StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs

Shucheng Wang, Qiang Cao, and Ziyi Lu, *Wuhan National Laboratory for Optoelectronics, HUST;* Hong Jiang, *Department of Computer Science and Engineering, UT Arlington;* Jie Yao, *School of Computer Science and Technology, HUST;* Yuanyuan Dong, *Alibaba Group*

https://www.usenix.org/conference/atc22/presentation/wang-shucheng

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs

Shucheng Wang[1], Qiang Cao[1*], Ziyi Lu[1], Hong Jiang[2], Jie Yao[3] and Yuanyuan Dong[4]
[1]*Wuhan National Laboratory for Optoelectronics, HUST*,
[2]*Department of Computer Science and Engineering, UT Arlington*,
[3]*School of Computer Science and Technology, HUST*, [4]*Alibaba Group*

Abstract

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, StRAID presents a two-phase stripe write mechanism to opportunistically aggregate stripe-associated writes to minimize write I/Os; and designs a parity cache to reduce write-induced read I/Os on parity disks. We evaluate a StRAID prototype with a variety of benchmarks and real-world traces. StRAID is demonstrated to consistently outperform MD by up to 5.8 times in write throughput without affecting the read performance.

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 [11, 19, 35] including distributed datacenter storage systems [49, 70] 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 [27, 64], machine learning [8], enterprise storage [48], and cloud services [1, 32, 38, 46, 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 [9, 16, 67], reduce garbage-collection induced performance jitter [23, 33, 42], and optimize AFA using declustering RAID approach to balance load within devices and reduce tail-latency [30, 75]. 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) [45], 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

*Corresponding author. Email: caoqiang@hust.edu.cn

up to 29GB/s [29], 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 model, but achieves a limited performance gain due to severe lock contention on the centralized data structures.

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 further designs a two-phase stripe submission mechanism that opportunistically aggregates subsequent incoming writes belonging to the same stripe within a limited time window. Meanwhile, StRAID proposes a parity-block cache to speed up frequent write-induced parity reads. 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 offers sufficient inexpensive-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 to unleash the full performance potentials of modern hardware. We also present two key techniques, opportunistic stripe-aggregation and parity-block cache, to improve the performance of partial-stripe writes.
- 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 5 and describe related works in Section 6 . Section 7 concludes this paper.

2 Background

2.1 RAID Systems

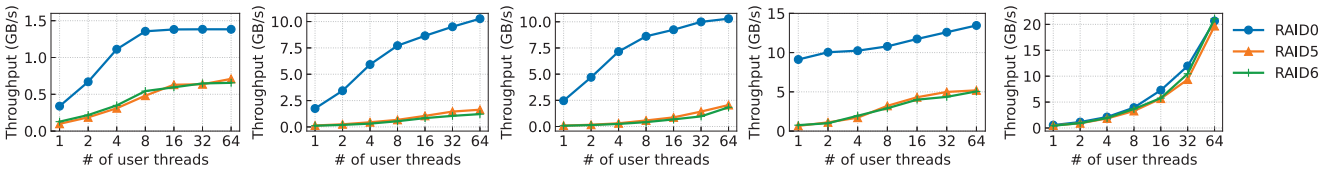
Redundant Array of Inexpensive Disks (RAID) [56] is a classic system-level approach that combines multiple disks to improve performance, reliability and capacity simultaneously. Over the past decades, RAID has been used ubiquitously to construct and manage efficient storage servers, distributed storage [5, 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 [9, 30]. 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) [45], 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



(a) 860PRO (64KB Write) (b) 970PRO (64KB Write) (c) 980PRO (64KB Write) (d) 980PRO (1MB Write) (e) 980PRO (64KB Read)

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

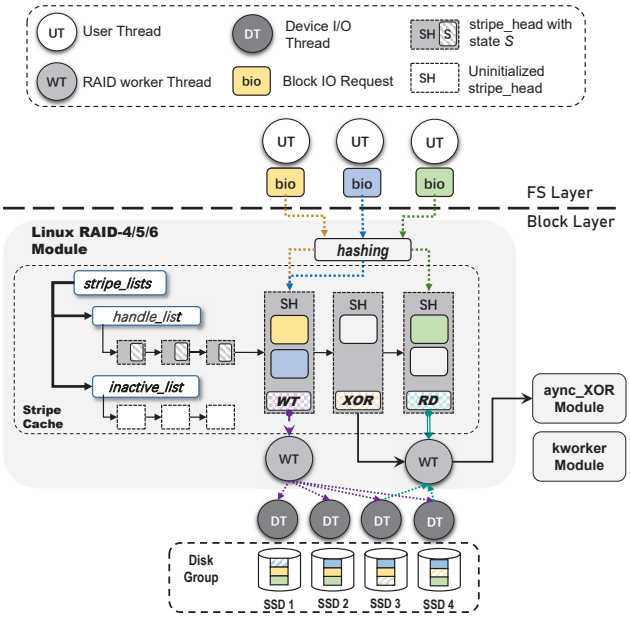


Figure 2: Architecture of Linux MD parity-based RAID.

