Explorations and Exploitation for Parity-based RAIDs with Ultra-fast SSDs

SHUCHENG WANG, China Mobile (Suzhou) Software Technology Co., Ltd., China; Huazhong University of Science and Technology, China
QIANG CAO, Huazhong University of Science and Technology, China
HONG JIANG, Department of Computer Science and Engineering, University of Texas at Arlington, USA
ZIYI LU and JIE YAO, Huazhong University of Science and Technology, China
YUXING CHEN and ANQUN PAN, Tencent Inc., China

Following a conventional design principle that pays more fast-CPU-cycles for fewer slow-I/Os, popular software storage architecture Linux Multiple-Disk (MD) for parity-based RAID (e.g., RAID5 and RAID6) assigns one or more centralized worker threads to efficiently process all user requests based on multi-stage asynchronous control and global data structures, successfully exploiting characteristics of slow devices, e.g., Hard Disk Drives (HDDs). However, we observe that, with high-performance NVMe-based Solid State Drives (SSDs), even the recently added multi-worker processing mode in MD achieves only limited performance gain because of the severe lock contentions under intensive write workloads.

In this paper, we propose a novel stripe-threaded RAID architecture, StRAID, assigning a dedicated worker thread for each stripe-write (one-for-one model) to sufficiently exploit high parallelism inherent among RAID stripes, multi-core processors, and SSDs. For the notoriously performance-punishing partial-stripe writes that induce extra read and write I/Os, StRAID presents a two-stage stripe write mechanism and a two-dimensional multi-log SSD buffer. All writes first are opportunistically batched in memory, and then are written into the primary RAID for aggregated full-stripe writes or conditionally redirected to the buffer for partial-stripe writes. These buffered data are strategically reclaimed to the primary RAID. We evaluate a StRAID prototype with a variety of benchmarks and real-world traces. StRAID is demonstrated to outperform MD by up to 5.8 times in write throughput.

CCS Concepts: • **Information systems** → **Disk arrays**; **Storage management**; • **Software and its engineering** → *Software architectures;*

Additional Key Words and Phrases: RAID systems, multi-thread scheduling, solid-state drive

This work was supported in part by National Key Research and Development Program of China No. 2018YFA0701800, NSFC No.62172175, NSFC No.61821003, the US National Science Foundation Grant CNS-2008835, and the Key Research and Development Project of Hubei No.2022BAA042.
Authors' addresses: S. Wang, China Mobile (Suzhou) Software Technology Co., Ltd., China; and Huazhong University of Science and Technology, China; e-mail: wsczq@hust.edu.cn; Q. Cao (Corresponding author), Z. Lu, and J. Yao, Huazhong University of Science and Technology, China; e-mails: {caoqiang, luziyi, jackyao}@hust.edu.cn; H. Jiang, Department of Computer Science and Engineering, University of Texas at Arlington, US; e-mail: hong.jiang@uta.edu; Y. Chen and A. Pan, Tencent Inc., China; e-mails: {axingguchen, aaronpan}@tencent.com.

1553-3077/2024/01-ART6 $15.00
https://doi.org/10.1145/3627992

ACM Reference format:
Shucheng Wang, Qiang Cao, Hong Jiang, Ziyi Lu, Jie Yao, Yuxing Chen, and Anqun Pan. 2024. Explorations and Exploitation for Parity-based RAIDs with Ultra-fast SSDs. *ACM Trans. Storage* 20, 1, Article 6 (January 2024), 32 pages.
https://doi.org/10.1145/3627992

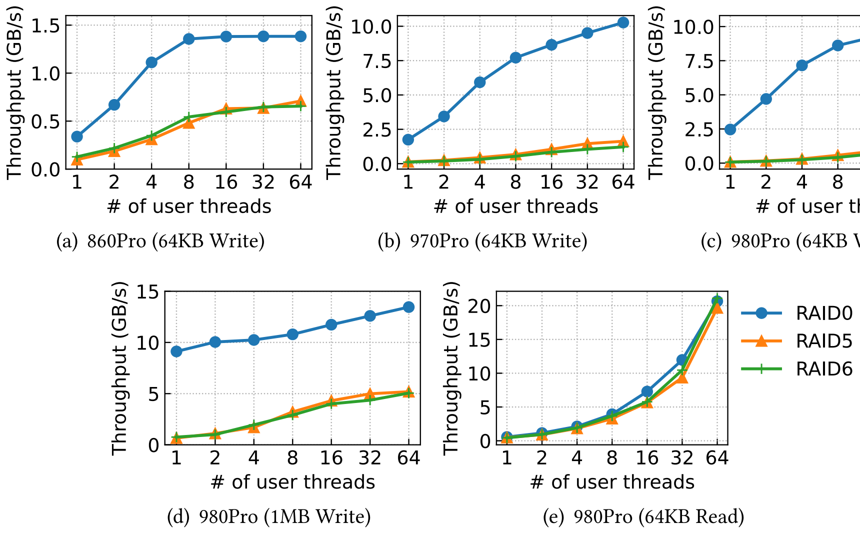
1 INTRODUCTION

The advent of ultra-fast storage devices such as NVMe-based **Solid-State Drives (SSDs)** and **Non-volatile Memory (NVM)** with GB/s-level I/O bandwidth has dramatically narrowed the performance gap between memory and storage. **Redundant Array of Inexpensive Disks (RAID)** [56] can combine multiple such high-performance storage devices to further promote the overall storage performance, reliability, and capacity simultaneously. Many empirical studies [10, 18, 35] including distributed datacenter storage systems [49, 72] and enterprise storage systems [48] report that SSD drivers exhibit reliability problems in that more than 20% of SSDs develop uncorrectable errors in a four-year period [58]. Therefore, parity-based RAIDs composed of ultra-fast SSDs have become attractive storage systems for modern data-intensive applications in supercomputing [55], big data analytics [25, 64], machine learning [7], enterprise storage [48], and cloud services [1, 32, 38, 45, 61].

HDD-based RAIDs have been extensively studied since 1988 [56]. In the literature, recent studies focus on SSD-based RAID and **All-Flash-Array (AFA)**, with efforts to reduce SSD write-penalty by mitigating parity update [8, 15, 68], reduce garbage-collection induced performance jitter [21, 33, 42], and optimize AFA using declustering RAID approach to balance load within devices and reduce tail-latency [28, 77]. Existing RAID I/O handling techniques generally adopt a centralized stripe-processing architecture following a classic principle that trades more fast-CPU-cycles (e.g., scheduling algorithms) for fewer slow-I/Os. Nonetheless, the question of whether such RAID architecture can fully exploit the power of emerging fast storage remains unanswered.

We experimentally measure the actual performance of **Multiple-Disk (MD)** [46], the most popular and mature software RAID integrated into the Linux kernel for over two decades. We conduct MD running on 6 NVMe-based SSDs with 64 user threads (i.e., issuing block requests) and 64 workers threads (i.e., handling RAID stripe-writes), with the experiment environment summarized in Tables 1 and 2. The results are shown in Figure 1 (detailed in Section 3.1). With RAID0 (non-parity RAID-level), MD obtains an expected performance that approaches the aggregate raw I/O capacity of the underlying SSDs, i.e., 20GB/s and 14GB/s for read and write throughputs, respectively. However, MD falls far short of the expectation in write performance in RAID5 and RAID6 (parity-based RAID-levels). Specifically, the write throughput of RAID5 is below 2.2GB/s under partial-stripe writes and below 5.2GB/s under full-stripe writes, which are only about 1/7 and 1/3 of that of RAID0, respectively. Although parity-RAIDs introduce extra parity-compute overheads, our measured XORing rate on a CPU core can reach up to 29GB/s [26], which is clearly not the bottleneck.

Through profiling (detailed in Section 3.2), we experimentally uncover that the write inefficiency of parity-based RAID comes from a centralized stripe-handling architecture in the legacy MD. Specifically, a worker thread using shared data structures (e.g., stripe-list) handles write requests by efficiently collaborating with user threads, XORing threads, and device I/O threads. For HDDs and slow SSDs, this one(worker thread)-for-all(stripe) architecture utilizes fast CPU sufficiently by postponing stripe-writes to absorb more requests for reducing actual I/Os. However, a single worker thread is upper-bounded in its processing capability that fails to keep up with the fast storage. The latest MD introduces a multi-worker mechanism, referred to as the N-for-all processing



(a) 860Pro (64KB Write) (b) 970Pro (64KB Write) (c) 980Pro (64KB Write)

(d) 980Pro (1MB Write) (e) 980Pro (64KB Read)

Fig. 1. The throughput of Linux software RAIDs on three-types of SSDs under varying number of user threads.

model, but achieves a limited performance gain due to severe lock contention on the centralized data structures, especially for handling partial-stripe writes.

In this paper, we propose a novel **stripe-threaded architecture**, called **StRAID**, for parity-based RAIDs built on ultra-fast storage devices such as NVMe-based SSDs. To address the architectural drawback of the existing software RAID (MD), StRAID employs a one(worker)-for-one(stripe) model, thus significantly reducing the number of stripe-states and their lock-based checks. Furthermore, StRAID adopts a fine-grained stripe-level lock, substantially mitigating contentions on shared data structures. To tame the notoriously performance-degrading partial-stripe writes, StRAID proposes a two-stage stripe submission mechanism and a two-dimensional SSD write buffer distributed across the RAID member disks. StRAID first aggregates incoming stripe-associated writes into full-stripe in memory within a limited time window, which opportunistically filters requests with sequential or in-place update patterns. Afterward, the aggregated full-stripes are flushed into the underlying primary RAID and the partial-stripe writes are written into the Log-Buffer. The Log-Buffer employs a two-dimensional log data-layout and a parallel I/O processing model to fully exploit SSDs' parallelism. Finally, the buffered data are strategically written back to the primary RAID. Fundamentally, StRAID effectively exploits stripe-based data parallelism while mitigating intra-stripe conflicts between the dedicated stripe worker thread and other threads. StRAID leverages the power of multicore CPUs that offer sufficient threads to fully unleash the superior IOPS provided by fast SSDs.

The main contributions of this paper are as follows.

- We experimentally observe a serious write inefficiency problem in the current MD when parity-based RAID is running on ultra-fast storage. We further reveal that the root cause is the centralized one-for-all stripe-handling architecture.
- We propose a novel parity-RAID processing architecture, StRAID, guided by a stripe-threaded one-for-one model and a two-stage stripe submission mechanism to unleash the full performance potentials of modern hardware while improving partial-stripe writes on parity-based RAIDs.

- We present an SSD-based write buffer seamlessly embedded with the StRAID to absorb infamous partial-stripe writes of RAID. The write buffer employs multi-log data-layout and a parallelized I/O processing model to reap the performance of multiple SSDs.
- We prototype and evaluate StRAID with a variety of benchmarks and real-world workloads. StRAID consistently outperforms MD by up to 5.8 times in write throughput without affecting the read performance while reducing CPU utilization.

The rest of the paper is organized as follows. Section 2 presents the background for RAID. Section 3 analyzes the performance behaviors of Linux software RAID (MD) and motivates the StRAID design. Section 4 describes StRAID's design. We evaluate StRAID in Section 6 and describe related works in Section 7. Section 8 concludes this paper.

2 BACKGROUND

2.1 RAID Systems

Redundant Array of Inexpensive Disks (RAID) [56] is a classic system-level approach that combines multiple disks to improve performance, reliability and capacity simultaneously. Over the past decades, RAID has been used ubiquitously to construct and manage efficient storage servers, distributed storage [4, 54], and cloud storage [1, 38] from within and/or among storage devices.

The RAID architecture is categorized into various RAID levels based on the amount of redundancy and how redundancy is incorporated, including non-parity RAIDs (e.g., striping-only RAID0 and mirroring-only RAID1) and parity RAIDs (e.g., RAID5 and RAID6 that can tolerate one and two disk failures respectively). RAID can be implemented in either software or dedicated hardware (e.g., I/O controllers or firmware) to offer the block-addressable volume. A common N-disk RAID internally consists of multiple stripes, each of which comprises user data chunks and their corresponding parity data chunks across N disks according to an algorithmic address-mapping method. Normal reads without disk failure are directly decomposed to their constituent chunk I/Os served by the underlying disks. Normal writes in non-parity RAIDs behave like normal reads without accessing parity chunks.

Normal writes in parity-based RAIDs need extra parity generation, update, or construction operations. For a full-stripe user write where all data chunks of a stripe are written, the RAID system generates all new parity chunks at once, and then writes both data chunks and parity chunks into their corresponding disks. For a partial-stripe write where only a subset of the data chunks of a stripe are written, only after its constituent old data or parity chunks are read from the disks is the stripe updated and then written into the disks again, thus inducing numerous extra I/Os [8, 28]. This read-modify-write nature of partial-stripe writes makes them notoriously costly. When disks fail within the failure-tolerance range, the RAID transitions from its normal mode to a degraded mode to perform read, write, or resync operations.

