ScalaRAID: Optimizing Linux Software RAID System for Next-Generation Storage

Shushu Yi[1,2], Yanning Yang[3], Yunxiao Tang[1], Zixuan Zhou[1], Junzhe Li[1], Chen Yue[3]
Myoungsoo Jung[4], Jie Zhang[1]
Computer Hardware and System Evolution Laboratory,
Peking University[1], Nanjing University[2],
Beijing University of Posts and Telecommunications[3], KAIST[4]

ABSTRACT

RAID has been widely adopted to enhance the performance, capacity, and reliability of the existing storage systems. However, we observe that the Linux software RAID (mdraid) suffers from its poor implementation of the lock mechanism. 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 99.99^{th} percentile latency by 85.4% compared to mdraid.

CCS CONCEPTS

• **Information systems → RAID**; • **Software and its engineering → Secondary storage**.

KEYWORDS

Solid State Drive, RAID, Operating System, Lock

1 INTRODUCTION

Over the past years, solid state drives (SSDs) have become the dominant storage media, which are widely adopted in diverse computing domains including datacenters [4, 22, 27, 29, 31], high-performance computers [24, 32, 34], and portable devices [3]. While SSDs exhibit their superiority over the traditional storage media in terms of throughput, latency, and power efficiency, they, unfortunately, suffer from the low reliability imposed by the NAND flash intrinsics [6, 11, 12] and the limited storage capacity. Redundant Array of Inexpensive Disks (RAID) technology [30] is a cost-efficient approach to address the aforementioned challenges. Specifically, RAID can mitigate the penalty of flash errors by introducing data redundancy. To extend the capacity of SSD storage in scale, RAID also groups multiple SSD devices as an array, which can deliver a uniform large storage space.

However, as the SSD technology has experienced significant technology shifts, the RAID technology is becoming the performance bottleneck of the future storage system that employs the next-generation SSDs. To be precise, the emerging PCIe 4.0 SSDs can increase their I/O bandwidth capability up to 7 GB/s [2, 33]. In contrast, most hardware RAID engines are designed for low-speed storage interface (i.e., SATA), whose maximum throughput is limited to 500 MB/s [1]. Linux software RAID, referred to as mdraid [26], 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 huge burden to the CPU. Consequently, the performance of mdraid, unfortunately, cannot scale as the number of CPU threads and SSD devices increase. Specifically, we set up an experiment to analyze the execution time breakdown of the storage software stack. Our evaluation results reveal that the overheads of the lock mechanism account for 30.8% of the total software delays (cf. Section 2.2 for more details). One may consider to remove the lock mechanism from the mdraid. However, the locks play a critical role in guaranteeing crash consistency and taking charge of data management.

Tackling the aforementioned challenges, we propose *ScalaRAID*, a *Scala*ble *RAID* system to aggregate the performance and capacity of the next-generation storage devices with low CPU cost [1]. Specifically, we employ multiple fine-grained locks rather than the traditional coarse-grained lock to protect the critical resources in mdraid, which can minimize the lock preemption. We also redesign the key data structures in RAID to mitigate the overheads imposed by the existing crash consistency mechanism. Furthermore, we scatter the entire address space in RAID thereby preventing the collision in metadata updates. With these proposed designs, ScalaRAID can maximize thread-level parallelism and reduce

HotStorage '22, June 27–28, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539740

[1]ScalaRAID is available at https://github.com/ChaseLab-PKU/ScalaRAID.

CPU suspension time caused by lock contention. Compared to the mdraid design in the Linux system, ScalaRAID improves the overall storage throughput by 89.4% and decreases the 99.99^{th} percentile I/O latency by 85.4%.

The main **contributions** of this work can be summarized as follows:

• *Deep analysis of mdraid:* We observe that a mdraid system that consists of multiple high-performance SSDs can only achieve similar performance as a single SSD. This is because mdraid cannot fully utilize the striping mechanism to improve the storage bandwidth. We then dig deeply into the CPU utilization and the software overheads. Our evaluation results reveal that the lock mechanism consumes up to 78× more CPU power than the parity computation. In other words, the root cause of the performance degradation becomes the lock mechanism rather than RAID's parity computation. This is because mdraid usually employs multiple CPU threads to accelerate the request processing. However, the CPU threads are serialized in front of the locks. To the best of our knowledge, this is the first study of lock issues in mdraid when one uses it with next-generation SSDs.

