FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays

Tianyang Jiang, Guangyan Zhang, and Zican Huang, *Tsinghua University;*
Xiaosong Ma, *Qatar Computing Research Institute, HBKU;* Junyu Wei,
Zhiyue Li, and Weimin Zheng, *Tsinghua University*

https://www.usenix.org/conference/fast21/presentation/jiang

This paper is included in the Proceedings of the
19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays

Tianyang Jiang[†], Guangyan Zhang[†*], Zican Huang[†], Xiaosong Ma[‡],
Junyu Wei[†], Zhiyue Li[†], Weimin Zheng[†]
[†]*Tsinghua University,* [‡]*Qatar Computing Research Institute, HBKU*

Abstract

The use of all-flash arrays has been increasing. Compared to their hard-disk counterparts, each drive offers higher performance but also undergoes more severe periodic performance degradation (due to internal operations such as garbage collection). With a detailed study of widely-used applications/traces and 6 SSD models, we confirm that individual SSD's performance jitters are further magnified in RAID arrays. Our results also reveal that with SSD latency low and decreasing, the software overhead of RAID write creates long, complex write paths involving more drives, raising both average-case latency and risk of exposing worst-case performance.

Based on these findings, we propose *FusionRAID*, a new RAID architecture that achieves *consistent, low latency on commodity SSD arrays*. By spreading requests to all SSDs in a shared, large storage pool, bursty application workloads can be served by plenty of "normal-behaving" drives. By performing temporary, replicated writes, it retains RAID fault-tolerance yet greatly accelerates small, random writes. Blocks of such transient data replicas are created in stripe-ready locations based on RAID declustering, enabling effortless conversion to long-term RAID storage. Finally, using lightweight SSD latency spike detection and request redirection, FusionRAID avoids drives under transient but severe performance degradation. Our evaluation with traces and applications shows that FusionRAID brings a 22%–98% reduction in median latency, and a 2.7×–62× reduction in tail latency, with a moderate and *temporary* space overhead.

1 Introduction

The use of all-flash arrays (AFAs) has been increasing and projected to have a 400% market growth in the next five years [11]. For example, ANZ bank in Australia recently adopted an AFA solution with 400TB SSD arrays [31]. AFAs aggregate the IOPS and bandwidth of individual drives and compensate for SSDs' higher rate of uncorrectable errors [53] using parity-based fault-tolerance. So far, AFAs can support large numbers of SSDs (up to 5760) behind a single controller [20, 22, 49, 51, 52].

	Median latency (ms)	Avg. latency (ms)	P99 latency (ms)	Variance factor
HDD RAID (clean)	68.67	134.37	835.35	12.16
HDD RAID (aged)	69.18	133.61	826.77	11.95
SSD RAID (clean)	0.275	3.57	25.62	93.16
SSD RAID (aged)	0.307	14.11	221.03	719.96

Table 1: `Exchange` latency, HDD vs. SSD RAID

However, SSDs are less array-friendly than hard disks, which RAID [50] was initially designed for decades ago. Compared with single-disk accesses, RAID write significantly amplifies average write latency, thereby also delaying read requests. Also, SSD RAIDs have a much higher *tail-to-average latency ratio* than HDD ones [23, 38, 66].

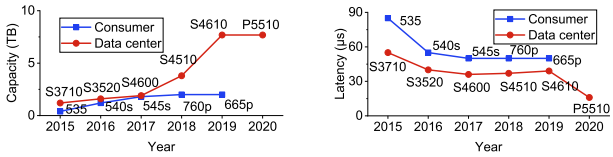
We illustrate this by testing two software RAID5 arrays, built on Seagate HDDs and Intel commodity consumer SSDs. Table 1 lists the median, average, and 99 percentile (P99) latencies measured running the write-intensive `Exchange` trace from Microsoft Production Server Traces [55]. We also report the *variance factor* [68], the ratio of P99 to median latency. For each RAID, we test its *clean* and *aged* states, using the `fio` benchmark [4]. Here we adopt an existing aging method [35], writing the whole disk sequentially and then issuing random writes with total volume exceeding its capacity, to guarantee that each random write generates invalid pages.

Our results confirm that, though SSD RAID offers much smaller median latency (over 225× lower than HDD RAID), its worst-case performance deviates more from the norm, with a much higher variance level. The average latency, as a result, is much more amplified from the median with SSD than with HDD RAID. Second, the HDD RAID appears quite resilient to aging, with hardly any visible performance degradation. The SSD RAID, on the other hand, deteriorates significantly when aged, delivering a median latency of 11.6% higher, and P99 tail 8.6× higher. Such severe performance variability makes it difficult to ensure QoS to customers [18, 23, 24, 59, 69], potentially causing significant revenue losses [47]. This problem is not specific to Linux soft-

*Corresponding author: gyzh@tsinghua.edu.cn

	Median latency (ms)	Avg. latency (ms)	P99 latency (ms)	P999 latency (ms)
SSD 0	0.049	0.68	0.42	24.40
SSD 1	0.049	1.26	0.46	**702.24**
SSD 2	0.050	0.63	0.39	30.16
SSD 3	0.049	1.64	0.53	**895.02**
SSD 4	0.050	1.71	0.64	**827.91**

Table 2: `Exchange` latency, individual aged SSDs within RAID



(a) Capacity (b) Latency

Figure 1: Trends in technical specification of Intel SSDs (no consumer SSDs issued in 2020)

ware RAID overhead: our measurement also shows a hardware controller (LSI MegaRAID 9260-8i) producing very similar average latency (3.4ms) to Linux software RAID (3.6ms) on the same SSDs.