2.2 Linux Software RAID

The Linux software RAID module, referred to as **Multiple-Disk (MD)** [46], is the most commonly used software RAID evolving with the Linux Kernel for over two decades. Currently, MD supports various RAID levels and RAID compositions. Non-parity-based RAIDs in MD perform an algorithmic block-to-chunk address mapping. For parity-based RAIDs, normal reads are similar to those in a non-parity-based RAID without parity operation. However, writes inevitably introduce several additional parity-generation/modification operations. Figure 2 shows the architecture of MD parity-based RAIDs and Figure 3 shows the workflow of their stripe-writes. The centralized data structure (stripe-cache) comprises inactive and handling stripe-lists, which maintain the metadata of the stripes (up to 256 by default). Each stripe has its own stripe_head containing stripe states

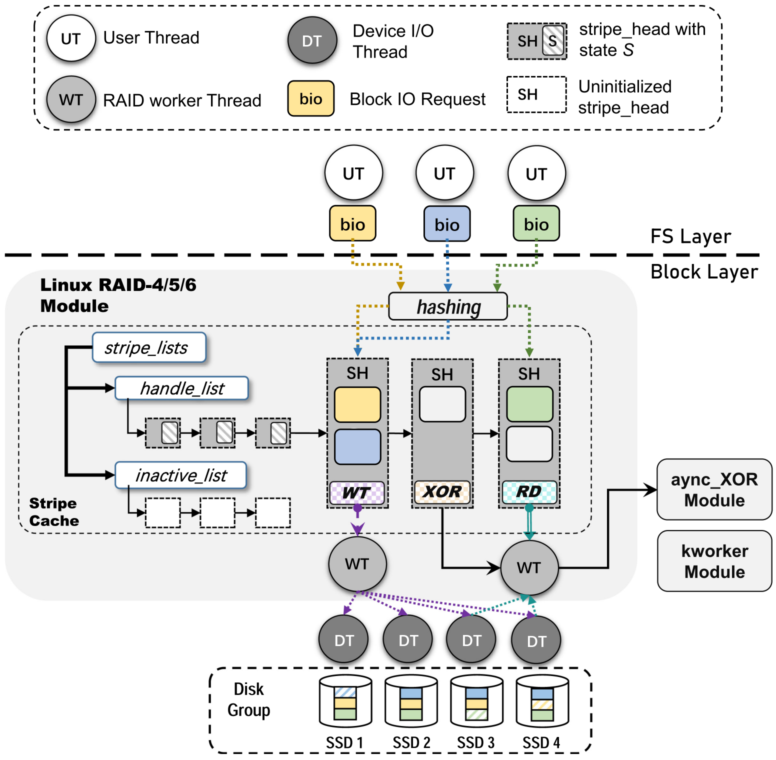


Fig. 2. Architecture of Linux MD parity-based RAID.

and device states (*Devs*). *Devs* contains a set of block request structures (*bios*) pointing to their buffered pages. Specifically, a stripe and its corresponding *Devs* have 28 and 27 states respectively that are used to precisely identify the handling states of this stripe. When a stripe is processed and cleared, its corresponding stripe_head will be transferred into the inactive_list.

MD handles stripe-writes using a state machine represented as a **directed acyclic graph (DAG)** [16]. As shown in Figure 3, a normal user write process can be divided into 5 consecutive stages: 1) **inserting/aggregating *bios* to a stripe (INS)**; 2) **reading data/parity chunks (RD)**; 3) **computing parity (XOR)**; 4) **writing data/parity (WT)**; and 5) **clearing stripe (CLR)**. Specifically, in the first stage ❶, **user threads (*UT*)** invoke *make_request()* to attach *bios* to their corresponding stripe_head structures. Afterwards, a daemon worker thread (*WT*), i.e., *RAID5d* in MD by default, handles all active stripe_heads in a circular manner with priority.

For a full-stripe write, MD skips the second stage. For a partial-stripe write, MD must introduce write-induced reads ❷, resulting in I/O amplification. More specifically, there are two stripe-updating schemes, **read-modify-write (RMW)** and **read-construction-write (RCW)** [28]. MD calculates the required number of disk-read I/Os of both RCW and RMW, selects the I/O-minimum approach, and launches the relevant disk-read I/Os. When a disk I/O thread (*DT*) completes the read, it sets a data-prepared flag to its *bios*. Afterwards, ❸ WT checks all the involved *bios* until prepared, and then launches a parity-calculation executed by other XORing threads. When WT verifies that the parity has been prepared, ❹ it invokes disk-write I/Os. ❺ WT finally validates the completed state and clears the stripe_head. Therefore, the write process orchestrates WT, UT, and DT threads via shared-state setting and checking.

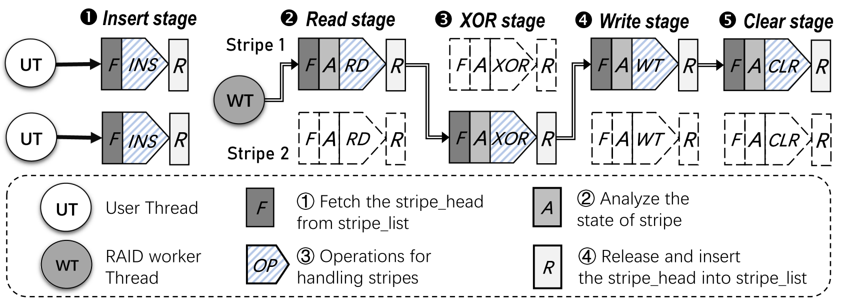

❶ Insert stage
❷ Read stage
❸ XOR stage
❹ Write stage
❺ Clear stage
UT
WT
Stripe 1
Stripe 2
F
INS
A
RD
XOR
WT
CLR
R
User Thread
RAID worker Thread
① Fetch the stripe_head from stripe_list
② Analyze the state of stripe
OP
③ Operations for handling stripes
④ Release and insert the stripe_head into stripe_list

Fig. 3. Stripe-write workflow of parity-based MD RAID.

WT handles each stage of a stripe-write in four steps, as described in Figure 3: ① getting a stripe_head from a stripe_list; ② analyzing the current state of this stripe and all its involved *bios*, to determine whether this stripe is still in-flight; ③ handling the stripe by launching a given operation (e.g., XOR) through executing DAG; and ④ updating the stripe state, inserting it back into a stripe_list and selecting the next stripe. The worker thread handling a stripe exclusively accesses shared data structures and stripe-states using multiple locks. For example, in step ④, WT exclusively modifies handle_list with a global device lock.

For HDD-based RAID, a disk I/O takes at least several milliseconds. Therefore, a WT in Linux MD has sufficient CPU-cycles to drive all stripe-writes. With the emerging SSDs that have 2-3 orders of magnitude lower I/O latency than HDD, MD also introduces a multi-worker mechanism [39, 40] that enables more numbers of functionally equivalent worker threads to process stripes concurrently, referred to as the N-for-all processing model.

3 ANALYSIS AND MOTIVATION

3.1 Understanding the Write Performance

Experiment Setup We start with measuring the MD performance in the RAID0, RAID5 and RAID6 levels running on three types of SSD devices, whose I/O characteristics are listed in Table 2. The platform configuration is shown in Table 1. The XORing throughput on a single CPU-core can reach up to 29GB/s. We deploy six SSD devices to construct parity-based RAIDs, that is, 5+1 RAID5 and 4+2 RAID6, respectively. The chunk size in all RAIDs is set to 64KB as default. We pin each **user thread (*UT*)** to a unique CPU-core and increase the number of UTs from 1 to 64. Each UT issues random 64KB-sized writes over 30 seconds. For parity-RAIDs, we invoke up to 64 extra RAID **worker threads (*WT*)**, and enlarge the stripe cache capacity from the default of 256 stripe_heads to 16K stripe_heads.

Write Inefficiency with Parity-RAID Figure 1 reports the throughput performance of MD. In all the cases, the write performance and scalability of the non-parity RAID0 far exceed those of parity-based RAID5 and RAID6. RAID0 achieves a write performance of about 1.4GB/s and 11GB/s peak throughput on 860Pro and 980Pro SSDs at 64 UTs, while RAID5 in the multi-worker mode achieves a peak write performance of lower than 0.72GB/s and 5.3GB/s, respectively. On 980Pro, the 64KB partial-stripe write throughputs of parity-RAIDs are below 2.1GB/s, which is only 1/7 of that of RAID0. Even for the 1MB full-stripe writes, the throughputs of RAID5 and RAID6 with 64 UTs are below 5.2GB/s and 5.3GB/s respectively, only about 38% of that of RAID0. It indicates that parity-RAIDs fall short of leveraging the write I/O performance of modern SSDs and the bottleneck on CPU processing is the main reason. We will show more details in the next section. Besides, normal reads of MD in all RAID levels are generally similar and scale well with the number of UTs.

Table 1. Evaluation Platform Specifications

Components	**Configurations**
Processor	Duel Socket Intel Xeon Gold 6328, 56 Cores, 128MB LLC
Memory	256GB 2666MHz DDR4
Operating System	Ubuntu 20.10 LTS with the Linux kernel version 5.13.0
MD controller	mdadm v4.2 2021-12-30

Table 2. Characteristics of Three Representative SSD Products

Device Types	**Device Modules**	**Capacity**	**Stable Write Thr. (MB/s)**	**Stable Read Thr. (MB/s)**	**Interfaces**
SATA SSD	Samsung 860 Pro	512GB	500	510	SATA
NVMe SSD	Samsung 970 Pro	512GB	2200	3200	PCIe 3.0
NVMe SSD	Samsung 980 Pro	1TB	2600	6900	PCIe 4.0

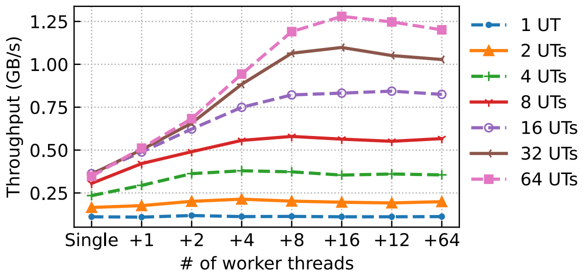


Fig. 4. Write throughput of MD RAID5 under the multi-worker mechanism.

We further analyze the write inefficiency of the multi-worker mechanism with RAID5 on six 980Pro SSDs. We invoke 64 UTs in either the single-worker (i.e., *Single*) mode or the multi-worker mode with the number of WTs varying from 1 to 64 (i.e., *+1W* to *+64W*). Figure 4 shows that the parity-based RAID gains limited benefits from the multi-worker mode. For example, MD with 8 more WTs has a write throughput improvement of 2.4× and 3.6× over the single-worker mode under 16 and 64 UTs, respectively. However, MD's performance gain peaks at 16 WTs, beyond which MD's throughput starts to gradually decrease, e.g., with a 5% decline at 64 WTs. This indicates that the multi-worker mode has a diminishing return in performance beyond a relatively small number of WTs. Therefore, even in the case of 64 UTs and 64 WTs, parity-RAIDs still fall short of fully leveraging the I/O bandwidth offered by the fast SSDs.

3.2 Identifying the Root Causes

We investigate the CPU usage distribution to identify the root causes of poor write scalability of MD. We use RAID5 with fixed 64 UTs and vary the number of WTs from 1 to 64. We use `perf` [44] to measure CPU cycles of key functions within a WT thread, detailed in Table 3. We randomly select one WT for analysis since all WTs behave very similarly in our experiments. Figure 5 shows that CPU cycles of disk I/O (*RD/WT*) and XORing (*XOR*) decrease as the number of WTs increases, accounting for 42% of the total CPU cycles in the single-worker mode, but only 9.7% at 64 WTs. Meanwhile, the CPU cycles of stripe-write process (i.e., *F/R List*, *Lock*, *Analyze* and *Others*) increase significantly as WTs increase.

First, the global device lock (*Lock*) consumes a mere 4.3% of CPU-cycles in the single-worker mode but a dominant 54.6% in the 64-worker mode. As shown in Table 3, the device lock in Linux

Table 3. Key Function Calls and Locks of Linux Parity-RAID

Operations	**Function as example**	**Description**
RD/WT	generic_make_request()	Send bio to block device queues (② in stages ❷ and ❹)
XOR	async_xor()	Compute parity data (② in stage ❸)
F/R List	release_stripe()	Insert the stripe_head to stripe_list according to its states (① and ④)
Lock	spin_lock_irq(device_lock)	Global MD device Lock, mainly used for updating shared structs
Analyze	analyze_stripe()	Analyze the states of a stripe and its Devs before handling (②)
Others	-	Other software overhead

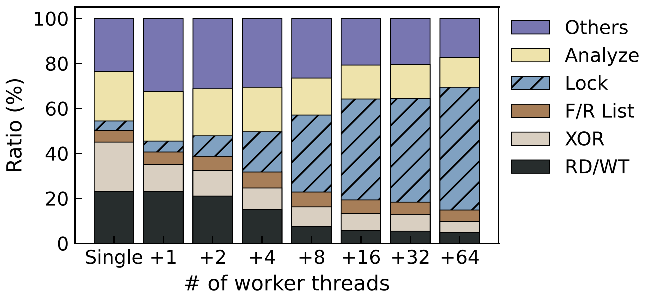


Fig. 5. Breakdown of CPU cycles on key functions and locks of the worker threads in Linux MD.

MD is spin lock, which controls concurrent accesses from WTs, UTs, and DTs to all the stripe_lists and metadata of RAID. In most cases, each WT exclusively accesses the handle_list, thus causing severe lock contention among these threads. Recently, Linux Kernel contributors also found high overhead of the device lock in the read path [52] and replaced them with a lockless memory barrier, thus achieving 7× improvement in small-sized reads. However, the device lock in the write path remains a serious source of contention.

Second, checking for stripe states (*Analyze*) consumes 22% and 13.2% CPU usage in the single-worker and 64-worker modes, respectively. In Linux MD, most of the stripe states and device bio states use a set of semaphores to orchestrate UTs, WTs, and DTs. In summary, through extensive experiments, we observe that the architectural deficiency of the N-for-all centralized handling model leads to severe lock contentions due to highly-concurrent accesses to global data structures and the states of stripes.

4 DESIGN

Given the above identified root causes of write inefficiency of MD with parity-RAIDs running on ultra-fast SSDs, we propose a stripe-threaded architecture of parity-RAID, StRAID for short. StRAID assigns a dedicated worker thread for each stripe-write, which significantly reduces lock contentions among multiple threads, and addresses the partial-stripe-write penalty with a two-stage write submission and a two-dimensional multi-log SSD write buffer.