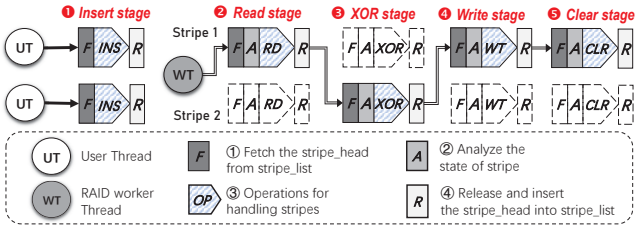


Figure 3: Stripe-write workflow of parity-based MD RAID.

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 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) [17]. 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); 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) [30]. 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.

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

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 v4.1

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

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.

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 2.4x and 3.6x 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.

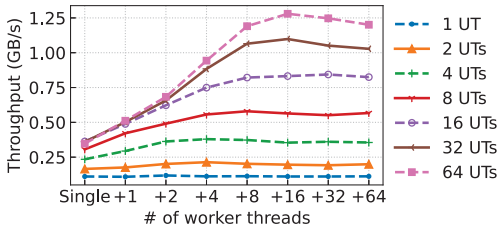


Figure 4: Write throughput of MD RAID5 under the multi-worker mechanism.

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

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 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

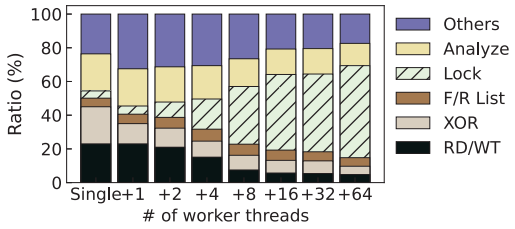


Figure 5: Breakdown of CPU cycles on key functions and locks of the worker threads in Linux MD.

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 7x 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-phase write submission and a parity cache.

4.1 Architecture

Figure 6 illustrates the StRAID architecture for parity-RAID. StRAID does not change the data layout of the legacy MD. It persists RAID's metadata at the pre-defined location of each disk. Each user thread (*UT*) pushes a block-write to a dedicated worker thread (*WT*) that exclusively handles its corresponding stripe. Multiple WTs process their own stripes independently, exploiting the intrinsic data parallelism among stripes. StRAID pre-allocates at least 256 WTs in the WT Pool to alleviate frequent thread creation/destroy overhead in runtime.

A normal stripe-write process in StRAID can be divided into 6 consecutive stages of ❶ initializing stripe_heads and inserting bios (INS); ❷ reading parity/data chunks (RD); ❸ performing I/O batching (BAT); 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). A notable workflow difference between StRAID in Figure 6 and the legacy MD in Figure 3 is that

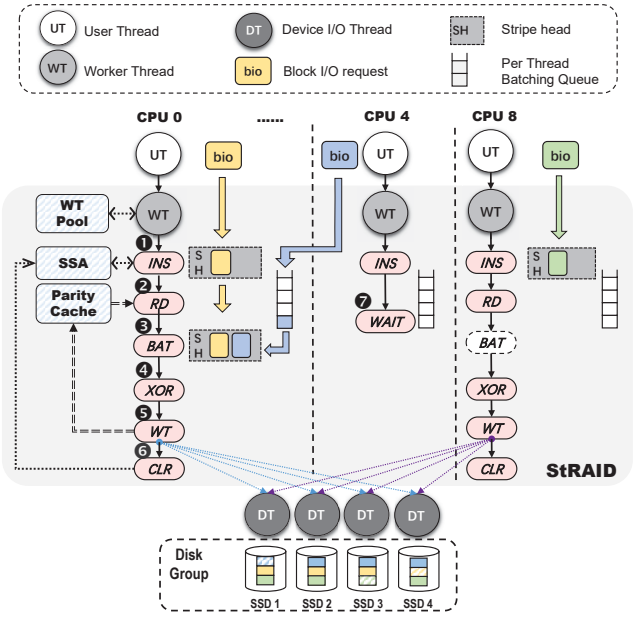


Figure 6: Architecture and process flow of StRAID

the latter's stages of ① stripe acquisition, ② analysis and ④ stripe release are 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 (§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 optimize partial-stripe writes, StRAID presents a two-phase stripe submission mechanism (§4.3) to opportunistically aggregate SS-writes by employing a batching queue per WT. Further, StRAID employs a parity cache (§4.4) in memory to buffer hot parity blocks, for significantly mitigating write-induced parity-reads.

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.

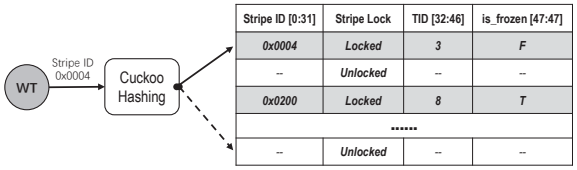


Figure 7: Stripe state table.