We further examine the latency distribution of individual SSD drives within the 4+1 RAID5 array, with results listed in Table 2 (aged state only). Comparing results from both tables, one sees that when we group SSDs into RAID, for the gain in space and bandwidth aggregation, we may be trading off individual request's processing speed. Note that with the `Exchange` workload (details in Table 4), considering the `MD` default 64KB stripe unit size, the majority of requests would each land on a single drive. However, the added complexity of parity updates not only generates more work but involves more drives, rendering a P99 latency nearly 400× higher on RAID than on individual SSDs for the same workload. The last column in Table 2 highlights a challenge with SSD RAIDs: three of the drives appear to experience garbage collection (GC) during our 15-minute trace execution and have a P999 latency over 23× higher than the other two. With highly coupled operations across multiple drives, isolated tail latency from a single drive affects more requests with RAID, making the entire array more vulnerable to performance anomalies.

In this work, we first conduct a comprehensive study to investigate the sources of SSD RAID latency. More specifically, we (1) examine 5 real-application workloads on modern SSDs, plus 3 workloads from widely used storage trace repositories, and systematically characterize the behavior of their mixes in fine time granularity, (2) perform a detailed analysis of the RAID write path and identify the software overhead, which has a dependency on the I/O performance of member disks, and becomes a major component in request latency, in both average and worst-case scenarios, and (3) conduct detailed profiling on 4 consumer- and 2 datacenter-grade SSDs to characterize the device-side degradation due to flash internal activities, finding both types plagued by severe latency spikes with clearly identifiable amplitude and long duration.

We then propose a new RAID architecture, FusionRAID, designed to simultaneously reduce the average- and worst-case latencies of SSD RAID, especially for latency-critical applications. FusionRAID runs on commodity SSD arrays without requiring any special hardware support or FTL modification, instead relying on three key techniques:

- flat resource sharing across an SSD pool to utilize available I/O concurrency in serving bursty application I/Os,
- shortened write operations that use temporary, replicated writes as a prelude to long-term RAID storage, yet carefully placed "in-stripe" so that one replica can be directly converted without data migration, and
- a lightweight latency spike detection and request redirection mechanism that allows requests (both read and write) to sidestep SSDs under severe performance degradation.

We argue that our solution, which temporarily trades space for performance, aligns well with current hardware trends. Figure 1(a) and 1(b) portray changes in capacity and latency, using historical data we gathered on Intel SSD models [29]. In the past years, SSDs have become much larger (especially datacenter models) and more affordable, with smaller improvement in latency. FusionRAID uses more space to quickly absorb small, random writes (with little long-term extra space overhead). In return, it makes SSD RAIDs *several times faster on average* and *orders of magnitude faster in tail cases*, as shown in our real-system evaluation. In addition, FusionRAID proposes a new way to utilize emerging large AFAs (90 SSDs or more), such as the NetApp AFF [49], EMC VMAX [20], and Fujitsu ETERNUS [22] series.

Compared with other systems that address FTL-induced latency during writes, FusionRAID considers SSDs as black boxes, without assuming SSD internal information/control. Its novelty lies in several aspects. First, to our knowledge, this is the first work that applies RAID declustering [3, 25, 46, 71] to SSD arrays. It leverages Latin-square based deterministic addressing [71], but with a significant extension to perform explicit block mapping for its two-phase writes and out-of-place updates. Second, FusionRAID adopts a new "in-position conversion" mechanism from its "replicated" to "RAID" area, removing the extra copying required by existing hybrid systems [21, 65]. Our intensive analysis reveals that SSD spikes possess clearly identifiable amplitude and long duration, making reactive methods feasible. FusionRAID's I/O redirection, based on constant spike-detection, eliminates the need for periodic probe I/Os, used by previous methods [23, 66, 69].

2 SSD RAID Latency Source Study

2.1 Workload I/O Characteristics

I/O requests from applications are often issued in a bursty manner, so the instantaneous bandwidth of a single workload

varies significantly. When multiple workloads co-execute on a storage system, the instantaneous aggregated bandwidth is often dominated by a small number of workloads, as they read/write a large amount of data while others access a little, in a short period of time.

To assess the load behavior of representative applications, we collect five block-level traces from multiple popular data-intensive workloads running individually on SSDs. We also include three publicly available traces for diversity. The eight workloads are characterized in Table 4. Then, we examine the mixes of those eight workloads in fine time granularity.