• *Fine-grained lock mechanism:* mdraid employs coarse-grained locks to manage the storage resources and guarantee crash consistency. The lock contention is not a serious issue when running a few CPU threads to serve software RAID services. However, the next-generation storage requires more CPU threads to process I/O requests simultaneously thereby maximizing the I/O bandwidth. This in turn imposes great burden to the existing lock mechanism. To address this, we propose a parallelism-aware lock mechanism. In particular, we refine the scope of lock management and increase the number of locks with minor overheads. We further split the storage resources into multiple segments and assign a fine-grained lock to each segment. This allows different threads to access the segments in parallel. Our lock mechanism improves the throughput by 39.9%, compared to mdraid.

• *Customized data structure to avoid collisions:* While our proposed lock mechanism allows parallel accesses to different segments, CPU threads can still be blocked from accessing the same segment owing to ill-designed data structure. To address this, we redesign the data structure in mdraid to reap the benefits from thread-level parallelism. Specifically, we customize different types of lock structures to manage data and metadata by carefully considering their own characteristics. We further scatter each segment across the entire address range in the storage such that CPU threads can be interleaved to access different segments. These designs further improve the performance by 34.8%, on average, compared to the existing software RAID system.

2 BACKGROUND AND MOTIVATION

2.1 RAID and Its Implementations

Array basics. Redundant Array of Inexpensive Disks (RAID) is a storage technology that combines many disks into one

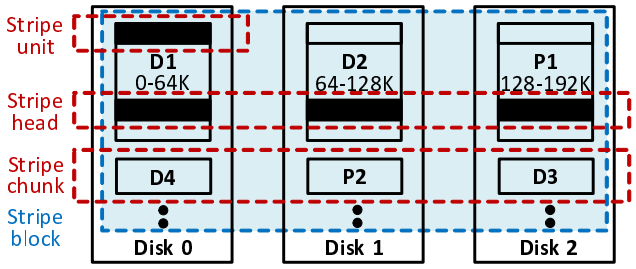


Figure 1: Data organization in a RAID system.

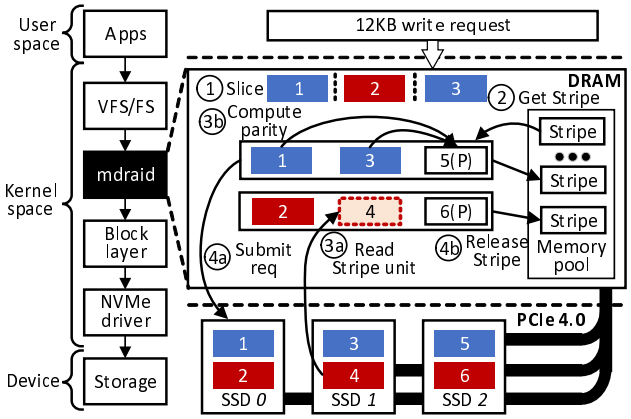
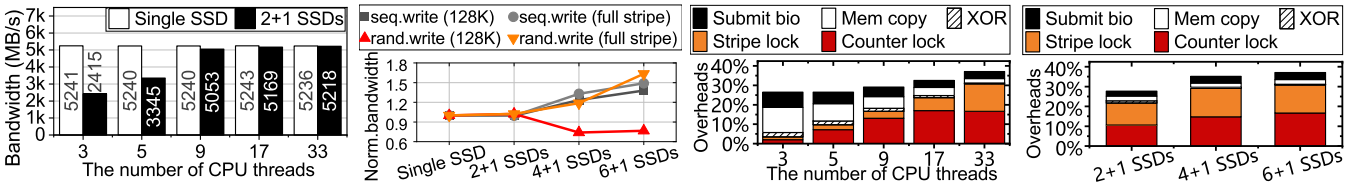