SST adopts a hash table to index up to 4096 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-phase Stripe Submission

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 Eq.1 and Eq.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 2x read I/Os and 2x write amplification (one data-block write and one parity-block write) with optimal RMW strategy. As WS increases, the amount of write-induced-read data decreases (0 for a full-stripe write). The write amplification is larger than 1.2x on RAID5 with no disk failures.

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

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

Existing optimizations for partial-stripe writes can be characterized by the three general approaches of write-aggregation [45], dynamic stripe size [7, 76], and parity logging [9, 10, 15, 71, 72]. 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. RAIDZ [7] uses a dynamic stripe size mechanism to eliminate partial-stripe writes, but assumes the support of the ZFS file system. The logging approach first persists incoming writes to an auxiliary fast-disk (e.g., SSD or NVM), and then rewrites the relevant stripes to original locations in the background, thus leading to at least 2x write amplification and bottlenecks from log-devices.

Two-phase Stripe Submission Without a global stripe-cache, StRAID designs a two-phase stripe submission mechanism to opportunistically absorb SS-writes. StRAID divides the stripe-write process into two phases: a batching phase and a frozen phase. Specifically, Figure 8 (referred by circled numbers) and Algorithm 1 (referred by line numbers) describe the two-phase 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 it determines the reconstruction method (RCW/RMW) for this stripe and reads the required parity/data blocks from the disk (*line 4-6*). 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 14*). It inserts bios belonging to this stripe to the batching queue of the handling thread *WT1* (*Time ③, line 15*) and then suspends itself.

When *WT1* completes its batching phase, it immediately transitions the stripe into the frozen phase (*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 re-executes parity read (if required) in accordance with the aggregated stripe-write, and performs XORing and data/parity writes to reconstruct the stripe. 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 12*). Next, *WT3* successively acquires the *Stripe Lock* to handle its requests on the stripe.

In contrast to the stripe-cache approach for aggregating SS-writes used in Linux MD, the novelty of the two-phase writing approach in StRAID leverages the latency of executing a reconstruction read in the batching phase to opportunistically aggregate incoming SS-writes. It ensures the efficiency of each handling thread, thus achieving better throughput without sacrificing I/O latency.

4.4 Parity Cache

Partial-stripe writes induce frequent parity-block accesses and cause performance degradation. To alleviate this problem, StRAID further designs a parity cache to keep hot parity-blocks in the memory to reduce disk I/Os. Previous works [9, 63, 72] use the logging approach to absorb parity updates at the cost of write amplification and potential bottlenecks at the log-devices. StRAID, instead, uses the parity cache only for eliminating parity reads induced by partial-stripe writes.

Algorithm 1 Two-phase stripe submission

```
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 is partial-stripe write then
6:              read from disks
7:          set is_frozen = true in SST and pull batching bios from queue
8:          if Data is not enough for reconstruction then
9:              Re-read from disks
10:         Compute XOR and reconstruct stripe
11:         clear_SST(stripe_id)
12:         release_stripe_lock(stripe_id)
13:     else
14:         if !is_frozen(stripe_id) then
15:             insert bio to queue with TID
16:         else
17:             handled = false
18:             continue
19: Waiting for all bios to complete
```

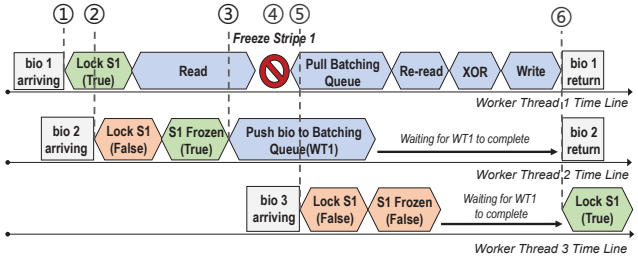


Figure 8: Workflow of two-phase stripe submission, an example with 3 concurrent worker threads (*WT1*-*WT3*) targeting the same stripe.

The stripe parity has higher access-frequency than its stripe data for partial stripe writes. Therefore, caching parity data is significantly valuable.

The architecture of parity cache is shown in Figure 9. It contains a high concurrency hash table with O(1) lookup cost, and uses an LRU-based cache replacement policy to capture frequently accessed parity data. Each hash entry corresponds to a physical stripe and contains three fields, *Stripe_ID*, *p_data* and *q_data* with the latter two pointing to cached parity data aligned with the 4KB block size. RAID5 only uses the *p_data* pointer and RAID6 uses both pointers. In addition, each entry has a fine-grained read-write lock for synchronization.

The cache module updates and queries at the block granularity, but inserts and deletes hash entries at the stripe granularity. For updates and queries, a WT needs to first insert the Stripe_ID into the LRU and searches the relevant entry. The read lock is required to access cached data because each stripe can only be updated exclusively by one WT. During the stripe-write process, a WT first searches the parity cache and acquires hit parity blocks. When missed, WT updates both disks and cache in a write-through scheme with the newest parity data after XORing.