4.1 Architecture

Figure 6 illustrates the StRAID architecture for parity-RAID. StRAID horizontally separates the space for SSD arrays into two components: a primary RAID array and a Log-Buffer. StRAID does

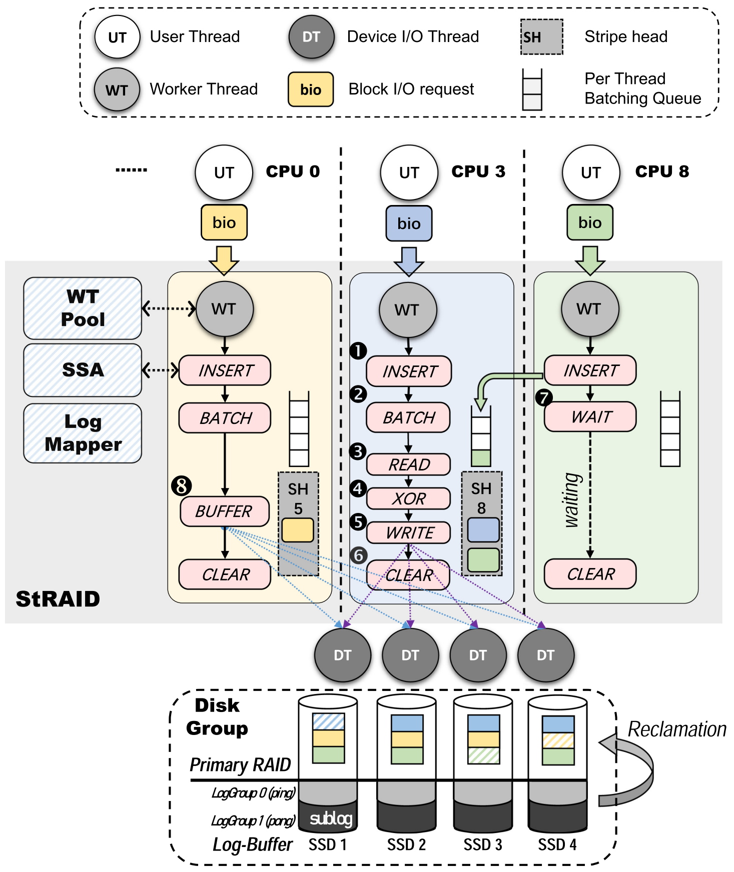


Fig. 6. Architecture and process flow of StRAID.

not change the data layout of the legacy MD for the primary RAID, and persists RAID's metadata at the pre-defined location of each disk.

When a user thread (*UT*) submits a block-write to StRAID, the I/O request (bio) will be sent to a dedicated worker thread (*WT*) that completely and exclusively handles its corresponding stripe. StRAID uses a dynamic allocation strategy that prioritizes assigning bio to a WT that is in idle state or has less unprocessed requests. When a bio updates multiple stripes, each stripe is sequentially processed by its dedicated WT. Multiple WTs process their own stripes independently, exploiting the intrinsic data parallelism among stripes. StRAID pre-allocates at least 128 WTs in the WT Pool to alleviate frequent thread creation/destroy overhead in runtime. Note that all idle threads in the WT Pool are in the sleep state and will not be allocated actual memory space to reduce CPU and memory overhead.

A normal stripe-write process in StRAID can be divided into six consecutive stages of ❶ initializing stripe_heads and inserting bios (INS); ❷ performing I/O batching (BAT); ❸ reading parity/data chunks (RD); ❹ computing parity (XOR); ❺ writing data/parity; and ❻ clearing stripe states in SST (CLR). Moreover, ❼ user threads being batched must wait for completion (WAIT) and ❽ partial-stripe writes that have not been efficiently batched should be conditionally redirected to the Log-Buffer. A notable workflow difference between StRAID in Figure 6 and the legacy MD in Figure 3 is that the latter's stages of ① stripe acquisition, ② analysis and ④ stripe release are

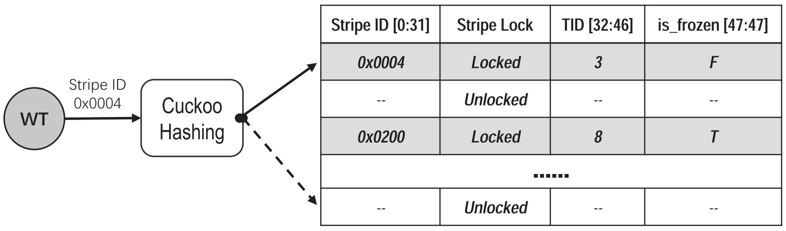


Fig. 7. Stripe state table.

removed in the former. Compared to the legacy MD, StRAID removes the centralized stripe_head lists and their corresponding concurrent operations. Furthermore, StRAID minimizes the number of shared stripe-states and global-state checking among WTs, because a dedicated WT handles a stripe-write exclusively. Finally, the parity computation and I/O execution processes of a stripe write are pinned to the same CPU core, thus avoiding frequent context switches and CPU cache pollution.

However, StRAID faces new challenges in effectively conducting thread collaboration and reducing the partial-write penalty. StRAID still needs a minimal shared-data structure to orchestrate UTs, WTs, and DTs in handling stripe-writes. To this end, StRAID proposes a Stripe State Table (Section 4.2) with lockless access features. Further, the legacy MD uses the global stripe-cache and active/passive delays to aggregate stripe-associated writes (SS-writes) that target the same stripe, thus reducing partial-write-induced disk I/Os. However, in StRAID, a user write triggers a dedicated WT to immediately and exclusively handle the corresponding stripe-write, which does not address the costly partial-write penalty. To solve this problem, StRAID presents a two-stage stripe submission mechanism (Section 4.3) to opportunistically aggregate SS-writes in memory by employing a batching queue per WT (Section 4.3.2). Then, StRAID writes aggregated full-stripes into the primary RAID and conditionally redirects partial-stripe writes to the Log-Buffer (Section 4.4). Finally, StRAID executes buffer reclamation by merging and flushing logging data to the primary RAID in background.

4.2 Stripe State Table

StRAID designs a **Stripe State Table (SST)**, as shown in Figure 7, to maintain a minimal set of shared stripe-states. SST adopts a hash table to index up to 4,096 active stripe-entries, each of which is handled by a dedicated worker thread. An SST-entry (48-bit) contains four fields: 32-bit *Stripe ID* uniquely specifying a stripe; 1-bit *Stripe Lock* indicating whether this stripe is currently being processed; 14-bit *TID* identifying the thread ID of the dedicated WT handling this stripe; and 1-bit *is_frozen* recording the shared stripe-state that indicates whether the stripe is allowed to batch. SST is a globally shared structure between WTs and DTs, where each entry is uniquely associated with a physical stripe and can only be exclusively modified by a WT using CAS [57] at any time. SST employs Cuckoo hashing [53] for achieving high table occupancy while preventing hash collisions. The total memory footprint of SST is smaller than 40KB.

4.3 Two-stage Stripe Submission

4.3.1 Partial-stripe Write Overhead. A partial-stripe write causes write-induced reads and write amplification. The **write-induced-read ratio (**$WIRR$**)** and **write amplification (**WA**)** of RAID5 are estimated by Equation (1) and Equation (2) respectively, where WS, CS and SS represent write-size, chunk-size and stripe-size, respectively. When WS is smaller than CS in RAID5, a block-size write induces 2× read I/Os and 2× write amplification (one data-block write and one parity-block

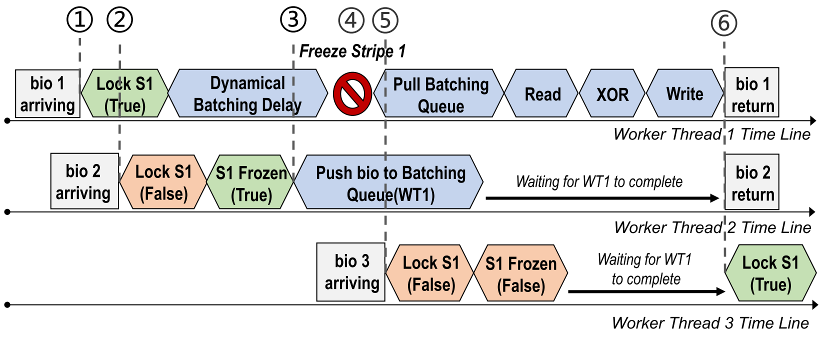


Fig. 8. Workflow of two-stage stripe submission, an example with three concurrent worker threads (*WT1*-*WT3*) targeting the same stripe.

write) with optimal RMW strategy. As *WS* increases, the amount of write-induced-read data decreases (0 for a full-stripe write).

$$Write\text{-}induced\text{-}read\ \ Ratio = \begin{cases} 2 & WS \le CS & (RMW) \\ 1 + \frac{CS}{WS} & CS \le WS < \frac{SS}{2} & (RMW) \\ \frac{SS}{WS} - 1 & \frac{SS}{2} \le WS < SS & (RCW) \\ 0 & WS = SS & \end{cases} \tag{1}$$

$$Write\ \ Amplification = \begin{cases} 2 & WS \le CS \\ 1 + \frac{CS}{WS} & WS > SS - CS \end{cases} \tag{2}$$

Existing optimizations for partial-stripe writes can be catergorized into four general approaches as dynamic stripe size [6, 78], write-aggregation [46], write buffering [31, 68], and parity logging [8, 9, 14, 73, 74]. RAIDZ [6] uses a dynamic stripe size mechanism to eliminate partial-stripe writes, but needs the support of the ZFS file system.

The legacy Linux MD employs a global stripe-cache to absorb active user writes by postponing stripe-writes actively or passively. This write-aggregation approach reduces actual disk I/Os but increases the latency of the postponed requests, which may hurt the overall performance for low-latency SSDs. Generally, it facilitates aggregation of sequential writes, but performs no benefit on random RAID accesses. Moreover, the RAID write buffering approaches quickly persist incoming writes to an auxiliary fast-disk (e.g., SSD or NVM) and then immediately acknowledge to the user. These methods achieve better write performance and improve aggregation efficiency by absorbing more SS-writes. However, it needs to rewrite the relevant data to original locations in the background, which leads to at least 2× write amplification. Simply buffering all write requests would result in shorter lifetime and performance bottlenecks for the buffering device.

4.3.2 *Stripe Aggregation.* Without a global stripe-cache, StRAID designs a two-stage stripe submission mechanism to opportunistically absorb SS-writes in memory. It divides the stripe aggregating process into two stages: a batching stage and a frozen stage. Specifically, Figure 8 (referred by circled numbers) and Algorithm 1 (referred by line numbers) describe the two-stage submission using an example where three concurrent I/O threads issue requests targeting the same stripe (*S1*).

A worker thread 1 (*WT1*) receives bios from its corresponding UT, and acquires a stripe lock to begin stripe processing (*Time* ①, *line 2*) by CAS operation. *WT1* first initializes the stripe states in SST, then performs the in-memory batching by executing the function *dynamical_delay()*. It needs a short initial delay ($D_{initial}$) to quickly determine whether the current SS-write is worth continuing to batch. After that, *WT1* periodically extends the batching time-window by polling

ALGORITHM 1: Two-stage stripe submission

```
function stripe_submission($bios$)
 1: while all $bios$ are handled do
 2:     if get_stripe_lock($stripe\_id$) then
 3:         init_SST($stripe\_id$)
 4:         Determine reconstruction method
 5:         if $bio$ is partial-stripe write then
 6:             dynamical_delay()
 7:         set $is\_frozen = true$ in SST and pull batching $bios$ from queue
 8:         if enable_redirect($bios$) == 1 then        // write to primary RAID
 9:             if Data is not enough for reconstruction then
10:                 Read from disks
11:             Compute XOR and reconstruct stripe to RAID
12:         else
13:             Write $bios$ to Log-Buffer
14:         clear_SST($stripe\_id$)
15:         release_stripe_lock($stripe\_id$)
16:     else
17:         if !is_frozen($stripe\_id$) then
18:             insert $bio$ to queue with TID
19:         else
20:             handled = $false$
21:             continue
22: Waiting for all bios to complete

function dynamical_delay()
 1: que_len = queue($TID$).length()
 2: usleep($D_{initial}$)
 3: $D_{sum} += D_{initial}$
 4: while que_len < queue($TID$).length() do
 5:     que_len = queue($TID$).length()
 6:     usleep($D_{extend}$)
 7:     $D_{sum}$ += $D_{extend}$
 8:     if $D_{sum} \geq D_{max}$ then
 9:         break
10: return

function enable_redirect($bio$)
 1: Calculate $WIRR$ and $WA$ for current $bio$
 2: if LogMapper.hash($stripe\_id$) == true then
 3:     return 0;       // write to Log-Buffer
 4: if $bio$ is full-stripe write then
 5:     return 1;       // write to primary RAID
 6: else if $WIRR \geq C_{WIRR}$ or $WA \geq C_{WA}$ then
 7:     return 0;
```

the corresponding batching queue length. StRAID will continue to aggregate within a fixed time (D_{extend}) in each period until no more requests arrive or reaches the time limit threshold D_{max}. To strike a balance between aggregation efficiency and latency overhead, StRAID sets the default value of $D_{initial}$, $D_{extended}$ and D_{max} to 1/3, 1/4 and 1 of the average IO latency of RAID member SSDs.

Shortly after *WT1*'s arrival, a second worker thread (*WT2*) arrives and seeks SST, only to find that the targeted stripe is locked but enables batching (*Time ②, line 17*). It inserts bios belonging to this stripe to the batching queue of the handling thread *WT1* (*Time* ③*, line 18*) and then suspends itself. When *WT1* completes its batching stage, it immediately transitions the stripe into the frozen stage (*Time* ④*, line 7*) by using the CAS operation. At this point, the stripe is not allowed to accept new bios. Hence, the newly arrived bios from worker thread 3 (*WT3*) (*Time* ⑤) are blocked and have to wait for the stripe write's completion.