Figure 2: Write path of RAID 5 in mdraid layer

logical unit to satisfy the requirements of capacity, performance and reliability [13]. Figure 1 shows a RAID 5 system with three member disks. *Chunks* are the basic data units to manage all the member disks, whose sizes are typically 64 KB. Chunks of the same offset in different member disks are grouped as *stripe chunk* (S-Chunk). For a RAID 5 system of *N* disks, each S-Chunk has *N-1* data chunks and 1 parity chunk. The data in the parity chunk is calculated by XORing the remaining *N-1* data chunks. When one member disk fails, the missing chunk located in the failed member disk can be recovered by XORing the remaining chunks in the S-Chunk. Therefore, RAID 5 can deliver reliability guarantee. Note that this paper mainly focuses on RAID 5. Our designs can also be applied to other RAID levels (e.g., RAID 6) and schemes if they suffer from the same lock mechanism and ill-designed data structures as what RAID 5 in mdraid has. For simplicity, we use the terms "RAID" and "RAID 5" interchangeably.

Software RAID in Linux kernel. mdraid manages the underlying block devices (e.g., SSDs) meanwhile providing the I/O services to the upper-level filesystem. Figure 2 shows the write path of RAID 5 in the mdraid layer [26]. 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, cf. Figure 1), which are managed by *Stripe* data structure in mdraid (cf. Figure 2). *Stripe* records the states of S-Head (e.g., read waiting and computation completed) and acts as the cache of S-Head. The write procedure in mdraid can be described as



(a) Bandwidth of sequence write.

(b) Bandwidth variation with the number of SSDs.

(c) Overhead breakdown with different number of threads.

(d) Overhead breakdown with different number of SSDs.

Figure 3: Performance and overhead analysis of mdraid.

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**). Afterwards, 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*.

Crash consistency. In addition to the locks for *Stripe* allocation, mdraid requires extra locks to enable crash consistency. Specifically, 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 [19]. To address this issue, mdraid employs a bitmap mechanism to guarantee the crash consistency. In detail, a group of S-Heads are clustered as *stripe block* (S-Block, cf. Figure 1). 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.

2.2 Motivation

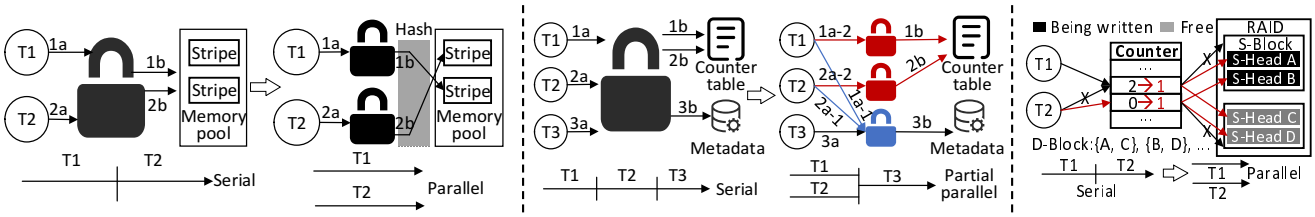
To better understand the impacts of mdraid on the system performance, we perform an experiment on the real PCIe 4.0 SSD arrays (cf. Section 4.1 for experiment details). Figure 3a shows the write throughput of RAID that consists of three SSDs. We vary the CPU threads from 3 to 33. The write bandwidth of RAID increases as we put more CPU threads for computation. However, it cannot exceed the bandwidth of a single SSD. Figure 3b shows the peak performance of RAID that employs different numbers of member SSDs. Increasing the number of member SSDs can slightly improve the overall throughput. For example, mdraid of 7 member SSDs only exceeds the performance of a single SSD by 63.5%. This indicates that mdraid cannot fully reap the benefits from the striping mechanism. We further analyze the CPU cost of mdraid with different numbers of threads and member SSDs, which is shown in Figure 3c and 3d. 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. Lastly, `XOR` is the time of parity computation. The overheads of the lock mechanisms (including `Counter lock` and `Stripe lock`) only account for 3.5% of the total I/O access time when employing 3 CPU threads. Nevertheless, the overheads reach 30.8% when using 33 threads, which is 78 times more than `XOR`. In this work, we primarily focus on mitigating the write penalty imposed by the lock mechanism. Note that the lock mechanism imposes near-zero overheads for read I/O requests as read request handling is slightly different from the write one. This is because, in most cases, read requests can be easily handled by *chunk_aligned_read()* function without using the aforementioned locks.