StRAID periodically triggers a dedicated thread to clean up the stripe in the background. When the cache size exceeds

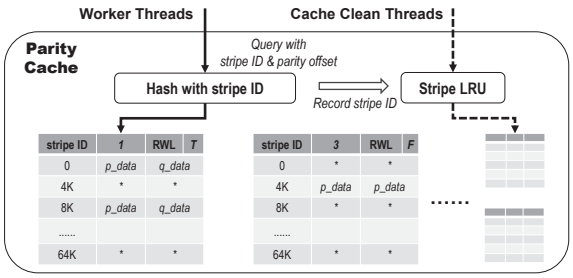


Figure 9: Architecture of the parity cache.

a threshold (64MB capacity for 16K parity blocks as default), the cleaning thread removes the entries of evicted stripes from the cache according to the LRU policy.

4.5 Recovery and Degraded Mode

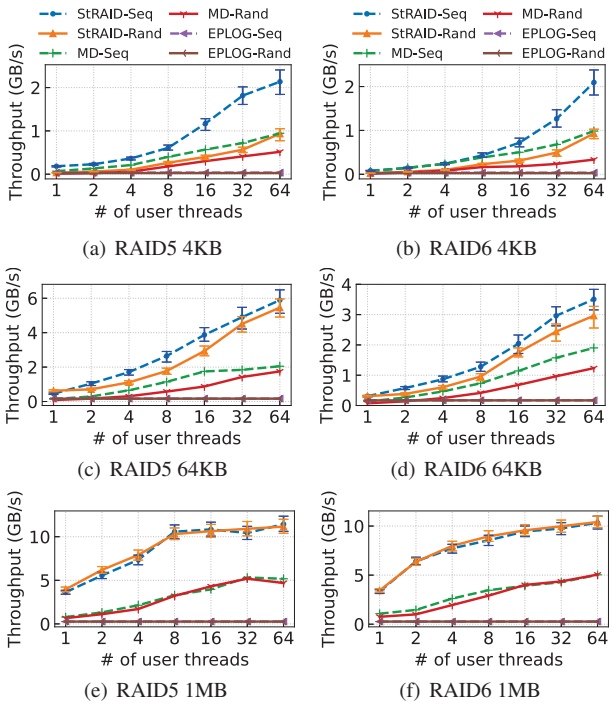
Crash Consistency and Recovery After a system crash, part of the chunk writes belonging to a stripe-write may be lost, making the stripe inconsistent between its data and parity. StRAID uses a bitmap to record the current update-state of each chunk. Compared with Linux MD, StRAID's bitmap has basically 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. Similar to MD, StRAID groups bitmap updates 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, StRAID has the option to use a journal device [3] as a writeback cache to prevent the write hole problem [22].

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 phase. 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, or reconstruction from disk replacement. We evaluate the performance of StRAID in degraded mode in Section 5.

5 Evaluation

5.1 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

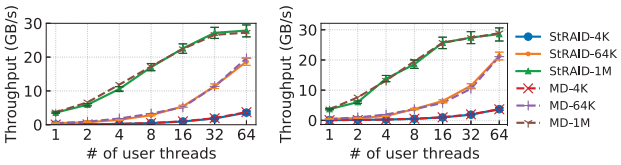


(a) RAID5 4KB (b) RAID6 4KB

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

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

Figure 10: Write scalability on three different RAID systems.



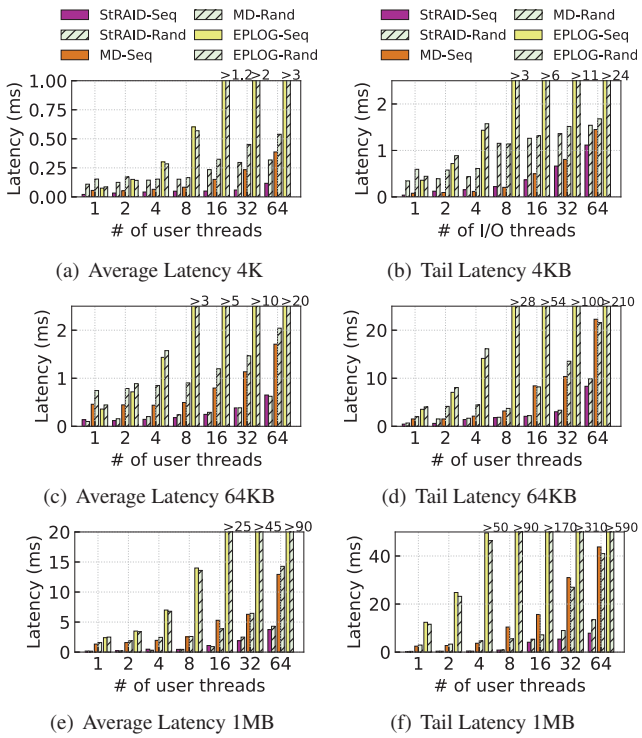
(a) RAID5 (b) RAID6

Figure 11: Read scalability of StRAID and MD.

throughput and the PCIe I/O bandwidth is 48GB/s [21], 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` 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 a 64MB-sized parity cache. 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. In addition, we compare StRAID with EPLOG [9] that mitigates parity update overhead by redirecting parity traffic to separate dedicated HDD logging devices. To prevent the log-devices from becoming a bottleneck, we replace the HDDs of EPLOG with the same type of SSDs used in the main RAID array.

