Optimizations of Linux Software RAID System for Next-Generation Storage

Shushu Yi[1], Yanning Yang[2], Yunxiao Tang[1], Zixuan Zhou[1], Junzhe Li[1], Chen Yue[2]
Myoungsoo Jung[3], Jie Zhang[1]
Computer Hardware and System Evolution Laboratory
Peking University[1], Beijing University of Posts and Telecommunications[2]
Korea Advanced Institute of Science and Technology (KAIST)[3]

I. Introduction

Redundant Array of Inexpensive Disks (RAID) has been widely adopted to enhance the performance, capacity, and reliability of the SSD array. However, as SSD has experienced significant technology shifts, the RAID engine is becoming the performance bottleneck of the future storage system that employs the next-generation SSDs. Linux software RAID, referred to as mdraid [2], can break the performance bound by employing multiple CPU threads to prepare the parity data simultaneously. However, this approach can impose significant software overheads, which in turn introduces a huge burden to the CPU. Consequently, the performance of mdraid, unfortunately, cannot scale as the number of CPU threads and SSD devices increase [3]. Our evaluation results reveal that the overheads of the lock mechanism account for 30.8% of the total delays in mdraid storage stack. One may consider removing the lock mechanism from the mdraid. However, the locks play a critical role in guaranteeing crash consistency and taking charge of data management.

To address this, we propose *ScalaRAID*, which refines the role domain of locks and designs a new data structure to prevent different threads from preempting the RAID resources. By doing so, ScalaRAID can maximize the thread-level parallelism and reduce the time consumption of I/O request handling. Our evaluation results reveal that ScalaRAID can improve throughput by 89.4% while decreasing 99.99th percentile latency by 85.4% compared to mdraid.

II. Background and Motivation

Software RAID in Linux kernel. Figure 1 shows the write path of RAID 5 in the mdraid layer. The minimum data unit for RAID operations is *stripe unit* (S-Unit), whose typical size is 4 KB. S-Units of the same address offset in all member disks are grouped as *stripe head* (S-Head), which are managed by *Stripe* data structure in mdraid. *Stripe* records the states of S-Head (e.g., read waiting and computation completed). The write procedure in mdraid can be described as follows. When a write request arrives in mdraid, it will firstly be sliced into S-Units (①). Next, to prepare for data processing, S-Units of the same offset then request for a *Stripe* via *get_active_stripe* function (②). If the number of S-Units in a *Stripe* does not equal to the length of an S-Head, a read request will be sent to the underlying block device for the missing S-Unit (③a). Afterward, the CPU threads calculate the parity codes (③b). Once the calculation completes, the S-Head will be sent to the storage device (④a) and finally the *Stripe* will be recycled (④b). To prevent multiple threads from competing for the same *Stripe*, Linux uses few global locks to guarantee the exclusive allocation of the *Stripes*.

Write hole. When a power failure occurs in the process of chunk write, the chunk being written to the storage becomes an uncertain value. During system reboot, mdraid is unable to locate and fix the write faults, which shatters the fault tolerance of RAID. This phenomenon is referred to as write hole [1]. To address this issue, mdraid employs a bitmap mechanism. In detail, a group of S-Heads are clustered as *stripe block* (S-Block). An S-Block typically covers address range of 64 MB on each disk. mdraid maintains a table of counters, each mapping to a specific S-Block. The counter records the number of S-Heads in an S-Block that are being written. This table will be flushed back to the member disks in batches by a daemon process and stored as a bitmap. During the recovery procedure, we can scan the bitmap to figure out which S-Block should be recalculated to synchronize data and parity. mdraid employs a single global lock to avoid the competition in counter updates.

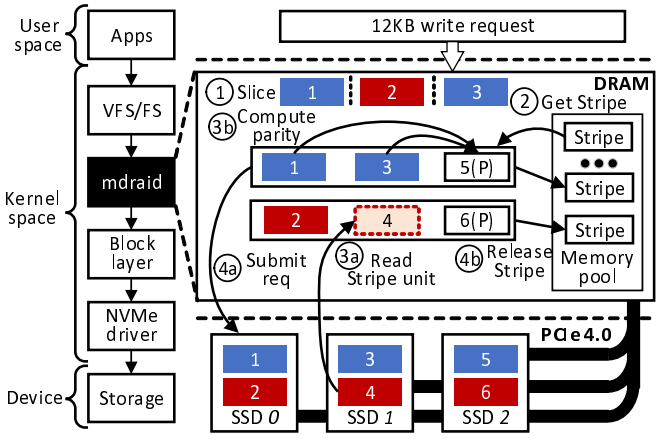
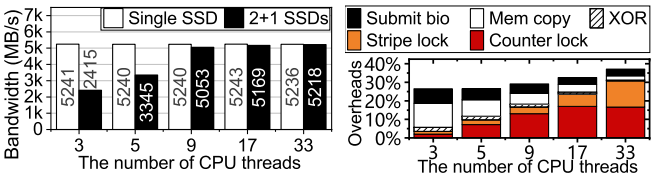


Fig. 1: Write path of RAID 5 in mdraid layer.