3 SCALARAID DESIGN

To mitigate the aforementioned overheads, ScalaRAID manages the *Stripes* and the counter table with multiple fine-grained locks (§3.1). ScalaRAID further prevents multiple CPU threads from contending for the same data by employing a new data structure to shuffle these accesses (§3.2).

3.1 Multiple Locks, Not One

Owing to the constraints of DRAM capacity, the number of *Stripes* in mdraid is usually limited (e.g., 256). Therefore, mdraid introduces a lock mechanism to prevent multiple CPU threads from preempting the *Stripe* allocation. However, the rudimentary lock mechanism in mdraid leads to a large number of CPU threads blocked in the process of requesting *Stripes*. This introduces the penalty of Stripe lock. Figure 4a shows our solution to resolve the lock issues. Our



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

Figure 4: Design details of ScalaRAID.

key insight is that we can employ multiple locks to manage different *Stripes* separately. Specifically, we increase the number of Stripe locks and interleave CPU threads to access different Stripe locks by leveraging a hash algorithm. This hash algorithm takes targeted page number p as input and outputs $p\&(s-1)$, where s and "&" are the number of Stripe locks and the bit-wise AND operation, respectively. Our design allows different threads (cf. T1 and T2 in Figure 4a) to run in parallel and improves the overall RAID throughput while reducing request completion time in an I/O write burst by omitting the collision penalty.

Unfortunately, simply increasing the number of Stripe locks cannot thoroughly resolve the lock issues. Specifically, once a *Stripe* has been 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 4b, 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 break the bound of serialization, we propose a multi-lock counter table. Our key insight is that different CPU threads may access different counters separately. Thus, our ScalaRAID splits the counter table into multiple segments and assigns 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. Figure 4b shows an example of our design (i.e., red arrows). CPU threads T1 and T2 can acquire different counter locks and update the counters in parallel.

It is worth noting that the counter lock is also used to protect the metadata update of the counter table. For example, when we need to shrink the storage space of a RAID system, the corresponding entries within the counter table should be deleted (cf. T3 in Figure 4b). 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 [15] to protect the metadata, which can reap the most benefits from this condition. 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 4b). Otherwise, if the CPU threads need to revise the counter values, they apply for the reader locks and the counter locks successively. Note that our work concentrates on mitigating the lock overheads (e.g., counter locks used by the bitmap mechanism). Our design of the fine-grained locks does not affect the correctness of the existing crash consistency mechanism.

3.2 Distributed Blocks, Not Centralized

In MD, S-Block by default covers an address range of 64 MB. In other word, 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 4c 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.

One possible solution is to reduce the cover range of S-Blocks thereby reducing the chances of access collisions. For example, we can adjust an S-Block to cover only a single S-Head (4KB). However, as a trade-off, such design can introduce huge memory consumption in the kernel. Specifically, mdraid needs to allocate a 2-byte memory space for each counter. RAID of multiple 2TB SSDs requires 1 GB memory space to accommodate the counter table, which imposes huge overheads to the system. An alternative solution is to place the counter table in the SSDs. mdraid can allocate memory space to buffer the counter table via mmap [28]. However, this in turn introduces the overheads of page faults [9]. Considering that random writes usually modify different counters frequently, frequent page faults can significantly increase the tail latency of write completion. 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 4c. 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 $\lceil \log(Size/Dsize) \rceil$

Number of SSDs	2+1	4+1	6+1
full stripe	128KB	256KB	384KB

Table 1: Full stripe size of different RAID.

Hardware		Software		Data structure	
CPU	Intel Xeon 5320	OS	Ubuntu	S-Head	4KB
	1 x 26 cores/2.2 GHz		20.02	S-Chunk	64KB
	with hyperthreading	Linux kernel	v5.11.0	S-Block	64MB
Mem.	Samsung			D-Block	64MB
	8 x 16GB/DDR4	Test tools		Number of locks	
SSD	Samsung 980 Pro	FIO	v3.16	OrigRAID	8 Stripe lck./1 counter lck.
	up to 7 x 1TB	Perf	v5.11	HemiRAID	128 Stripe lck./1 counter lck.
	R/W: 7000/5000 MB/s	Mdadm	v4.1	ScalaRAID	128 Stripe lck./16,384 counter lck.