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 4 as trace-driven



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

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

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

Figure 12: Average and tail latency of RAID systems.

macro-benchmarks. We implement a trace player in C++ using POSIX sync to generate direct block I/O requests to the underlying RAID systems.

5.2 Micro-benchmark

We measure the write throughput, average and tail latency, and amount of disk read/written data on `MD`, `StRAID` and `EPLOG` with RAID5 and RAID6 on 980Pro SSDs, respectively. We generate workloads with a different number of concurrent I/O-issuing threads (i.e., UTs) and varying access patterns. 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, MD and EPLOG in RAID5 and RAID6, respectively. The errorbars are added on StRAID's results. The throughput of StRAID exceeds that of MD and EPLOG respectively by up to 2.1x and 1.4x with 4KB-sized writes and 1.5x and 1.3x with 64KB-sized writes with a single UT, respectively. This is because StRAID effectively reduces the overhead of handling stripe-states. As the number of UTs increases to 64, StRAID achieves up to 2.0GB/s±0.2GB/s and 6.0GB/s±0.8GB/s peak throughput with 4KB and 64KB writes respectively, representing 2.1x/59.1x and 2.9x/35.1x performance improvement over MD/EPLOG. In addition, EPLOG achieves 1.4x-1.9x higher throughput than MD under random write at a single UT, because it avoids partial-write-induced reads with parity-logging. However, EPLOG does not scale at all with more

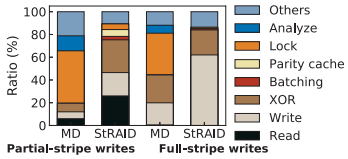


Figure 13: Breakdowns of CPU cycles on StRAID and MD

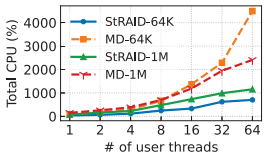
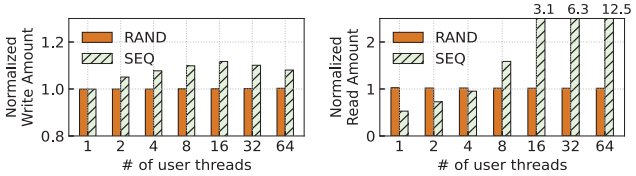


Figure 14: Total CPU utilizations



(a) Written data (b) Read data

Figure 15: Amount of data written to and read from disks by StRAID, normalized to that of MD.

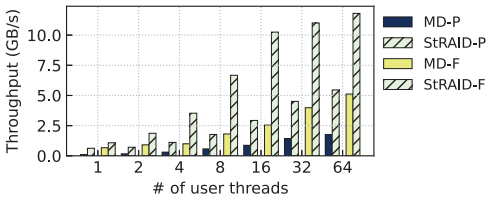


Figure 16: StRAID and MD throughput with partial-stripe *(*-P)* and full-stripe writes *(*-F)* of different chunk sizes.

Table 4: 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

UTs, because it uses a global lock for serializing each write operation.

For full-stripe writes (i.e., 1MB), StRAID achieves 4.6x/12x and 5.2x/13x higher write throughput in random and sequential cases than MD/EPLOG 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 2.1x higher than those of MD (5.2GB/s and 5.1GB/s) and 41x higher than EPLOG (0.25GB/s and 0.26GB/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.

Moreover, Figure 11 shows the read throughput of StRAID and MD with varying-size reads. The average read throughput difference between MD and StRAID is less than 5% in RAID5 and RAID6 respectively, demonstrating that StRAID does not affect read performance.

Latency and Breakdown of CPU-cycles Figure 12 shows the average and tail (99^{th}-percentile) latency under RAID5 in StRAID, MD and EPLOG, respectively. StRAID significantly outperforms MD and EPLOG in both average and tail latencies performance under 64 UTs, reducing latency by 75% and 98.2% with 4KB block-writes, 76% and 97.1% with 64KB partial-stripe writes, and by 69% and 95.2% with full-stripe writes. StRAID reduces 22%-67% tail latencies from MD under 64 UTs. The tail latency of EPLOG is 6.5x-42.1x higher than StRAID under multiple UTs, since the global lock in EPLOG makes its average and tail latencies much higher than those of MD and StRAID.