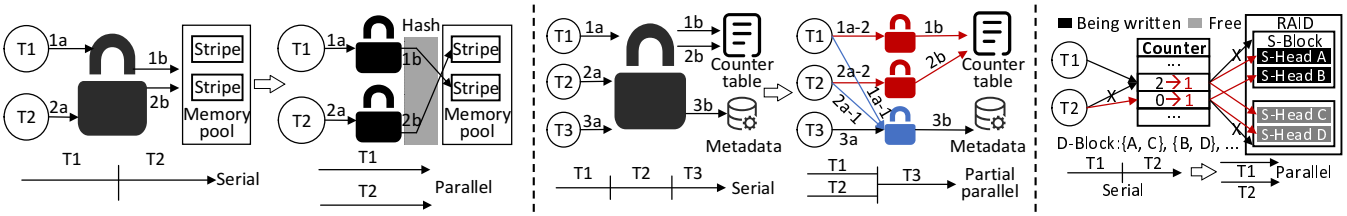
(a) Bandwidth of sequence write. (b) Overhead breakdown.

Fig. 2: Performance and overhead analysis of mdraid.

Motivation. To better understand the performance of mdraid, we take an experiment on the real PCIe 4.0 SSD arrays. Figure 2a shows the write throughput of RAID that consists of three SSDs. We vary the CPU threads from 3 to 33. The write bandwidth of RAID increases as we put more CPU threads for computation. However, it cannot exceed the bandwidth of a single SSD. We further analyze the CPU cost of mdraid with different numbers of threads, which is shown in Figure 2b. We categorize the cost into `Counter lock`, `Stripe lock`, `Submit bio`, `Mem copy`, and `XOR`. `Counter lock` and `Stripe lock` represent for the time consumed by the lock procedure of counters and Stripes, respectively. `Mem copy` and `Submit bio` are the time of bio preparation and submission to drivers. `XOR` is the time of parity computation. The overhead of the lock mechanisms (including `Counter lock` and `Stripe lock`) only accounts for 3.5% of the total I/O access time when employing 3 CPU threads. Nevertheless, it reaches 30.8% when using 33 threads.

III. ScalaRAID Design

Fine-grained locks. mdraid introduces a rudimentary lock mechanism to prevent multiple CPU threads from preempting the *Stripe* allocation, which imposes huge CPU overhead. Figure 3a shows our solution to resolve the lock issues. We increase the number of Stripe locks and interleave CPU threads to access different Stripe locks by leveraging a hash algorithm. Our design allows different threads (cf. `T1` and `T2` in Figure 3a) to run in parallel and improves the overall RAID throughput while reducing request completion time in a write burst by omitting the collision penalty. Once a *Stripe* has been



(a) Optimization of Stripe lock. (b) Optimization of counter lock. (c) Optimization of D-Block.

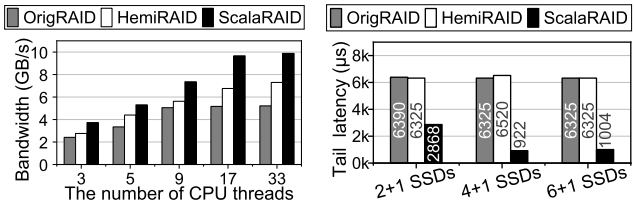
Fig. 3: Design details of ScalaRAID.

successfully acquired, the CPU thread continues to update the values in the counter table. As mdraid employs only a single spin lock to manage all the accesses to the counter table, all the CPU threads should be serialized in front of the counter table. As shown in Figure 3b, when both CPU threads `T1` and `T2` need to modify the counter table, they compete for the spin lock. `T2` has to wait until `T1` finishes its access (`1a` and `1b`) and releases the lock. To address this issue, we split the counter table into multiple segments and assign a dedicated lock to each segment. The counters in the counter table, which map to neighboring S-Blocks, are interleaved across different segments. Thus, CPU threads that target different segments can acquire the locks simultaneously, thereby improving the thread-level parallelism. For example, in Figure 3b, `T1` and `T2` can acquire different counter locks and update the counters in parallel. Note that the counter lock is also used to protect the metadata update of the counter table (cf. `T3`). However, modifying both the counter values and their metadata simultaneously can result in memory faults. We observe that mdraid rarely updates the metadata of the counter table. Thus, we employ a readers-writer lock mechanism to protect the metadata. Specifically, the reader locks can be owned by multiple CPU threads while the writer lock is exclusively held by a single thread. Before updating the metadata, the CPU thread acquires the writer lock (cf. 3a in Figure 3b). Otherwise, if the CPU threads need to revise the counter values, they apply for the reader locks and the counter locks successively.