WT1 coalesces all requests in its batching queue and processes them as a whole, then it strategically determines whether to redirect SS-writes to the Log-Buffer based on the aggregation results. As shown in the function *enable_redirect()*, StRAID directly stores full-stripe writes to the primary RAID array. For a partial-write after aggregation, StRAID compares the $WIRR$ (Equation (1)) and WA (Equation (2)) of this request to the threshold C_{WIRR} and C_{WA}. If an SS-write has low read and write amplification, StRAID will re-executes parity read (if required) in accordance with the aggregated stripe-write, and performs XORing and data/parity writes to reconstruct the stripe. Otherwise, this request will be redirected to the Log-Buffer. The default value of C_{WIRR} and C_{WA} is 1 and 1.5, which could be defined by the user. An additional case is that StRAID would redirect a SS-write if its corresponding stripe has existed in the Log-Buffer. It ensures that all the data in the Log-Buffer is up-to-date, and we will explain this in the next section. Finally, *WT1* clears up the stripe states of *S1* in SST and releases the stripe lock. The corresponding waiting thread *WT2* will also return successfully (*Time* ⑥*, line 15*). Next, *WT3* successively acquires the *Stripe Lock* to handle its requests on the stripe.

Note that StRAID aggregates requests in memory through the two-stage submission mechanism and returns to the user only after the data is persisted, so it will not increase the risk of RAID inconsistency when the system crashes.

4.4 Partial-write Buffering

To efficiently absorb partial-writes, StRAID employs an SSD-based Log-Buffer and a parallel I/O processing model to fully exploit inter-SSDs parallelism. The latest work Mlog [65] presented an SSD-based multi-log buffer for traditional parity-based RAIDs. We partially refer to MlogâĂŹs two-dimensional multi-log data layout but design a new parallel I/O processing mechanism seamlessly embedded with the StRAID model. Compared to Mlog, StRAID utilizes the dedicated worker threads to process SS-writes for primary RAID and Log-Buffer, eliminating additional thread scheduling overhead. Additionally, in contrast to Mlog buffering all write requests, StRAID first aggregates writes in memory through the two-stage submission and then just stores the rest of partial-stripe writes to the Log-Buffer, which reduces IO amplification caused by data transfer between the Log-Buffer and primary RAID.

4.4.1 Log-Buffer Architecture. The Log-Buffer physically partitions the global buffer space into multiple sublogs, each of which locates within a member SSD and stores partial-writes corresponding to its disks. The capacity of a sublog is 2GB by default. To exploit the intrinsic data parallelism among sublogs, StRAID sets a fine-grained write lock for a sublog to serialize its targeted requests. StRAID deploys a pair of sublogs in each SSD and a set of independent sublogs located across different SSDs constitute a LogGroup. The pair of LogGroups (i.e., LogGroup 0 and LogGroup 1) are considered as a ping-pong buffer, one of which is used to serve front-end requests while the other is dynamically reclaimed in the background. Besides, StRAID uses a Log Mapper to maintain the data mapping relationship between Log-Buffer and the primary RAID.

4.4.2 Request Allocation. Upon receiving a read request, StRAID queries the Log Mapper to determine whether the Log-Buffer has stored the most recent data. Otherwise, read requests are directly decomposed to their constituent chunk I/Os served by the primary RAID array. For a redirected partial-write, StRAID will split the request into chunk-size aligned sub-writes, then

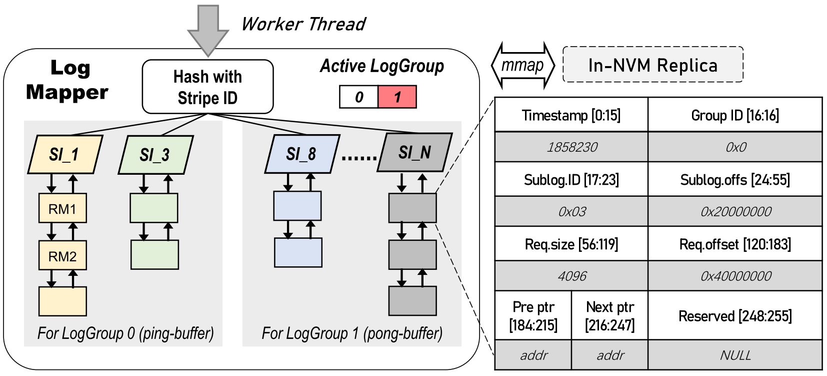


Fig. 9. Architecture of Log Mapper.

submits them to the tail of the corresponding sublogs in the same SSD one by one until all requests have been issued. This method utilizes inter-SSD parallelism to handle concurrent writes while balancing capacity usage across various sublogs. Finally, StRAID updates the index information for the buffered requests in the Log Mapper.

4.4.3 Log Mapper. Figure 9 illustrates the architecture of Log Mapper that speeds up queries and reclamation on buffered data by efficiently recording and indexing the data mapping relationship between the Log-Buffer and the primary RAID.

The Log Mapper allocates a fixed region for the ping-pong buffer to store all the correlated requests metadata in memory. Each buffered request is associated with a dedicated request metadata structure (i.e., *RM*). The RM-entry (256-bit) contains eight fields: 16-bit *Timestamp* recording request arrival time; 1-bit *Ping-pong ID*, 7-bit *Sublog ID*, and 32-bit *Sublog offset* together uniquely specifying the location of a buffered request; 64-bit *Request size* and 64-bit *Request offset* identifying the original write location on the primary RAID; two 32-bit *address pointers* for building double linked list structure. The last unused 8-bits are used as reserved bits.

To efficiently support concurrent RM queries and updates, Log Mapper proposes a two-level indexing structure. It contains a high concurrency hash table as the first-level index with O(1) lookup cost. Each of the hash entry corresponds to a StripeIndex structure (i.e., *SI*) that locates a physical stripe of the primary RAID. For updates or queries, a WT needs to first determine the stripe number (i.e., Stripe ID) of the request and then uses it as the hash key to find the correlated *SI* structure. The StripeIndex contains a second-level index called *RM_List* that uses a double-linked list to concatenate all RMs corresponding to this stripe in order of arrival time. Each StripeIndex uses a read-write lock to serialize concurrent updates to its corresponding *RM_List*. Log Mapper updates and queries at the request granularity, but deletes hash entries at the LogGroup granularity.

4.4.4 Buffer Reclamation. StRAID employs a dedicated flushing module to merge and flush buffer data to the primary RAID eventually. As shown in Algorithm 2, the flushing module monitors the LogGroup's capacity. If the average capacity of its contained sublogs exceeds a threshold C (i.e., 90% by default), the flushing module atomically updates the flag of active LogGroups to redirect incoming requests to another idle LogGroup.

The maximum number of flushing threads is 4 by default to efficiently utilize the SSD parallelism. During an I/O-intensive period, the flushing traffic will be limited to mitigate the performance impact on the foreground user requests. If the Log-Buffer is full, StRAID will disable the write buffering redirection until one of the ping-pong buffer is flushed out. Moreover, flushing buffered writes without merging induces numerous partial stripe updates in the RAID. To solve this problem,

ALGORITHM 2: Buffer reclamation and request merging

```
Input: The full $LogGroup$ with flushing threads $FT_{1-N}$;
 1: Calculate $Avg_{cap}$ through querying the capacity of contained sublogs
 2: if $Avg_{cap}$ > 0.90 then
 3:     Atomically change the active LogGroup
 4:     Determine the number of flushing threads
 5:     wakeup_thread($FT_{1-N}$)
 6:     $SIs$ = LogMapper.GetallSH($Group$)
 7:     for merging and flushing all stripes do
 8:         $RMs$ = $SI$.Traverse($RM\_List$)
 9:         $stripe$ = merge_to_stripe($RMs$)
10:         read_sublog($stripe$)
11:         write_RAID($stripe$)
12:         delete $RMs$ and $SI$ in LogMapper
13:     clear_data($Group$)
```

StRAID tries to merge logging data into full-stripe writes. The flushing thread first accesses the Log Mapper to fetch all *StripeIndex* of the corresponding LogGroup, then it iterates the *RM_list* to read buffered data in sublogs and merges them into full-stripes in memory. After flushing a stripe to RAID, the corresponding *SIs* and *RM*-entries are removed to prevent dirty read.

The data consistency between the Log-Buffer and primary RAID is another problem. Suppose that at time T1, a small request for a particular stripe is redirected to the buffer, and at time T2, a large request for the same stripe is submitted to the RAID. If Log-Buffer simply flushes the buffer at time T3, it will break the rule of sequential consistency. StRAID employs a request scheduling approach to ensure that the Log-Buffer always contains the most recent data in order to prevent such scenarios. Before sending a stripe-write to the RAID, the dedicated WT will check the Log Mapper to determine if the relevant stripe already exists in the buffer by hashing with the Stripe ID, as illustrated in Algorithm 1, function *enable_redirect()*, line 2. If it exists, the SS-write will be redirected to the Log-Buffer even for a full-stripe request.

5 IMPLEMENTATION

5.1 Recovery and Degraded Mode

Crash Consistency and Recovery After a system crash, part of the chunk writes belonging to a stripe-write may be lost, making the stripe inconsistent between its data and parity. StRAID uses a bitmap to record the current update-state of each chunk. Compared with Linux MD, StRAID's bitmap has basically the same data structure and layout, but can only be updated and flushed by dedicated threads. For each chunk update, StRAID first sets the corresponding bitmap bits and changes their involved memory-page as dirty, then flushes the page to the underlying SSDs via the memory mapping mechanism. The bits will be cleared after their corresponding chunks are written to the disk. StRAID groups bitmap updates of the aggregated write requests in a batch to avoid frequent disk I/Os. In the experiment, it is found that flushing the bitmap only incurs a very small overhead (less than 2%) when handling stripe writes. With unexpected power failures, StRAID will fetch the bitmap from the disks and restore it to the consistent state after reboot.

Moreover, the Log-Buffer in StRAID addresses the consistent issues of buffer reclamation by updating a reclamation completion flag in the Log Mapper and persisting it to NVM or SSD. StRAID sets the flag when launching a reclamation and clears it after the reclamation is finished, then the buffered data would be removed. After a crash, StRAID will scan and check the persistent Log Mapper replica to recover system states. If a reclamation process had not been done before the crash, StRAID would re-write all buffered data to the primary RAID to complete this reclamation.

Resync and Degraded Mode StRAID supports degraded reads, degraded writes and resync operations in the same way as the legacy MD because the underlying data layout is identical. For stripe writes, StRAID identifies the degraded stripe and handles it after entering the frozen stage. The resync operation reads all the data blocks from disks and compares their calculated parity results with their on-disk parity data. It is triggered upon RAID initialization, persistent bitmap failure or reconstruction from disk replacement. Besides, StRAID will flush all buffered data after recovery if the primary RAID array is in a normal state. We evaluate the performance of StRAID in degraded mode in Section 6.

5.2 Optimization for Persistent Memory

The Intel Optane DC Persistent Memory Model [76] is the first commercialized PM product. Compared to traditional block storage devices such as SSDs and **HDDs (Hard Disk Drives)**, PMs have extremely low access latency and byte-addressable features. Traditional storage I/O stack that were developed for SSDs are unable to fully take use of the PM's fast persistency features. In addition, the interleaving mode only manages multiple PMs in a RAID0-like manner [76] and lacks providing data reliability.

StRAID proposes three techniques to optimize I/O access when it is built upon multiple PMs. First, StRAID uses the memory I/O stack and interface to access PM spaces. To this end, it will memory-map the PM space in advance and persist data with memcpy and ntstore operations [76]. Second, StRAID will perform the pre-fault process when mmaping PM space, thus reducing the overhead of page-fault at runtime [30]. Third, previous studies [71, 76] indicate that PM has bounded thread-level scalability with a peak write throughput at 4 threads for a single PM. Therefore, StRAID schedules at most 4 WTs for accessing an underlying PM concurrently.

6 EVALUATION

6.1 Evaluation Setup

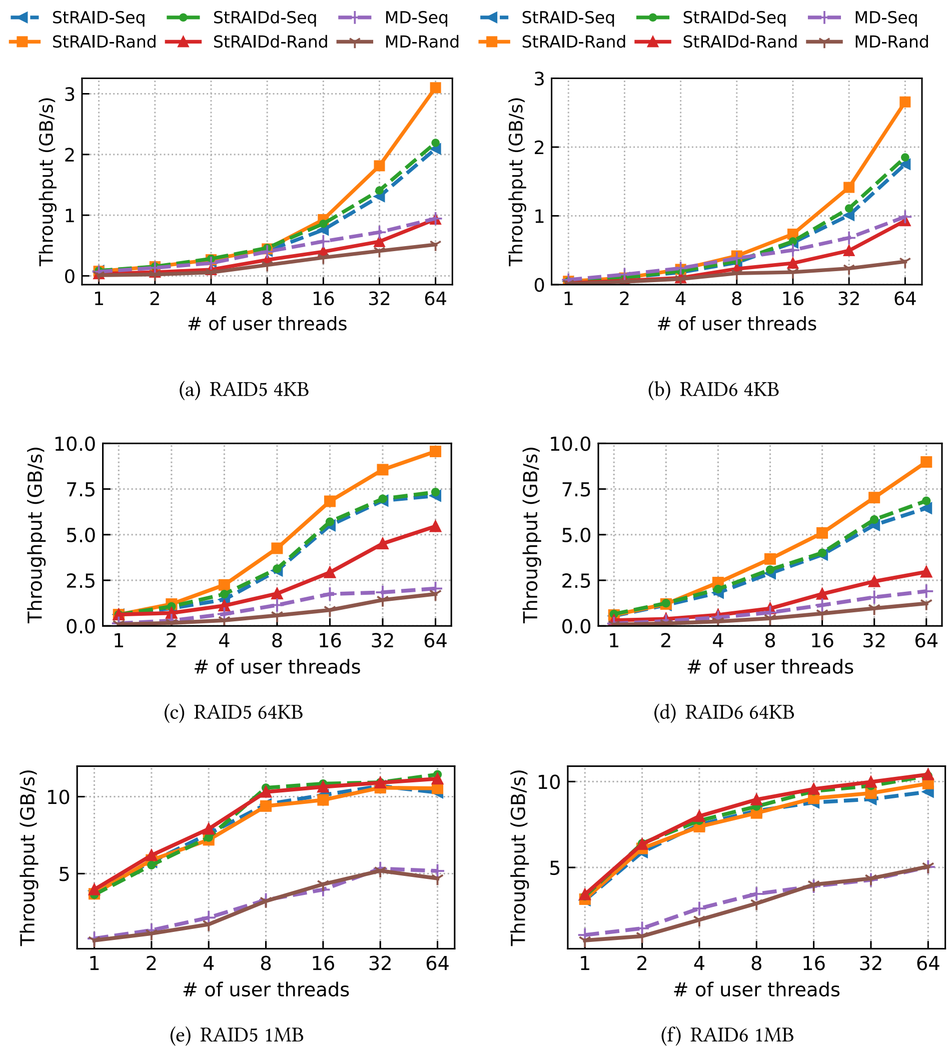
Platform We run all experiments on a server (detailed settings listed in Table 1) and three types of SSD devices (described in Table 2). The CPU-core can reach 29GB/s XORing throughput and the PCIe I/O bandwidth is 48GB/s [20], exceeding the aggregate sequential bandwidth of 6 NVMe-based SSDs (2.6GB/s stable write throughput per SSD, 15.6GB/s in total). In our experiments, we bind all the I/O threads and worker threads to the same CPU socket-0 to avoid remote accesses of memory and PCIe, i.e., the NUMA issues.