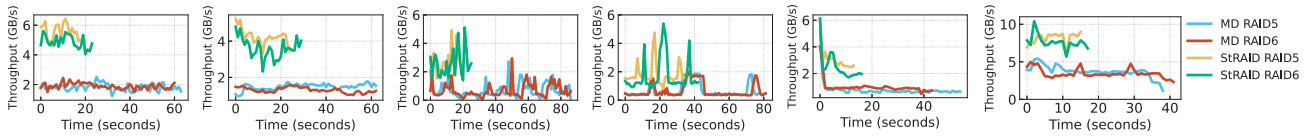
To better understand the reasons behind StRAID's superiority, Figure 13 shows the breakdown of the CPU-cycles of key functions consumed by MD and StRAID with 64 UTs issuing random writes, respectively. For partial-stripe writes, the combined CPU-cycles on XORing and disk I/Os account for 76% of the total CPU usage in StRAID, while that of MD is less than 20%. The average stripe-write handling overhead of StRAID, i.e., 60μs, is about 20 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-phase submission and the parity caching use only 6% and 1.5% 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 14 show that the total CPU utilization of MD is up to 6.3x higher than StRAID with 64 UTs. Even when the number of UTs is less than 8, the CPU usage of MD is 2x 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.

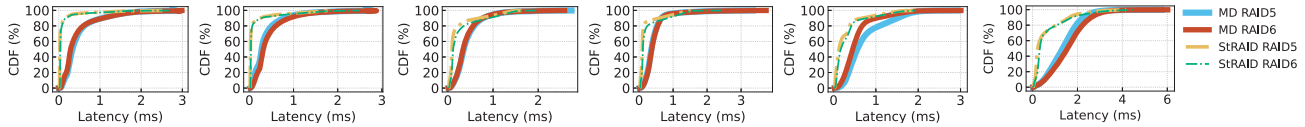
Moreover, 64KB-sized partial-stripe writes of MD (*MD-64K*) consume up to 80% more CPU than full-stripe writes (*MD-1M*) with 64 UTs. MD's inefficiency stems from its high consumption of CPU cycles required to handle in-flight partial-stripe writes. On the contrary, StRAID-64K consumes only 25% less 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.

Read/Write amplification Figure 15 shows the amount of



(a) Pangu-A (b) Pangu-B (c) prn0 (d) prxy0 (e) varmail (f) fileserver

Figure 17: Throughput of StRAID and MD on trace-driven workloads.



(a) Pangu-A (b) Pangu-B (c) prn0 (d) prxy0 (e) varmail (f) fileserver

Figure 18: Latency CDF of StRAID and MD on trace-driven workloads.

data written to and read from disks by StRAID in RAID5, normalized to that of MD. For random writes, StRAID and MD have exactly the same amount of data written and data read because stripe-write aggregation is rare for random writes. For sequential writes, however, the amount of data written in StRAID is up to 12% larger than MD. This is because the two-phase submission in StRAID has a smaller aggregation window (i.e., proportional to SSD-read latency), which is slightly less efficient than Linux MD's active delays of sequential stripe writes.

For sequential writes, the amount of data read in StRAID and MD varies significantly with different numbers of UTs. With a single UT, the amount of read data in StRAID is 1/2 of that in MD, because the parity caching mechanism effectively reduces parity reads. However, with 64 UTs as the worst case, StRAID reads 12x more data than MD. 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., 1.1% of user-written data). In contrast, a StRAID's worker thread immediately executes reading parity/data chunks before performing write-aggregation, resulting in a nearly fixed number of write-induced reads (e.g., 13.7% of user-written data). However, 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 5.2x higher than MD.

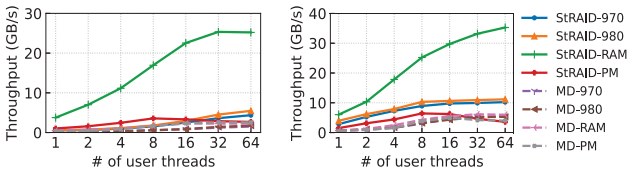
Chunk Sizes We evaluate the effect of chunk size configuration on the performance of StRAID and MD with 64KB-sized writes in RAID6. The chunk size is set to 8KB for the full-stripe-write case (*StRAID-F* and *MD-F*), and 64KB for the partial-stripe-write case (*StRAID-P* and *MD-P*), respectively. Figure 16 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 1.9x higher than StRAID-P. Similarly, the throughput of MD-F is up to 3.1x 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 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.

5.3 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 4 summarizes the characteristics of these workloads, most of which are read-write mixed or write-dominated. In the experiments, we enable 32 WTs in MD and StRAID, and evaluate them in both the RAID5 and RAID6 levels with a chunk size of 8KB. We invoke 32 user threads to replay these traces continuously, mimicking high-intensity workloads.

Figure 17 shows the throughput of StRAID and MD over time. StRAID achieves up to 2.8x higher throughput than MD, and shortens the total running time by an average of 64% across 6 workloads. In the fileserver workload, StRAID achieves peak and average throughput of 10.3GB/s and 7.9GB/s respectively, in contrast to their MD counterparts of 5.0GB/s and 3.2GB/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 3.1x and 3x 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 in RAID5 is 10%-15% better than in RAID6 among all the workloads, because RAID5 has less parity data than RAID6.