Table 2: System configurations.

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.

4 EVALUATION

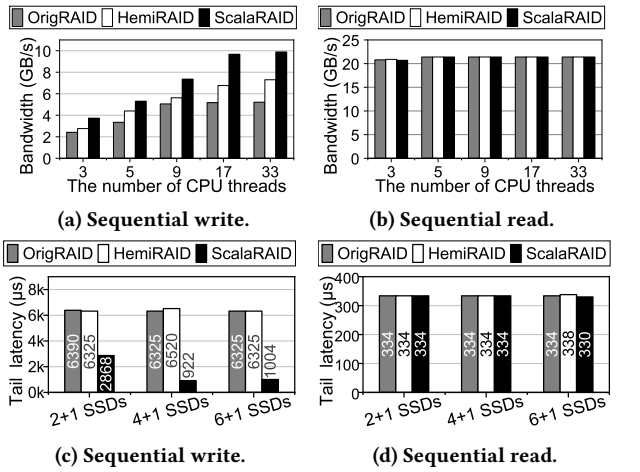
4.1 Experimental Setup

Methodology. We conduct the experiments on a server that consists of a 26-core processor and 128 GB DDR4 memory. We employ Linux v5.11.0 [23] as the default kernel in this evaluation. We also use mdadm v4.1 [25] to create RAID from up to seven 1TB Samsung 980Pro SSDs [2]. In this experiment, we use 2+1 SSDs (i.e., 3 SSDs, "n+1" is the usual expression to emphasize the parity) as default. We use FIO v3.16 [5] to evaluate the performance of different RAID systems. In FIO, We set the iodepth to 32 and employ an asynchronous I/O engine (libaio [7]). Besides, we use Perf v5.11 [17] to record the CPU usage of the OS software stack. We configure the I/O size as 4 KB in the latency evaluation while the I/O sizes are adjusted to full stripe in bandwidth evaluation (cf. Table 1). It is worth noting that we employ a single thread as the mdraid daemon (cf. Section 2.1). Additionally, we employ the same number of worker threads as FIO threads to maximize the performance of RAID, which is inspired by [21]. Worker threads do the same work as daemon (e.g., submitting bio to member disk). All the daemon threads, worker threads, and FIO threads are counted in the following evaluation. Table 2 lists the important configurations in our experiments.

RAID systems. 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 (cf. Section 3.1) and employ our D-Block (cf. Section 3.2). Considering that the evaluated RAID system consists of several 1TB SSDs, `ScalaRAID` totally requires 16,384 counter locks. We summarize the RAID system configurations in Table 2.

4.2 Performance Comparison

Bandwidth. Figures 5a and 5b show full stripe sequential write and read bandwidths of the evaluated RAID systems,



(a) Sequential write. (b) Sequential read.

(c) Sequential write. (d) Sequential read.

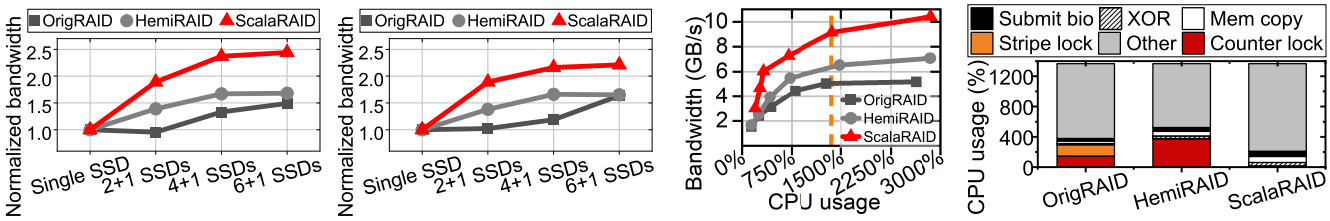
Figure 5: Performance comparison.

respectively. 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, compared to `HemiRAID`. This is because `ScalaRAID` resolves the counter lock contention thereby maximizing the parallelism of request handling. On the other hand, `ScalaRAID` achieves almost the same read performance as `OrigRAID` and `HemiRAID`. This is because `ScalaRAID` has minor impact on the read path of mdraid (cf. Section 2.2).