RAID systems setup We evaluate StRAID (`StRAID`) and Linux MD (`MD`) of the RAID5 (5+1) and RAID6 (4+2) levels built on 6 SSDs. The chunk size is set to 64KB by default. `StRAID` has enabled the optional Log-Buffer that is composed of two 2GB sublogs on each of the RAID member SSDs (24GB in total), while `StRAIDd` disabling the write buffer. Linux MD has a 16K-entry stripe-cache and up to 64 worker threads. This is a setting that MD is shown by our experiments to achieve the best throughput.

Workloads We implement a program to issue direct block I/O requests with sequential or random access patterns as micro-benchmark. We run each experiment ten times and take the average as the results. We further select six representative block traces summarized in Table 5 as trace-driven macro-benchmarks. We implement a trace player in C++ using POSIX sync to generate direct block I/O requests to the underlying RAID systems.

6.2 Micro-benchmark

We measure the write throughput, average and tail latency, and amount of disk read/written data on `MD`, `StRAID` and `StRAIDd` with RAID5 and RAID6 on 980Pro SSDs, respectively. We generate



(a) RAID5 4KB (b) RAID6 4KB

(c) RAID5 64KB (d) RAID6 64KB

(e) RAID5 1MB (f) RAID6 1MB

Fig. 10. Write scalability on three different RAID systems.

workloads with a different number of concurrent I/O-issuing threads (i.e., UTs) and three different access patterns: write-only, read-only and mixed read-write, respectively. Three default I/O sizes are: 4KB (default block-size of file systems, page cache, and block devices), 64KB (partial-stripe write size), and 1MB (full-stripe write size).

Throughput Figure 10 reports the write throughput of `StRAID`, `StRAIDd` and `MD` in RAID5 and RAID6, respectively. The throughput of StRAID exceeds that of MD by up to 1.5× with 4KB-sized writes and 2.1× with 64KB-sized writes on a single UT, respectively. This is because StRAID effectively minimizes the cost of analyzing stripe-states. As the number of UTs increases to 64, StRAID achieves up to 3.1GB/s ± 0.2GB/s and 9.1GB/s ± 0.7GB/s peak throughput with 4KB and 64KB writes respectively, representing 3.6× and 4.5× performance improvement over MD.

With 4KB-sized and 64KB-sized requests, the random write performance of StRAID is up to 2.9× higher than that of StRAIDd, and even generally 15%-30% higher than sequential writes. This is because the two-stage submission redirects random partial-writes to the Log-Buffer, thus largely

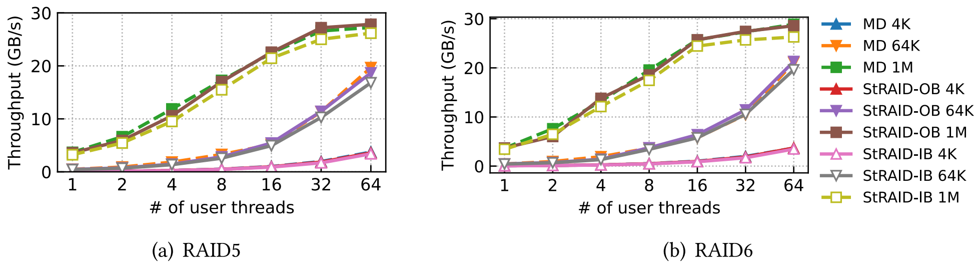


Fig. 11. Read scalability of StRAID and MD.

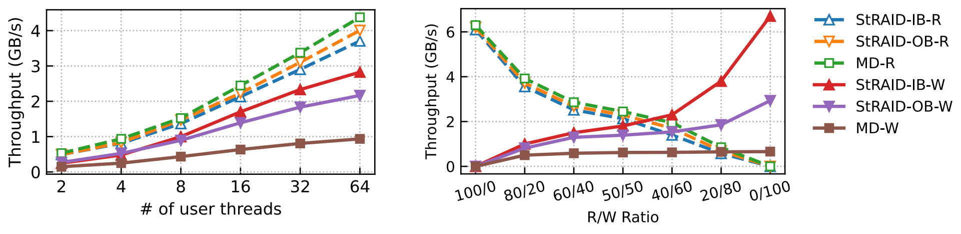
reducing the cost of stripe reconstruction (i.e., XORing and write-induced-read) in the user I/O path, as well as obtains better write scalability through unleashing the high inter-parallelism of multiple modern SSDs. However, the cost of enabling Log-Buffer is additional SSD read and write I/Os for each buffered requests. Besides, the sequential write throughput difference between StRAID and StRAIDd is smaller than 6%, because StRAID has effectively batched these requests through the in-memory aggregation.

For full-stripe writes (i.e., 1MB), StRAID achieves 4.6× and 5.2× higher write throughput in random and sequential cases than MD with a single UT, respectively. As the number of UTs increases to 8, StRAID's throughput saturates the device bandwidth, with an almost fixed increase of about 6GB/s over MD. With 64 UTs, the peak write throughput reaches 11.4GB/s ± 1.1GB/s and 10.4GB/s ± 1.0GB/s in StRAID under RAID5 and RAID6 respectively, which are 2.1× higher than those of MD (5.2GB/s and 5.1GB/s). StRAID's full-stripe writes nearly unleash the full power of the SSD performance, while MD suffers from heavy contention on the global data structures. In addition, StRAID shows a 3%-5% performance decrease than StRAIDd with full-stripe writes, because the WT has to check the Log Mapper for each SS-write to maintain data consistency.

Figure 11 shows the read throughput of StRAID and MD with varying-size reads. We measure the in-buffer read throughput (i.e., `StRAID_IB` and the out-of-buffer read throughput (i.e., `StRAID_OB`, respectively. The average read throughput difference between `MD` and `StRAID_OB` is less than 6% in RAID5 and RAID6 respectively, demonstrating that StRAID does not affect read performance. Moreover, the read throughput of `StRAID_IB` is 8%-13% lower than `StRAID_OB` at 64 threads, due to the overhead of searching Log Mapper and RM_Lists. StRAID will flush the buffered data to the primary RAID, and the long-term read performance is the same as `StRAID_OB`.

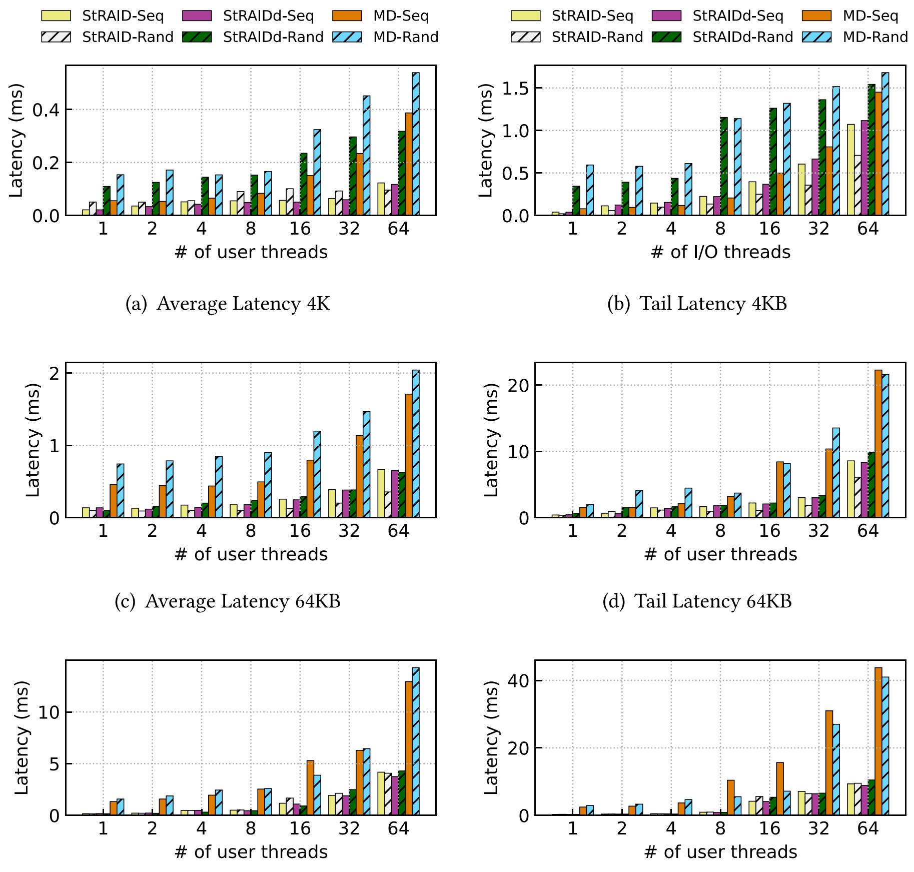
Moreover, Figure 12 shows the write and read throughput of StRAID and MD with mixed workloads. For StRAID, we perform in-buffer writes (`StRAID-IB-W`) and out-of-buffer RAID writes (`StRAID-OB-W`) while reading the main RAID array, respectively. The experiments shown in Figure 12(a) equally divided the total user threads into two groups that sent random 64KB read and write requests respectively (i.e., 50%-read and 50%-write workload). Results demonstrate that StRAID's write performance is still 2.1×–2.8× higher than MD under mixed workloads. However, with more than 16 user threads, the read performance under `StRAID-IB-W` and `StRAID-OB-W` decreases by 9% and 15% compared to MD, respectively. This is because the total bandwidth of SSD devices is limited, and the higher write throughput of StRAID will compete for reads in the mixed workload. Nevertheless, the total read and write throughput of StRAID is still 1.2×–1.4× higher than that of MD.

Furthermore, Figure 12(b) shows the throughput of StRAID and MD under different ratios of read/write mixed workload, with a fixed number of 16 user threads. When the read/write ratio is 40/60, the write throughput of `StRAID-IB` and `StRAID-OB` is up to 6.1× and 3× higher than that of MD, while the read throughput is reduced by a maximum of 19% and 12%, respectively. It



(a) 50R/50W Mixed (b) Different R/W Mixes

Fig. 12. Throughput of StRAID and MD with read-write mixed workloads.



(a) Average Latency 4K (b) Tail Latency 4KB

(c) Average Latency 64KB (d) Tail Latency 64KB

(e) Average Latency 1MB (f) Tail Latency 1MB

Fig. 13. Average and tail latency of RAID systems.

indicates that StRAID demonstrates lower interference between read and write requests because it effectively improves CPU utilization under concurrent workloads. Additionally, the interference between read and write requests is higher in write-mostly workloads due to the write-induced read and write amplification caused by RAID partial-writes.

Latency and Breakdown of CPU-cycles Figure 13 shows the average and tail (99^{th}-percentile) latency under RAID5 in `StRAID`, `StRAIDd` (with Log-Buffer disabled) and `MD`, respectively. StRAID

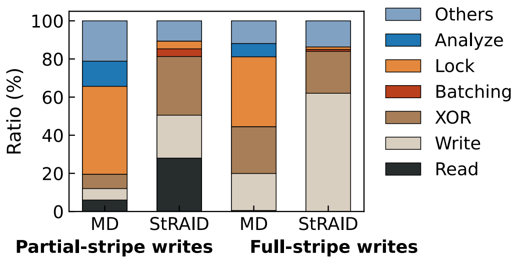


Fig. 14. Breakdowns of CPU cycles on StRAID and MD.

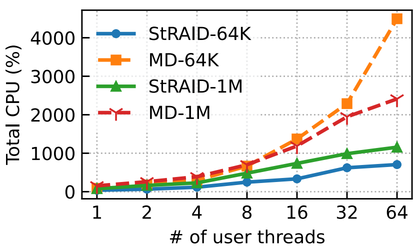


Fig. 15. Total CPU utilizations.

significantly outperforms MD in both average and tail latencies performance under 64 UTs, reducing latency by 75% with 4KB block-writes, 76% with 64KB partial-stripe writes, and by 69% with full-stripe writes. StRAID reduces 22%-67% tail latencies from MD under 64 UTs. Besides, the average and tail latency of StRAID is up to 1.5× and 2.2× better than StRAIDd for random writes respectively, because the write buffering method provides an efficient I/O path for quickly storing writes in the Log-Buffer.