Figure 18 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 workloads Pangu-A and Pangu-B, respectively. For the other four workloads, StRAID also has at least 49% lower average latency than MD. The median latencies of



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

Figure 19: Performances of StRAID and MD on different SSDs and RAMs.

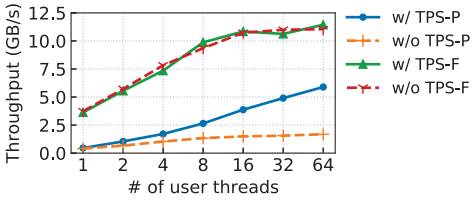


Figure 20: StRAID throughput with partial-stripe *(*-P)* and full-stripe writes *(*-F)* when running with/without two-phase stripe submission (*w/ TPS* and *w/o TPS*).

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. 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.

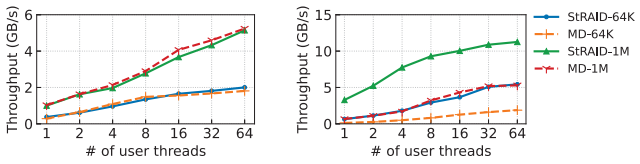
5.4 Sensitivity Study

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) [28] and lower-performance 970Pro SSDs, and compare these performances with that in 980Pros. The raw read and write bandwidth per PM can reach 6GB/s and 2GB/s [74], respectively. 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 19 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 50% and 35% higher than MD on partial-stripe and full-stripe writes, respectively. We also find that StRAID on PMs shows up to 26% higher throughput in partial-stripe writes than that SSDs. This is because PM has one order of magnitude lower read latency than SSDs, thus StRAID could handle stripes more efficiently.

Table 5: Parity cache capacity

	Written Data (GB)	Read Data (GB)	Cache Hit Rate	Average Thr. (MB/s)
StRAID-NC	54.10	13.64	-	1482
StRAID-4M	54.10	12.12	0.39	1593
StRAID-16M	54.10	12.01	0.45	1636
StRAID-64M	54.10	11.75	0.55	1711
StRAID-256M	54.10	11.23	0.57	1726
MD	53.80	6.10	-	667



(a) Degraded read (b) Degraded write

Figure 21: Read and write performance on degraded StRAID and Linux MD.

Moreover, StRAID on PM shows a throughput drop at higher than 8 UTs, because PM has a limited concurrency [74].

In addition, StRAID on RAMs delivers up to 5.8x 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 [18] in the near future.

Two-phase Submission We analyze the performance contributions of the two-phase stripe submission (TPS) mechanism of StRAID in RAID5. We run the experiment with (*w/ TPS*) and without (*w/o TPS*) two-phase submission, respectively. The request size is set to 1MB for the full-stripe-write case (**-F*), and 64KB for the partial-stripe-write case (**-P*). We issue requests with sequential access patterns. Figure 20 shows that StRAID with TPS achieves 3.5x improvement of average throughput than without TPS for partial-stripe writes at 64 UTs. The two-phase submission allows request aggregation on writes belonging to a same stripe and handles them in a batch. StRAID without TPS, by contrast, has to individually execute each writes on a stripe. Besides, the performance contribution of two-phase submission on full-stripe write load is less than 4%, because the requests targeting different stripes will not be aggregated.

Parity Cache We analyze the effectiveness of the parity cache on StRAID performance. We compare the StRAID with the parity cache disabled (*StRAID_NC*) against the StRAID with its parity cache capacity varying from 4MB to 256MB (*StRAID_4M* to *StRAID_256M*). We select the RAID5 level with 8KB chunk size as an example. We run the prxy0 workload with the most partial-stripe writes among all workloads tested, which has 9.5% write operations (0.51 million ops) with I/O sizes of 40KB or larger (i.e., full-stripe write size for 8KB chunk size), and the sequential write ratio is 66% (8.02 million ops). We measure the average throughput, cache hit rate, and amount of disk read/written data, respectively.

Results are shown in Table 5. As the parity cache capacity increases, the cache hit ratio increases from 39% at StRAID-4M to 57% at StRAID-64M, resulting in a 16% improvement of average throughput. In addition, increasing the cache capacity to beyond 64MB contributes less than a 5% increase in both the cache hit ratio and throughput, because the cache has captured most of the parity data under such conditions.

5.5 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. Results in Figure 21 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.