Latency. Figures 5c and 5d illustrate the 99.99^{th} percentile write and read latencies measured from different RAID systems, respectively. 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 (cf. Section 3.1). `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. In contrast, `ScalaRAID`, `HemiRAID`, and `OrigRAID` have similar 99.99^{th} percentile read latency. The subtle performance differences can be considered as an experiment noise. This performance behavior is similar to our observation in the sequential read throughput.

4.3 Scalability and Overhead Analysis

Scalability. We further measure the performance scalability of different RAID systems by increasing the number of



(a) Sequential write bandwidth. (b) Random write bandwidth. (c) Software efficiency. (d) CPU usage of RAID systems.

Figure 6: Analysis of scalability and CPU overheads.

member SSDs. The evaluation results are shown in Figures 6a and 6b. For full stripe sequential writes, the bandwidth of ScalaRAID exceeds that of OrigRAID to 1.9×, 1.8×, and 1.6×, respectively, in 2+1, 4+1, and 6+1 SSDs. Compared to OrigRAID, ScalaRAID also improves the performance by 67.2%, on average, for random full stripe writes. This is because ScalaRAID successfully parallelizes request handling and thus reaps higher bandwidth from more drives. In other word, ScalaRAID is more suitable for RAID consisting of a large number of SSDs than OrigRAID.

Overheads. To demonstrate the efficiency of different RAID systems, we measure the throughput under different CPU usages. The collected statistics are shown in Figure 6c. Compared to OrigRAID, HemiRAID and ScalaRAID can achieve 25.5% and 74.7% performance improvement under the same CPU usages, respectively. This indicates that HemiRAID and ScalaRAID can better utilize the CPU threads. Figure 6d shows usage breakdown of the orange line in 6c. Compared to OrigRAID, HemiRAID can reduce the Stripe lock overheads by 97.8%. However, as HemiRAID employs only a simple lock for the counter table, the counter lock becomes the performance bottleneck. Compared to HemiRAID, ScalaRAID designs new lock mechanisms for both the *Stripes* and the counter table. Therefore, ScalaRAID reduces the time cost of Stripe lock and counter lock to 3.2% and 0.5%, respectively. Nevertheless, there still exist other sophisticated management mechanisms in the storage software stack (e.g., the complex state machines to manage *Stripes*), which impose significant CPU overheads (84.6% in ScalaRAID). They become the major obstacles in scaling up the performance of ScalaRAID linearly. It is worth noting that ScalaRAID has minor modification to the existing system software, which exposes few memory cost. Specifically, as a single lock is 4 B, our proposed Stripe locks take up 512 B memory space. For ScalaRAID with multiple 1 TB member SSDs, 16,384 counter locks consume 64 KB memory space in total, which is negligible compared to a 128 GB memory system.

5 RELATED WORK

Crash consistency. In addition to the bitmap mechanism, there exist multiple approaches to guarantee crash consistency in RAID. [18] leverages journaling to record the transactions of RAID. Once an unclean shutdown [10] occurs, [18] replays these transactions to restore the data. [8] protects crash consistency by employing the Copy-on-Write semantic. As this design does not modify data in place, [8] can simply restore the parity after failures. Among the aforementioned solutions, the bitmap mechanism exhibits the best performance in terms of bandwidth and latency. Therefore, we select the bitmap mechanism as the baseline of ScalaRAID.

RAID optimization. Multiple prior works [14, 16, 20, 24] focus on mitigating the penalty of updating parity codes in RAID. Specifically, [16, 24] propose accelerating the parity update by offloading the computation tasks to GPUs. [14, 20] utilize NVRAM to cache data. It postpones the parity update until the cached data can be merged into full stripe. By doing so, it can eliminate the read overheads before parity computation. In contrast, ScalaRAID addresses the performance issues imposed by the lock mechanism in software RAID. Our design does not require any hardware modification. Note that ScalaRAID is orthogonal to the aforementioned approaches.