To better understand the reasons behind StRAID's superiority, Figure 14 shows the breakdown of the CPU-cycles of key functions consumed by MD and StRAID with 64WTs and 64 UTs, respectively. For partial-stripe writes, the combined CPU-cycles on XORing and disk I/Os account for 76% of the total CPU usage in StRAID, while that of MD is less than 20%. The average stripe-write handling overhead of StRAID, i.e., 62μs, is about 19 times less than that of MD, i.e., 1180μs. Besides, the lock overhead on StRAID and MD account for 5.1% and 46.1% of the total CPU usage, respectively. StRAID efficiently mitigates lock contentions through the stripe-threaded architecture and the lockless access features in SST.

For full-stripe writes, the lock, XORing and I/O-write of StRAID account for 1.3%, 22.5% and 62.6% of the total CPU usage, respectively, in contrast to their MD counterparts of 36.7%, 24.5% and 19.4%, suggesting that StRAID achieves to make better advantage of SSDs' high write bandwidth. In addition, the two-stage submission consumes only 8% and 2.1% of the stripe-write CPU-cycles of partial-stripe and full-stripe writes, respectively.

CPU utilization We compare the CPU utilizations of `StRAID` and `MD` under random full-stripe and partial-stripe write workloads respectively, with the same RAID5 settings in Figure 10. Results in Figure 15 show that the total CPU utilization of MD is up to 6.3× higher than StRAID with 64 UTs. Even when the number of UTs is less than 8, the CPU usage of MD is 2× higher than that of StRAID on average. Combining with the throughput results shown in Figure 10, MD with 4495% CPU-core utilization consumes only 1/3 of the SSDs bandwidth, in contrast to StRAID that consumes 86.9% of the SSDs bandwidth with 1156% CPU-core utilization.

Moreover, 64KB-sized partial-stripe writes of MD (`MD-64K`) consume up to 80% more CPU than full-stripe writes (`MD-1M`) with 64 UTs. MD's inefficiency stems from its high consumption of CPU cycles required to handle in-flight partial-stripe writes. On the contrary, `StRAID-64K` consumes only 25% less CPU cycles than `StRAID-1M` because StRAID gains higher throughput for full-stripe writes that consumes more CPU resources for computing XOR and issuing I/Os.

Amount of Disk Read/Written Data Figure 16 shows the amount of data written to and read from disks by `StRAID` and `StRAIDd` in RAID5, normalized to that of MD, including disk accesses on the Log-Buffer. For random writes, StRAIDd and MD have exactly the same amount of data written and data read because stripe-write aggregation is rare for random writes. Besides, StRAID buffers all random partial-writes in the Log-Buffer, which performs 6.1× more throughput at the cost of 1.5× more data written to and read from SSDs than MD.

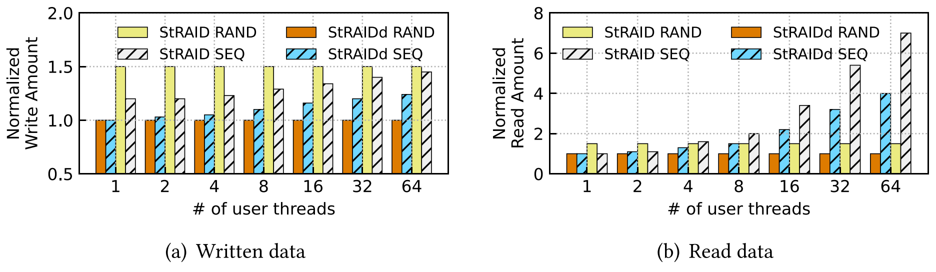


Fig. 16. Amount of data written to and read from disks by StRAID, normalized to that of MD.

Table 4. The Aggregation Degree of the Two-Stage Submission Mechanism in StRAID

User Threads	1	2	4	8	16	32	64
Aggregation Degree (%)	20	34.6	66.3	79.8	86.8	88.8	89.2

For sequential writes, the amount of disk written and read in StRAID varies significantly with different numbers of UTs, which is caused by the various efficiency of StRAID's in-memory stripe aggregation. With a single UT, StRAIDd and MD have the same amount of disk read and write because they can not batch stripe in memory. StRAID conducts 20% more written data than MD because it redirects all writes to the Log-Buffer and then merges them into full-stripes. With the number of UTs increasing, StRAID would batch more sequential writes through the in-memory aggregation and conducts less buffered write. However, the amount of data written in StRAIDd is up to 24% larger than MD. This is because the in-memory stripe aggregation has a much smaller batching window, which is slightly less efficient than Linux MD's active delays of sequential stripe writes.

For data read, with 64 UTs as the worst case, StRAID and StRAIDd read 7× and 4× more data than MD, respectively. It is because MD postpones and aggregates almost all stripe-associated writes (SS-writes) into full-stripe writes, thus reducing the amount of write-induced-read data (e.g., 1.6% of user-written data). In contrast, StRAID's worker thread exhibits less efficient stripe aggregation, resulting in a higher number of write-induced reads (about 6.5% of user-written data) and an additional read for recycling each buffered request. Table 4 presents the aggregation degree of StRAID, defined as the ratio of the average stripe write size to the full-stripe size after stripe batching, of the two-stage submission mechanism under a sequential 64KB workload. When the number of user threads is lower than 8, StRAID shows a lower aggregation efficiency due to limited workload concurrency. However, as the number of threads increases beyond 16, the two-stage submission mechanism achieves an 89.2% aggregation degree. Besides, since the high read IOPS of fast SSDs can completely absorb the increased number of read I/Os, StRAID's write performance can still be 5.1× higher than MD.

Chunk Sizes We evaluate the effect of chunk size configuration on the performance of StRAID and MD with 64KB-sized writes in RAID6. The chunk size is set to 8KB for the full-stripe-write case (`StRAID-F` and `MD-F`), and 64KB for the partial-stripe-write case (`StRAID-P` and `MD-P`), respectively. Figure 17 shows that both StRAID and MD benefit significantly from full-stripe write workloads. The throughput of StRAID-F reaches 11.8GB/s with 64 UTs, about 1.9× higher than StRAID-P. Similarly, the throughput of MD-F is up to 3.1× higher than MD-P. However, the peak throughput of MD-F (8KB chunk size and 64KB write size) remains at 5.3GB/s, consistent with the results shown in Figure 1(d) (with 64KB chunk size and 1MB I/O size). It indicates that the peak throughput of

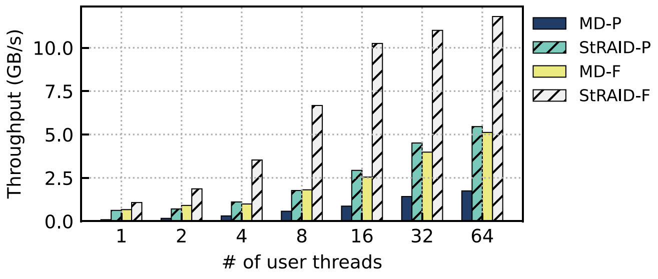


Fig. 17. StRAID and MD throughput with partial-stripe *(*-P)* and full-stripe writes *(*-F)* of different chunk sizes.

Table 5. Characteristics of Block I/O Traces used in the Macro-benchmark Evaluations

Trace	Write Ops (millions)	Data Written (GB)	Avg.write size (KB)	Read Ops (millions)	Data Read (GB)	Avg.read size (KB)
Pangu-A	1.89	113.24	63.21	0.24	4.06	17.99
Pangu-B	2.44	81.32	35.08	0.30	18.61	65.24
prxy_0	12.14	53.80	4.65	0.38	3.05	8.33
prn_0	4.98	45.97	9.67	0.60	13.12	22.84
varmail	3.39	39.20	12.13	0.05	5.38	114.05
fileserver	1.19	99.45	87.56	0.47	42.37	95.49

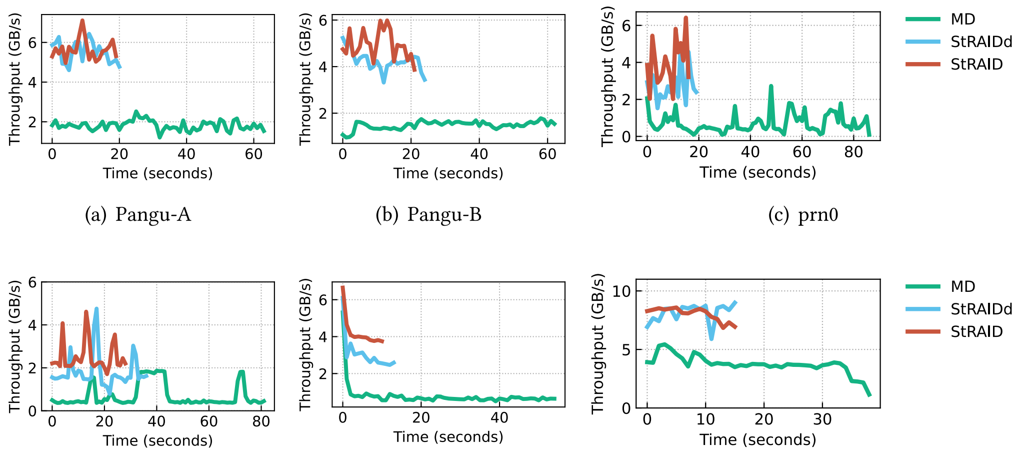
StRAID is sensitive to the chunk size setting. An insight from this experiment is that it is beneficial to set StRAID's chunk size smaller, such as 8KB, to take full advantage of full-stripe writes.

6.3 Macro-benchmark

We use six representative block traces from Filebench [60], cloud-based application traces from Alibaba-Pangu [47] and Microsoft [51] to evaluate StRAID's performance. Table 5 summarizes the characteristics of these workloads, most of which are read-write mixed or write-dominated. The average request size of these workloads is generally small (i.e., 20-60KB). Therefore, we evaluate StRAID and MD in the RAID5 level with the chunk size of 8KB, which is beneficial for StRAID to take advantage of full-stripe writes as we demonstrated in previous Section 6.2. In the experiments, we enable 32 WTs in MD and StRAID, and evaluate them in the RAID5 levels with a chunk size of 8KB. We invoke 32 user threads to replay these traces continuously, mimicking high-intensity workloads.

Figure 18 shows the throughput of StRAID and MD over time. StRAID achieves up to 2.8× higher throughput than MD, and shortens the total running time by an average of 64% across six workloads. In the fileserver workload, StRAID achieves peak and average throughput of 10.3GB/s respectively, in contrast to their MD counterparts of 5.0GB/s. The fileserver workload has the largest average write size, so that StRAID benefits from full-stripe writes. For the cloud-based workloads, StRAID's average throughput is 3.1× and 3× higher than MD's in Pangu-A and Pangu-B, respectively. The prxy0 workload exhibits the lowest average throughputs among all workloads, 1.8GB/s and 0.6GB/s for StRAID and MD, respectively. This is because the prxy0 trace has the smallest average write size (i.e., 4.6KB) among all workloads, leading to a large amount of partial-stripe writes for both StRAID and MD. Further, it is observed that StRAID is 20%-35% better than that without Log-Buffer in prxy0, prn0 and varmail, because these workloads have more performance-punishing partial-stripe writes.

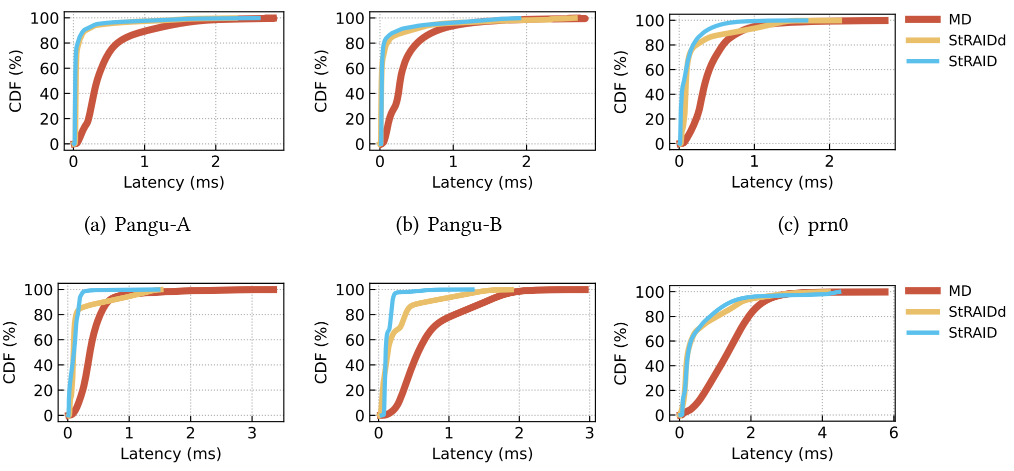
Figure 19 shows the latency CDFs of StRAID and MD across all the workloads. StRAID shows significantly better CDF profiles, with about 80% and 69% lower average latency than MD in



(a) Pangu-A (b) Pangu-B (c) prn0

(d) prxy0 (e) varmail (f) fileserver

Fig. 18. Throughput of StRAID and MD on trace-driven workloads.



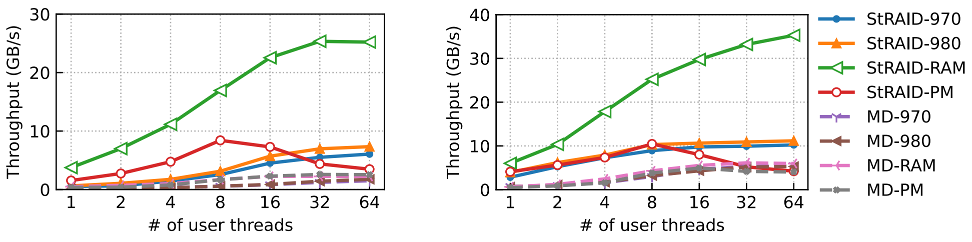
(a) Pangu-A (b) Pangu-B (c) prn0

(d) prxy0 (e) varmail (f) fileserver

Fig. 19. Latency CDF of StRAID and MD on trace-driven workloads.

workloads Pangu-A and Pangu-B, respectively. For the other four workloads, StRAID also has at least 45% lower average latency than MD. The median latencies of StRAID in workloads Pangu-A, Pangu-B and filebench are almost ten times lower than MD, while for workloads varmail, prn0, and prxy0 StRAID's is 78%, 74% and 75% lower than MD's, respectively. In addition, the 80^{th}-95^{th} percentile latency of StRAID is reduced by generally 42%-63% than StRAIDd in varmail, prxy0, and prn0 workloads. The primary source of these long latency is the partial-stripe updates, which could be efficiently handled by the write buffering mechanism.

