Mag. 51 (2013) 72 (DOI: 10.1109/mcom.2013.6476868).
[4] X. Cai, et al.: “Error characterization, mitigation, and recovery in flash-memory-based solid-state drives,” Proc. IEEE 105 (2017) 1666 (DOI: 10.1109/jproc.2017.2713127).
[5] W. Lee, et al.: “Inter-page-based endurance-enhancing lower state encoding for MLC and TLC flash memory stores,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (2019) (DOI: 10.1109/tvlsi.2019.2912228).
[6] M. Balakrishnan, et al.: “Differential read: rethinking read for ssd reliability,” ACM Trans. Storage 6 (2010) 1 (DOI: 10.1145/1807060.1807061).
[7] K. Greenan, et al.: “Building flexible fault-tolerant flash-based storage systems,” HotDep (2009).
[8] J. Kim, et al.: “Chip-level read with flexible stripe size and parity placement for enhanced ssd reliability,” IEEE Trans. Comput. 65 (2016) 1116 (DOI: 10.1109/tc.2014.2375179).
[9] Y. Du, et al.: “CD-RAIS: constrained dynamic striping in redundant array of independent SSDs,” CLUSTER (2014) 212 (DOI: 10.1109/cluster.2014.6968742).
[10] S. Im and D. Shin: “Flash-aware RAID techniques for dependable and high-performance flash memory SSD,” IEEE Trans. Comput. 60 (2010) 80 (DOI: 10.1109/tc.2010.197).
[11] B. Kim, et al.: “Design tradeoffs for SSD reliability,” FAST (2019) 281.
[12] C.-C. Chung and H.-H. Hsu: “Partial parity cache and data cache management method to improve the performance of an SSD-based RAID,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 22 (2014) 1470 (DOI: 10.1109/tvlsi.2013.2275737).
[13] J. Kim, et al.: “Improving SSD reliability with RAID via elastic striping and anywhere parity,” DSN (2013) (DOI: 10.1109/dsn.2013.6575359).
[14] J. Wan, et al.: “DEFT-cache: A cost-effective and highly reliable SSD cache for RAID storage,” IPDPS (2017) 102 (DOI: 10.1109/ipdps.2017.54).
[15] Y. Li, et al.: “Workload-aware elastic striping with hot data identification for SSD RAID arrays,” IEEE Trans. Comput.-Aided Design Intergr. Circuits Syst. 36 (2016) 815 (DOI: 10.1109/tcad.2016.2604292).
[16] Y. Chen, et al.: “Efficient and consistent NVMM cache for SSD-based file system,” IEEE Trans. Comput. 68 (2018) 1147 (DOI: 10.1109/tc.2018.2870137).
[17] S. Im and D. Shin: “Delayed partial parity scheme for reliable and high-performance flash memory SSD,” MSST (2010) 1 (DOI: 10.1109/mss Exchange (2010) 1 (DOI: 10.1109/jproc.2017.2716958).