6 CONCLUSION

The lock mechanism has become the performance bottleneck of mdraid. In this work, we propose *ScalaRAID*, which achieves scalable performance of the storage management software by relaxing the constraints imposed by the lock mechanism. ScalaRAID successfully aggregates the performance and capacity of the next-generation storage with low CPU cost.

ACKNOWLEDGEMENT

The authors thank Prof. Guangyan Zhang of Tsinghua University for shepherding their paper. This research is mainly supported by Peking University start-up package (7100603645). 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] Samsung 870EVO. 2021. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/870-evo-sata-2-5-ssd-250gb-mz-77e250b-am/.

[2] Samsung 980Pro. 2020. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for {SSD} Performance. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

[4] David G Andersen and Steven Swanson. 2010. Rethinking flash in the data center. *IEEE micro* 30, 04 (2010), 52–54.

[5] Jens Axboe. 2019. Flexible I/O Tester. https://github.com/axboe/fio.

[6] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)* 6, 2 (2010), 1–22.

[7] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*. 371–386.

[8] Jeff Bonwick and Bill Moore. 2008. ZFS: The Last Word in File Systems. https://www.snia.org/sites/default/orig/sdc_archives/2008_presentations/monday/JeffBonwick-BillMoore_ZFS.pdf.

[9] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".

[10] Neil Brown. 2001. Software RAID in 2.4. In *Proceedings of linux. conf. au, Sydney, Australia*.

[11] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. 2015. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 551–563.

[12] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review* 37, 1 (2009), 181–192.

[13] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. 1994. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)* 26, 2 (1994), 145–185.

[14] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1683–1694.

[15] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. 1971. Concurrent control with "readers" and "writers". *Commun. ACM* 14, 10 (1971), 667–668.

[16] Matthew L Curry, H Lee Ward, Anthony Skjellum, and Ron Brightwell. 2010. A lightweight, gpu-based software raid system. In *2010 39th International Conference on Parallel Processing*. IEEE, 565–572.

[17] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[18] Timothy E Denehy, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. Journal-guided Resynchronization for Software RAID.. In *FAST*.

[19] Brian Hickmann and Kynan Shook. 2007. ZFS and RAID-Z: The Über-FS? *University of Wisconsin–Madison* (2007).

[20] Soojun Im and Dongkun Shin. 2010. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Comput.* 60, 1 (2010), 80–92.

[21] Nikolaus Jeremic, Helge Parzyjegla, and Gero Muehl. 2016. Improving random write performance in homogeneous and heterogeneous erasure-coded drive arrays. *ACM SIGAPP Applied Computing Review* 15, 4 (2016), 31–53.

[22] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. {FusionRAID}: Achieving Consistent Low Latency for Commodity {SSD} Arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 355–370.

[23] Linux kernel v5.11.0. 2021. https://www.kernel.org/doc/html/v5.11/.

[24] Aleksandr Khasymski, M Mustafa Rafique, Ali R Butt, Sudharshan S Vazhkudai, and Dimitrios S Nikolopoulos. 2012. On the use of GPUs in realizing cost-effective distributed RAID. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 469–478.

[25] Mdadm. 2018. http://www.kernel.org/pub/linux/utils/raid/mdadm/.

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

[27] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 177–190.

[28] mmap. 2021. https://man7.org/linux/man-pages/man2/mmap.2.html.

[29] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD failures in datacenters: What? when? and why?. In *Proceedings of the 9th ACM International on Systems and Storage Conference*. 1–11.

[30] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. 109–116.

[31] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 67–80.

[32] Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu, and Yong Chen. 2017. Ssdup: a traffic-aware ssd burst buffer for hpc systems. In *Proceedings of the international conference on supercomputing*. 1–10.

[33] Western Digital Black SN850. 2021. https://www.westerndigital.com/products/internal-drives/wd-black-sn850-nvme-ssd.

[34] Greg Wong. 2013. SSD market overview. In *Inside Solid State Drives (SSDs)*. Springer, 1–17.