The 99^{th}-percentile tail latency in StRAID is 25% lower than that in MD among all workloads on average. For example, StRAID's tail latency is up to 31.1% and 31.9% lower in workloads Pangu-A and prn0. StRAID's advantage over MD in tail latency is lower than in average latency, because the high tail latency of both StRAID and MD mainly comes from write latency spikes caused by internal maintenance operations (e.g., garbage collection) within the SSD devices.



(a) Partial-stripe writes (b) Full-stripe writes

Fig. 20. Performances of StRAID and MD on different SSDs and RAMs.

6.4 Sensitivity Study

Experiment with other devices. Next, we evaluate the sensitivity of `StRAID` and `MD` to different types of storage devices. We first build StRAID and Linux MD on six Intel Optane PMs (in AppDirect Mode) [27] and lower-performance 970Pro SSDs, and compare these performances with that in 980Pros. The raw read and write bandwidth per PM can reach 6.6GB/s and 2.3GB/s [76], respectively. When running upon PMs, we will turn on the PM-oriented optimization techniques as described in Section 5.2 (`StRAID-PM`). Further, we test the extreme RAID performance over six ramdisks on 128GB DRAM. We invoke up to 64 UTs with 64KB write-size for partial-stripe write-load and 1MB for full-stripe write-load.

Results in Figure 20 show that StRAID on 980Pro SSDs exhibits up to 20% higher throughput than it on 970Pro SSDs. In contrast, the performance difference of Linux MD on these two different types of SSDs is less than 5%. The throughput of StRAID on PMs is up to 3.8× and 2.5× higher than MD with partial-stripe and full-stripe writes, respectively. We also find that StRAID on PMs shows 50%-82% higher throughput than SSDs on average with small number of threads. This is because PM has one order of magnitude lower read and write latency than SSDs, thus StRAID could handle stripes more efficiently. However, the PM-aware StRAID shows a throughput drop at higher than 8 UTs, because PM has a limited concurrency [76]. In addition, StRAID on RAMs delivers up to 5.8× higher write throughput than MD. At 64 UTs, StRAID reaches up to 35.2GB/s random write throughput and 32.7GB/s sequential write throughput, respectively, in contrast to their MD counterparts of 5.8GB/s and 5.7GB/s. It shows that StRAID has the potential to effectively exploit faster storage like the emerging PCIe 5.0 SSDs [17] in the near future.

To demonstrate the generality of StRAID, we further evaluated StRAID and MD over six low-performance 860Pro SATA SSDs (`*-860`) with limited 4 worker threads. Table 2 shows that the 860Pros have a lower read and write bandwidth of about 500MB/s, which is one order of magnitude lower than the 980Pros. In this experiment, we issue random 64KB write requests using 1 to 16 UTs. Figure 22 demonstrates that the throughput of StRAID peaks at 4 UTs because the stripe-threaded architecture uses dedicated threads to handle each stripe-writes exclusively. In contrast, MD's performance peaks at 12 and 16 user threads with 860Pros and 980Pros respectively, with a maximum throughput that is reduced by 10% and 40% compared to StRAID. These results indicate that StRAID still outperforms MD even with limited CPU resources and lower-performance SSDs.

Two-stage Submission. We analyze the performance contributions of the **two-stage stripe submission (TSS)** mechanism of StRAID in RAID5. We run the experiment with and without the two-stage submission (`StRAID` and `StRAID w/o TSS`), and just disable the write buffering (*StRAIDd*), respectively. The request size is set to 64KB for the partial-stripe-write case. We issue requests with random access patterns. Figure 21 shows that StRAID with TSS achieves 2.7× improvement of average throughput than without TSS for sequential partial writes at 64 UTs. The

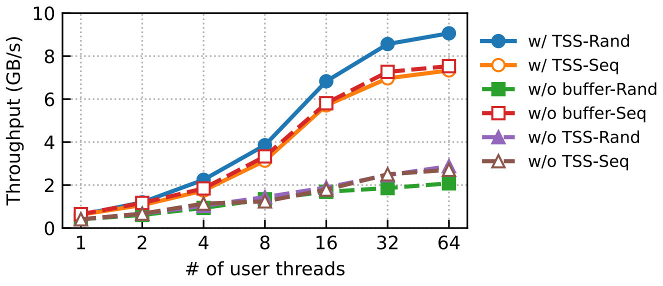


Fig. 21. StRAID throughput with random *(*-Rand)* and sequential *(*-Seq)* writes when running with two-stage stripe submission (*w/ TSS*), without write buffering (*w/o buffer*) and without TSS (*w/o TSS*).

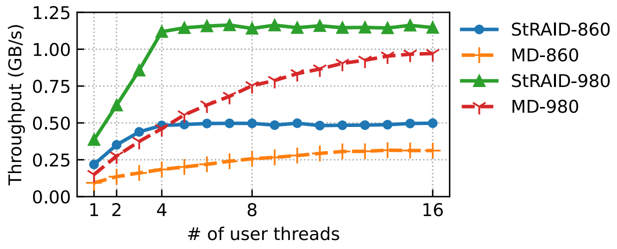


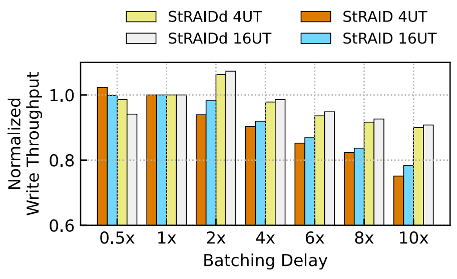
Fig. 22. Throughput of StRAID and MD with limited worker threads.

two-stage submission allows request aggregation on writes belonging to a same stripe and handles them in a batch. StRAID without TSS, by contrast, has to individually execute each writes on a stripe. Besides, the performance contribution of two-stage submission on full-stripe write load is less than 4%, because the requests targeting different stripes will not be aggregated.

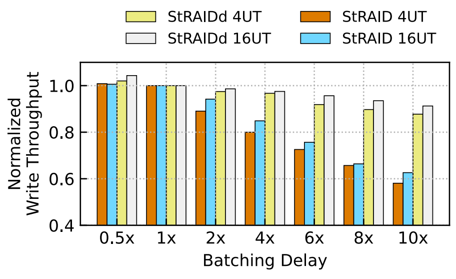
Furthermore, we analyze the effects of batching time in TSS on the overall performance of StRAID. The default parameter values ($D_{initial}$, $D_{extended}$, D_{max}) for StRAID built with 980Pro SSDs are 7us, 5us, 20us and with 860Pro SSDs are 20us, 15us, 55us, respectively. We conduct experiments to evaluate the effect of varying these parameters on StRAID's performance. Specifically, we use the default delay settings (`1x`) as a baseline and adjust all three parameters simultaneously, increasing them up to `10x` or decreasing them by 50% (`0.5x`). We employ 4 and 16 threads (i.e., 4UT and 16UT) for random and sequential 64KB-sized writes and measure the performance of StRAID (enable Log-Buffer) and StRAIDd (disable Log-Buffer), respectively.

Figure 23 shows that increasing the aggregation time beyond the default configuration significantly reduces the overall write performance of StRAID. For example, with 980Pro SSDs, StRAID's sequential and random throughput decreases by 17% and 35% respectively with the `8x` batching delay, because the performance penalty on IO processing latency is much higher than the performance gain from increased aggregation efficiency. A special case is that the sequential write throughput of StRAIDd is increased by 7%-9% compared to the baseline under the `2x` delay setting. This is because the longer batching time window improves the aggregation efficiency of SS-write. However, it also decreases the overall write performance of StRAID by up to 10% due to the increased IO processing latency. In contrast, decreasing the aggregation time (`0.5x`) slightly improves random write throughput by 3%-5%, but results in a 6% decrease in sequential writes due to the lower batching efficiency.

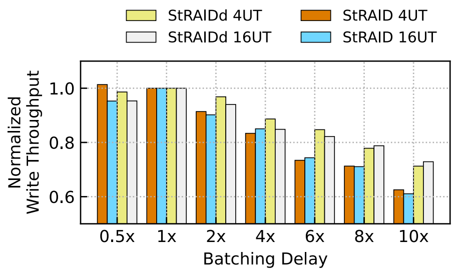
Moreover, the performance impact of increased aggregation time for StRAID is 1.2×–2× more than that for StRAIDd. This is because StRAID addresses the partial-write performance problem



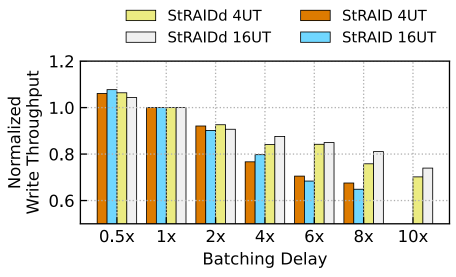
(a) Sequential Write (980Pro)



(b) Random Write (980Pro)



(c) Sequential Write (860Pro)



(d) Random Write (860Pro)

Fig. 23. Performance of StRAID with different batching delays in two-stage submission.

by combining the two-stage submission and the log-write buffer, the increased batching latency counteracts the performance gain from write buffering. In conclusion, the default batching latency settings strike a balance between optimizing aggregation efficiency and reducing IO processing latency, to fully utilize the performance of ultra-fast SSDs.

Besides, the results also indicate that these default parameters perform well on lower-performance 860Pro SSDs. StRAID exhibits a higher batching delay on 860Pro SSDs compared to 980Pro SSDs, resulting in improved aggregation efficiency. As a result, only increasing the batching delay (e.g., by 2x) would not improve the overall write performance.

Partial-write Buffering. We analyze the runtime features of the write buffering mechanism on the performance of StRAID. We evaluate StRAID, StRAIDd, and MD with continuous partial-stripe writes and measure their runtime throughput respectively. In addition, we explore the impact of Log-buffer size on RAID performance by changing the default buffer size (i.e., `StRAID-24G`, 2GB for each sublog and 24GB in total) to a smaller size (i.e., `StRAID-6G`, 0.5GB for each sublog and 6GB in total).

Figure 24 shows that StRAID outperforms StRAIDd and MD by 1.9× and 6.1× respectively when the Log-Buffer is empty. When the written data reaches 12GB (i.e., the capacity of a ping-pong LogGroup in `StRAID-24G`), the frontend write performance is reduced by 15% in average because the bandwidth is congested by background reclamation. When the written data reaches 30GB, StRAID stops buffering because the capacity of Log-Buffer is full, and the performance drops to that of StRAIDd. After buffer reclamation is completed (i.e., written data reaches 38GB), StRAID could restart the write buffering. Moreover, the results show that the Log-buffer sizes have less impact on the peak throughput of StRAID, with an average difference of less than 4%. This is because StRAID could write multiple sublogs concurrently to fully exploit the high inter-parallelism of SSDs. However, a small Log-buffer capacity can cause frequent buffer reclamation under intensive workloads and lead to system performance fluctuations.

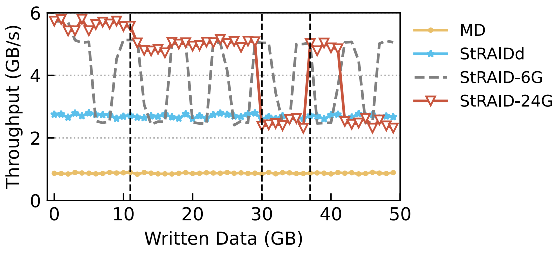


Fig. 24. The runtime features of the write buffering mechanism in StRAID.

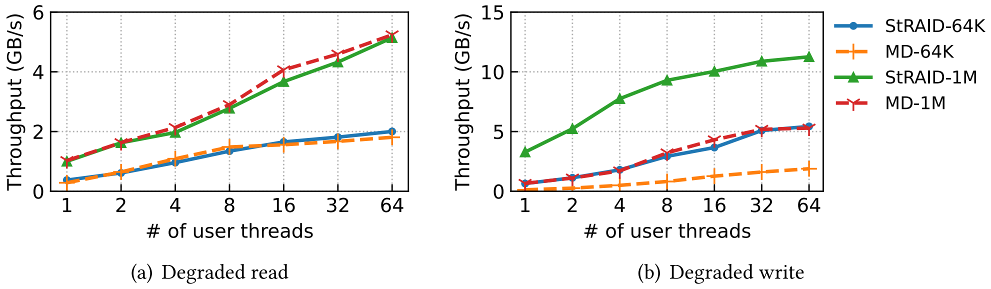


Fig. 25. Read and write performance on degraded StRAID and Linux MD.

In summary, the two-stage submission can effectively compensate for the performance degradation of traditional parity-based RAID. In addition, a user could increase the capacity of Log-buffer for better handling bursty workloads.

6.5 Resync and Degraded Mode

We assess the performance of StRAID and Linux MD in the degraded mode under RAID5. One random SSD in the RAID array is set as failed. Then, a varying number of UTs issue reads and writes of 64KB and 1MB size, respectively. StRAID will disable the write buffer in such case. Results in Figure 25 show that the read throughput of degraded StRAID and Linux MD is almost the same, with an average difference of less than 5%. Meanwhile, the write performance of degraded StRAID is 50-70% higher than that in Linux MD with multiple UTs. This is because the processing flow of write operation in degraded mode is basically the same as that in the normal mode. In addition, StRAID and MD apply the same resync approach.