Distributed blocks. In mdraid, S-Block by default covers an address range of 64 MB. In other words, a counter needs to record and manage such a wide range of address space. This design is optimized for large-size write requests but may harm small-size write requests. Figure 3c shows an example. Let's suppose that `T1` and `T2` need to modify different S-Heads `A` and `B` in the same S-Block. They then contend for modifying the same counter, which results in hanging up `T2`. Note that our multi-lock counter table allows multiple threads to access different counters simultaneously. It cannot prevent the congestion that targets the same counter. To tackle the challenges imposed by the traditional S-Block design, we propose a new data structure, called *Distributed Block* (D-Block), which is shown in Figure 3c. D-Block consists of multiple S-Heads. In contrast to S-Block, the S-Heads in D-Block are spread across different locations in the SSDs via a configurable hash function rather than mapping to a continuous SSD space. The default hash function takes the offset of S-Head as input and outputs $\lceil \log(Size/Dsize) \rceil$ rightmost bit(s) of the input offset, where *Size* and *Dsize* are the capacity of member disk and cover ranges of D-Block, respectively. Thus, S-Heads in the same D-Block are dispersed uniformly across the whole space. While a D-Block maintains a cover range of 64 MB, sequential write requests are shuffled to access different D-Blocks. By doing so, ScalaRAID can effectively reduce the counter preemption.

IV. Evaluation

Experiments. We conduct the experiments on a server that consists of a 52-thread processor and 128 GB DDR4 memory. We employ Linux v5.11.0 as the default kernel. We use mdadm to create RAID from up to seven 1TB Samsung 980Pro SSDs. We use fio to evaluate the performance of different RAID systems. In fio, we set the iodepth to 32 and employ libaio. We implement three different RAID systems. (1) `OrigRAID`: adopting the default configurations of mdraid; (2) `HemiRAID`: based on `OrigRAID`, we increase the number of Stripe locks to 128; (3) `ScalaRAID`: based on `HemiRAID`, we equip every counter with a counter lock and employ our D-Block. Considering



(a) Write throughput. (b) Tail latency.

Fig. 4: Performance comparison.

that the evaluated RAID system consists of several 1TB SSDs, `ScalaRAID` totally requires 16,384 counter locks.

Performance. Figure 4a shows full-stripe sequential write bandwidth of the evaluated RAID systems. As the number of CPU threads increases, the write bandwidth of `OrigRAID` gradually increases. Nevertheless, such bandwidth saturates when the number of CPU threads reaches 9. `HemiRAID` outperforms `OrigRAID` by 30.8% and 39.9% when using 17 and 33 threads, respectively. This is because `HemiRAID` allows more threads to get *Stripe* simultaneously and thus processes multiple requests in parallel. `ScalaRAID` can further improve the bandwidth by 34.8%, on average. This is because `ScalaRAID` resolves the counter lock contention thereby maximizing the parallelism of request handling. Figure 4b illustrates the 99.99^{th} percentile write latency measured from different RAID systems. The 99.99^{th} percentile write latency of `OrigRAID` and `HemiRAID` are close to each other. This is because although `HemiRAID` reduces the time for threads to request *Stripes*, these threads are still blocked by the only one counter lock. `ScalaRAID`, on the other hand, increases the number of counter locks and employs a new data structure (D-Block) to mitigate the contention imposed by simultaneous counter updates. `ScalaRAID` also prevents the CPU threads from being blocked by the *Stripe*, which can minimize the software overheads. Therefore, the 99.99^{th} percentile latencies of `ScalaRAID` are reduced to only 44.9%, 14.6%, and 15.9% for RAIDs that consists of 2+1, 4+1, and 6+1 member SSDs, respectively.

V. Original Publication

S. Yi et al. 2022. ScalaRAID: Optimizing Linux Software RAID System for Next-Generation Storage. Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. *https://dl.acm.org/doi/abs/10.1145/3538643.3539740*

VI. Acknowledgement

This research is mainly supported by Peking University start-up package (7100603645) and NSFC Excellent Young Scientists Fund Overseas Program (8206100425). Dr. Jung is in part supported by NRF 2021R1AC4001773 and IITP 2021-0-00524 & 2022-0-00117, KAIST IDEC & Start-up (G01190015), Samsung HiPHER, and Samsung Research Grant (G01200447). Jie Zhang is the corresponding author.

References

[1] B. Hickmann and K. Shook, "Zfs and raid-z: The über-fs?" *University of Wisconsin–Madison*, 2007.

[2] mdraid layer, https://github.com/torvalds/linux/tree/master/drivers/md, 2022.

[3] S. Wang, Q. Cao, Z. Lu, H. Jiang, J. Yao, and Y. Dong, "{StRAID}: Stripe-threaded architecture for parity-based {RAIDs} with ultra-fast {SSDs}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 915–932.