6 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 [33,68,69,73]; 2) enhancing data reliability by optimizing parity distribution or conducting wear leveling across SSDs [4, 41, 65]; and 3) mitigating the overhead of parity writes [9, 15, 26, 31, 72]. 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, 75] and taming tail-latency by alleviating GC impact [33, 59]. FusionRAID [30] improves the latency performance of the RAID system for SSD pools by leveraging the Latin-square-based deterministic addressing methods proposed in RAID+ [75], 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. Previous works [15, 16, 26, 63, 72] use an NVRAM or SSD as a cache to absorb incoming write data and/or parity information and delay parity updates with extra devices. In contrast, the parity cache in StRAID is located in memory and only used to accelerate read I/Os for stripe reconstructions. Existing systems [20, 30, 67] first steer all writes to a logging zone and then write back to the RAID zone in the background. Such an aggregation approach requires additional storage and double the amount of data written to SSDs. In comparison to these efforts, StRAID performs an in-place update for stripe writes and opportunistically aggregates writes without extra storage requirements.

Block IO Scheduling Prior studies on block IO scheduling are focused on optimizing multi-queue management including prioritization [37], fairness queuing [24, 77], policy-based storage provisioning and management [2, 62] and providing low scheduling latency [25]. 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, 66, 78], 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 [12, 13], file systems [6, 14, 43], volume management [36] and block drivers [25] 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.

7 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-phase 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 shepherd and reviewers for their valuable feedback and suggestion. This work was supported in part by NSFC No. 62172175, Creative Research Group Project of NSFC No. 61821003, National key research and development program of China under Grant 2018YFA0701800, the US National Science Foundation Grant CNS-2008835, and Alibaba Innovative Research.

References

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 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*, pages 229–240, 2010.

[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. Remote regions: a simple abstraction for remote memory. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[3] Apache. A journal for md/raid5. 2021. https://lwn.net/Articles/665299/.

[4] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: rethinking RAID for SSD reliability. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 15–26. ACM, 2010.

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

[6] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 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*, pages 69–86, 2017.

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

[8] John F. Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 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*, pages 233–242. IEEE Computer Society, 2015.

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

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

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

[12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 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*, pages 17–32, 2021.

[13] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091. ACM, 2020.

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

[15] Ching-Che Chung and Hao-Hsiang Hsu. 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):1470–1480, 2014.

[16] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 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*, pages 1683–1694, 2015.

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

[18] Samsung Electronics. 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/.

[19] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. Exploiting intra-request slack to improve SSD performance. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 375–388. ACM, 2017.

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

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

[22] Matthias Grawinkel, Lars Nagel, and André Brinkmann. Lonestar raid: Massive array of offline disks for archival systems. *ACM Transactions on Storage (TOS)*, 12(1):1–29, 2016.

[23] Mingzhe Hao, Gokul Soundararajan, Deepak R. Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 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*, pages 263–276, 2016.

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

[25] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 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*, pages 113–128, 2021.

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

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

[28] Intel. Intel optane dc persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology.

[29] Intel. Isa-l performance report. https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version/documentation/documentation.

[30] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity SSD arrays. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 355–370, 2021.

[31] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. 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*, pages 1–6, 2011.

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

[33] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 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*, pages 799–812, 2019.

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

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

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

[37] Kyber. multiqueue i/o scheduler. 2017. https://lwn.net/Articles/720071/.

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

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

[40] Shaohua Li. 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. Analysis of reliability dynamics of SSD RAID. *IEEE Trans. Computers*, 65(4):1131–1144, 2016.

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

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

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

[45] Linux. Linux raid. https://raid.wiki.kernel.org/index.php/Linux_Raid.

[46] Linux. Hdfs-raid. 2020. https://wiki.apache.org/confluence/display/HADOOP2.

[47] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 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*, pages 403–415. ACM, 2019.

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

[49] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In Bill Lin, Jun (Jim) Xu, Sudipta Sengupta, and Devavrat Shah, editors, *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015*, pages 177–190. ACM, 2015.

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

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

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

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

[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. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 217–231, 2021.

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

[56] David A. Patterson, Garth A. Gibson, and Randy H. Katz. 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*, pages 109–116, 1988.

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

[58] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 67–80. USENIX Association, 2016.

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

[60] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 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. Experiences in managing the performance and reliability of a large-scale genomics cloud platform. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 973–988. USENIX Association, 2021.

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

[63] Jiguang Wan, Wei Wu, Ling Zhan, Qing Yang, Xiaoyang Qu, and Changsheng Xie. 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*, pages 102–111. IEEE Computer Society, 2017.

[64] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 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} 20)*, pages 559–571, 2020.

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

[66] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. 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):822–833, 2018.

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

[68] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. 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*, pages 1367–1372, 2019.

[69] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. 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*, pages 296–305, 2018.

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

[71] Gaoxiang Xu, Dan Feng, Zhipeng Tan, Xinyan Zhang, Jie Xu, Xi Shu, and Yifeng Zhu. 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*, pages 23:1–23:10. ACM, 2019.

[72] Gaoxiang Xu, Zhipeng Tan, Dan Feng, Yifeng Zhu, Xinyan Zhang, and Jie Xu. 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*, pages 59–66. IEEE Computer Society, 2018.

[73] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 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*, pages 15–28, 2017.

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *FAST 2020*.

[75] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 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*, pages 279–294, 2018.

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

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

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