7 RELATED WORKS

SSD-aware RAID SSD-based RAIDs have been extensively studied and can be roughly classified into three groups: 1) taming tail-latency by alleviating GC impact [33, 69, 70, 75]; 2) enhancing data reliability by optimizing parity distribution or conducting wear leveling across SSDs [3, 41, 66]; and 3) mitigating the overhead of parity writes [8, 14, 24, 29, 74]. StRAID focuses on the multi-threaded processing architecture in RAID systems and can complement these works.

All-Flash-Array Systems RAID for **AFA (all-flash-array)** systems have been studied for RAID data layout optimization [50, 77] and taming tail-latency by alleviating GC impact [33, 59]. FusionRAID [28] improves the latency performance of the RAID system for SSD pools by leveraging the Latin-square-based deterministic addressing methods proposed in RAID+ [77], while proposing an out-of-place write method for optimizing parity-updates. SWAN [33] tames tail-latency by alleviating SSD GC impact in an all-flash-array system. Complementary to them, StRAID focuses on the stripe-write process on multi-core processors and fast SSDs without any modification of the RAID data layout. Therefore, StRAID as a new stripe-handling engine can be used in AFA systems to exploit modern hardware with high internal parallelism.

Parity Write Optimization The stripe aggregation method is widely studied to construct full-stripe writes for reducing the write-induced reads or reducing the number of parity writes to SSDs. The Linux MD performs stripe aggregation by postponing user writes to absorb more requests for reducing actual I/Os. However, this approach increases the latency for handling requests, which would hurt the overall performance for low-latency SSDs. Besides, it cannot speed up random partial-writes. In contrast to the stripe-cache in Linux MD, the two-stage stripe submission mechanism in StRAID leverages a short batching delay to ensure the efficiency of each handling thread, thus achieving better throughput without sacrificing I/O latency.

The RAID buffering mechanisms have been studied to absorb partial-stripe writes for reducing the write-induced reads and mitigating parity updates to SSDs. Previous works [14, 15, 24, 63, 74] use an extra NVRAM or SSD as a buffer to absorb incoming write data and/or parity information for delaying parity updates. Existing RAID systems [19, 28, 68] first steer all writes to a logging zone and then write back to the primary RAID in the background. LDM [68] steers small-sized writes to a logging space and merges them in the background. Such an aggregation approach deprives the opportunity to construct full-stripe writes in memory to bypass the buffer, thus doubling the amount of data written to SSDs. Besides, a single buffering device could become the bottleneck under intensive workloads.

Compared with these efforts, StRAID opportunistically aggregates SS-writes into full-stripes with TSS, while conditionally buffering partial-writes to the Log-Buffer. It archives to maintain the efficiency of stripe aggregation while minimizing the disk-I/O amplification and requirements on extra storage. Besides, StRAID proposes the two-dimensional Log-Buffer architecture and the parallel write I/O processing to fully exploit inter-parallelism of SSDs, preventing the buffering devices from becoming a bottleneck.

Block IO Scheduling Prior studies on block IO scheduling are focused on optimizing multi-queue management including prioritization [37], fairness queuing [22, 79], policy-based storage provisioning and management [2, 62] and providing low scheduling latency [23]. StRAID is a RAID stripe-write engine on top of and thus complementary to these block IO scheduling approaches. Additionally, compared with other RAID systems that adopt FTL-level block I/O scheduling [34, 67, 80], StRAID considers SSDs as black boxes, making it highly portable and non-intrusive.

Multicore Optimization Previous studies have addressed the scalability issues in key-value stores [11, 12], file systems [5, 13, 43], volume management [36] and block drivers [23] with multicore processors and high-performance devices (e.g., SSDs and NVMs). MAX [43] demonstrates that lock contentions are the major reasons for poor scalability in file systems. These works exploit the potentials of parallelism on multicore processors and fast SSDs through localized key data structures and fine-grained lock designs. The Linux kernel contributors optimize lock mechanisms to improve read performance [52]. In this paper, StRAID focuses on optimizing the write path of the MD parity-RAID architecture and addresses the software overhead in handling stripe writes.

8 CONCLUSION

We experimentally reveal that Linux MD with parity-based RAIDs cannot fully exploit the potentials offered by high-performance SSDs due to the architectural drawback of centralized stripe-writes. We propose a stripe-threaded parity-RAID (StRAID) to efficiently handle stripe-writes in parallel. StRAID introduces a two-stage stripe submission mechanism for aggregating partial-stripe writes and a parity cache for hot parity-accesses. Through extensive trace-driven evaluations, StRAID is shown to significantly and consistently outperform MD parity-based RAID in write performance without sacrificing read performance.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.* ACM, 229–240.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.* USENIX Association, 775–787.

[3] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential RAID: Rethinking RAID for SSD reliability. In *European Conference on Computer Systems, Proceedings of the 5th European Conference on Computer Systems, EuroSys 2010, Paris, France, April 13-16, 2010.* ACM, 15–26.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel, et al. 2010. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, Vol. 10. 1–8.

[5] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* ACM, 69–86.

[6] J. Bonwick and B. Moorei. ZFS: The last word in file systems. http://opensolaris.org/os/community/zfs/docs/zfslast.pdf

[7] John F. Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 2015. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015.* IEEE Computer Society, 233–242.

[8] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation. *IEEE Trans. Parallel Distributed Syst.* 29, 10 (2018), 2241–2253.

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation. *IEEE Trans. Parallel Distributed Syst.* 29, 10 (2018), 2241–2253.

[10] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011.* USENIX, 77–90.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021.* USENIX Association, 17–32.

[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020.* ACM, 1077–1091.

[13] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021.* USENIX Association, 81–95.

[14] Ching-Che Chung and Hao-Hsiang Hsu. 2014. Partial parity cache and data cache management method to improve the performance of an SSD-based RAID. *IEEE Trans. Very Large Scale Integr. Syst.* 22, 7 (2014), 1470–1480.

[15] 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, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* ACM, 1683–1694.

[16] W. V. Courtright, G. Gibson, M. Holland, and J. Zelenka. 1996. A structured approach to redundant disk array implementation. In *Proceedings of IEEE International Computer Performance and Dependability Symposium.* 11–20.

[17] Samsung Electronics. 2021. Samsung develops high-performance PCIe 5.0 SSD for enterprise servers. https://www.samsungsemiconstory.com/global/samsung-develops-high-performance-pcie-5-0-ssd-for-enterprise-servers/

[18] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. 2017. Exploiting intra-request slack to improve SSD performance. In *Proceedings of the Twenty-Second Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. ACM, 375–388.

[19] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. 2011. DiskReduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *SC11* (2011).

[20] Dean Gonzales. 2015. PCI express 4.0 electrical previews. In *PCI-SIG Developers Conference*.

[21] Mingzhe Hao, Gokul Soundararajan, Deepak R. Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. USENIX Association, 263–276.

[22] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-queue fair queuing. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 301–314.

[23] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux storage stack for μs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 113–128.

[24] Soojun Im and Dongkun Shin. 2011. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Computers* 60, 1 (2011), 80–92.

[25] NETAPP INC. 2010. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/data-ontap-8/

[26] Intel. 2020. ISA-L performance report. https://01.org/intel-storage-acceleration-library-open-source-version/documentation/documentation

[27] Intel. 2021. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology

[28] 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 2021, February 23-25, 2021*. USENIX Association, 355–370.

[29] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. 2011. RAID6L: A log-assisted RAID6 storage architecture with improved write performance. In *IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST 2011, Denver, Colorado, USA, May 23-27, 2011*. IEEE Computer Society, 1–6.

[30] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 804–818.

[31] The kernel development community. 2020. RAID 4/5/6 cache. https://www.kernel.org/doc/html/latest/driver-api/md/raid5-cache.html

[32] Ram Kesavan, Jason Hennessey, Richard Jernigan, Peter Macko, Keith A. Smith, Daniel Tennant, and Bharadwaj V. R. 2019. FlexGroup volumes: A distributed WAFL file system. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 135–148.

[33] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. ACM, 799–812.

[34] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. 2012. Coordinating garbage collection for arrays of solid-state drives. *IEEE Trans. Comput.* 63, 4 (2012), 888–901.

[35] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. ACM, 219–231.

[36] Pradeep Kumar and H. Howie Huang. 2017. Falcon: Scaling IO performance in multi-SSD volumes. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 41–53.

[37] Kyber. 2017. multiqueue I/O scheduler. https://lwn.net/Articles/720071/

[38] Jing Li, Peng Li, Rebecca J. Stones, Gang Wang, Zhongwei Li, and Xiaoguang Liu. 2020. Reliability equations for cloud storage systems with proactive fault tolerance. *IEEE Trans. Dependable Secur. Comput.* 17, 4 (2020), 782–794.

[39] Shaohua Li. 2013. raid5: Make stripe handling multi-threading. https://lwn.net/Articles/563142/

[40] Shaohua Li. 2013. raid5 offload stripe handle to workqueue. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=851c30c9badfc6b294c98e887624bff53644ad21

[41] Yongkun Li, Patrick P. C. Lee, and John C. S. Lui. 2016. Analysis of reliability dynamics of SSD RAID. *IEEE Trans. Computers* 65, 4 (2016), 1131–1144.

[42] Yongkun Li, Biaobiao Shen, Yubiao Pan, Yinlong Xu, Zhipeng Li, and John C. S. Lui. 2017. Workload-aware elastic striping with hot data identification for SSD RAID arrays. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 36, 5 (2017), 815–828.

[43] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A multicore-accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. 877–891.

[44] Linux. 2015. Linux perf. https://perf.wiki.kernel.org/

[45] Linux. 2020. HDFS-RAID. https://wiki.apache.org/confluence/display/HADOOP2

[46] Linux. 2020. Linux RAID. https://raid.wiki.kernel.org/index.php/Linux_Raid

[47] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 2019. Analysis of and optimization for write-dominated hybrid storage nodes in cloud. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 403–415.

[48] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 137–149.

[49] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. 2015. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015*. ACM, 177–190.

[50] Richard R. Muntz and John C. S. Lui. 1990. Performance analysis of disk arrays under failure. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Morgan Kaufmann, 162–173.

[51] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*. USENIX, 253–267.

[52] Gal Ofri. 2021. raid5 avoid device lock in read one chunk. https://github.com/torvalds/linux/commit/97ae27252f4962d0fcc38ee1d9f913d817a2024e

[53] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144.

[54] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. USENIX Association, 217–231.

[55] Tirthak Patel, Suren Byna, Glenn K. Lockwood, Nicholas J. Wright, Philip H. Carns, Robert B. Ross, and Devesh Tiwari. 2020. Uncovering access, reuse, and sharing characteristics of I/O-intensive files on large-scale production HPC systems. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 91–101.

[56] David A. Patterson, Garth A. 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, Chicago, Illinois, USA, June 1-3, 1988*. 109–116.

[57] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. 1994. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.* 43, 5 (1994), 548–559.

[58] 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 2016, Santa Clara, CA, USA, February 22-25, 2016*. USENIX Association, 67–80.

[59] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott A. Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 463–474.

[60] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.* 41, 1 (2016).

[61] Michael Hao Tong, Robert L. Grossman, and Haryadi S. Gunawi. 2021. Experiences in managing the performance and reliability of a large-scale genomics cloud platform. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 973–988.

[62] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 449–462.

[63] Jiguang Wan, Wei Wu, Ling Zhan, Qing Yang, Xiaoyang Qu, and Changsheng Xie. 2017. DEFT-cache: A cost-effective and highly reliable SSD cache for RAID storage. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 102–111.

[64] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 559–571.

[65] Shucheng Wang, Qiang Cao, Ziyi Lu, and Jie Yao. 2022. Mlog: Multi-log write buffer upon ultra-fast SSD RAID. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, August 29 - September 08, 2022.* ACM, 1–11.

[66] Wei Wang, Tao Xie, and Abhinav Sharma. 2016. SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays. *ACM Trans. Storage* 12, 3 (2016), 10:1–10:21.

[67] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. 2018. Overcome the GC-induced performance variability in SSD-based RAIDs with request redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 822–833.

[68] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. 2016. LDM: Log disk mirroring with improved performance and reliability for SSD-based disk arrays. *ACM Trans. Storage* 12, 4 (2016), 22:1–22:21.

[69] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. 2019. HotR: Alleviating read/write interference with hot read data replication for flash storage. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019.* IEEE, 1367–1372.

[70] Suzhen Wu, Weidong Zhu, Gui xin Liu, Hong Jiang, and Bo Mao. 2018. GC-aware request steering with improved performance and reliability for SSD-based RAIDs. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018.* IEEE Computer Society, 296–305.

[71] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of Intel Optane persistent memory: A close look at its On-DIMM buffering. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022.* ACM, 488–505.

[72] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and actions: What we learned from 10k SSD-related storage system failures. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019.* USENIX Association, 961–976.

[73] Gaoxiang Xu, Dan Feng, Zhipeng Tan, Xinyan Zhang, Jie Xu, Xi Shu, and Yifeng Zhu. 2019. RFPL: A recovery friendly parity logging scheme for reducing small write penalty of SSD RAID. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019.* ACM, 23:1–23:10.

[74] Gaoxiang Xu, Zhipeng Tan, Dan Feng, Yifeng Zhu, Xinyan Zhang, and Jie Xu. 2018. Cap: Exploiting data correlations to improve the performance and endurance of SSD RAID. In *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018.* IEEE Computer Society, 59–66.

[75] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017.* USENIX Association, 15–28.

[76] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020.* USENIX Association, 169–182.

[77] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 2018. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018.* USENIX Association, 279–294.

[78] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-end data integrity for file systems: A ZFS case study. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010.* USENIX, 29–42.

[79] Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu, and Hai Jin. 2019. Preemptive multi-queue fair queuing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019.* ACM, 147–158.

[80] You Zhou, Fei Wu, Weizhou Huang, and Changsheng Xie. 2021. LiveSSD: A low-interference RAID scheme for hardware virtualized SSDs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 40, 7 (2021), 1354–1366.

Received 1 January 2023; revised 4 July 2023; accepted 28 September 2023