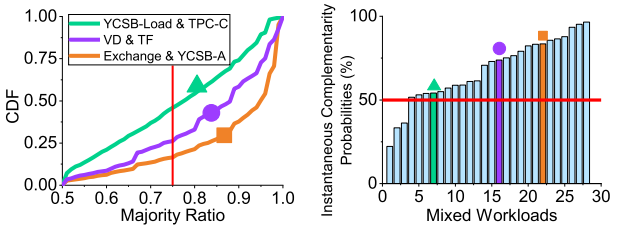
We capture all five traces from our testbed (more details in Section 5). Three of them are from running representative standard YCSB [16] workloads using RocksDB [10], a popular KV store: `YCSB-A`, `YCSB-B`, and `YCSB-Load` (specifications in Table 5). The RocksDB database size is 40GB, with 4KB KV pairs. Another latency-sensitive application trace is collected from the `TPC-C` benchmark, on a 77GB MySQL database. In addition, we trace `TensorFlow` (`TF`), which reads training datasets and checkpoints a CNN model periodically in search of better accuracy. Finally, we include three traces from the SNIA repository [58]: `VirtualDesktop` (`VD`) [40], the only recent trace from server environments, plus `Exchange` and `Proxy`, the heaviest two among a total of 49 traces within the Microsoft and SPC trace collections [30, 55–57].

Next, we carefully examine the microscopic features of mixed workloads by analysing the instantaneous bandwidth at millisecond granularity using the aforementioned 8 traces. In each 1ms timeslot, we partition a workload mix into a *giant* and a *dwarf* set, each containing the same number of workloads, with any workload in the former heavier than all in the latter. We define the ratio between the total instantaneous bandwidths of the giant set and the entire mix as *majority ratio*, R_{maj}. We claim that a workload mix possesses *instantaneous complementarity* in a timeslot with $R_{maj} > 0.75$, where the dwarf set can lend spare resources to the giant one.

Based on their average throughput, we coarsely divide the 8 traces into two groups, "heavy" and "light". We inspect all the three types of 2-workload mixes: light-light, light-heavy, and heavy-heavy. Figure 2(a) shows the CDF of R_{maj} across all timeslots for such mixes. Instantaneous complementarity appears quite frequently in the light-heavy (`Exchange` +`YCSB-A`) and light-light (`VD` +`TF`) workload mixes, making 83.6% and 73.9% of all timeslots, respectively. Even with the heavy-heavy mix (`YCSB-Load` +`TPC-C`), 54.1% of timeslots have instantaneous complementarity.

With the 8 workloads, we enumerate all 2-workload mixes and find that 25 out of the 28 have instantaneous complementarity in over half of the timeslots, with an average ratio across all 28 mixes sitting at 67.8% (Figure 2(b)). Across all 70 4-workload mixes, this average ratio of timeslots possessing instantaneous complementarity rises to 91.4%.

Implications Our analysis shows that when workloads run together, the chances of their instantaneous complementarity are quite high. This suggests that modern commodity storage servers, easily managing dozens of devices or more, can serve concurrent workloads better in an "all-for-all" model, rather than being held back for the fear of inter-workload performance interference. Involving all disks in serving the busier workloads at the moment alleviates their queue wait time, a major source of application-induced tail latency.



(a) CDF of R_{maj}

(b) Ratio of timeslots w. instantaneous complementarity

Figure 2: 2-workload mix analysis, the left showing 3 selected mixes, the right all 28 mixes

2.2 Write Overhead in SSD RAID

Parity-based RAID is unfriendly to write-intensive workloads, especially for those dominated by random small writes. The inherent read-modify-write logic makes partial-stripe writes go through a lengthy sequence of ordered operations of reads, calculation, and writes. Optimizations targeting bandwidth, such as the mechanism used in the Linux `MD` software RAID driver that postpones submission of writes in anticipation that subsequent requests fall into the same stripe, may hurt latency-sensitive applications.

Figure 3 illustrates this with a comparison between (4+1) RAID-5 arrays using three types of devices: Intel 545s SATA SSDs, Seagate 7200RPM SATA HDDs, and RAM disks. All are software RAID arrays through `MD`. We run the Microsoft Enterprise `Exchange` workload and show the breakdown of write latency across operations: read, xor, and write, with the rest categorized into software overhead. The left plot describes all requests, while the right one focuses on the 1% requests with the highest latency. Numbers at the top of the bars give the average latency values for each group.

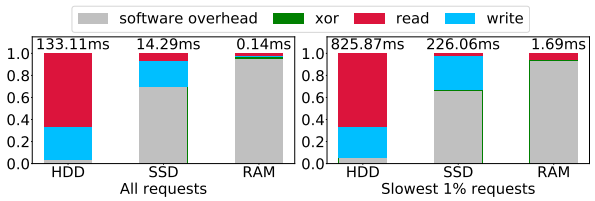


Figure 3: Write latency breakdown

With this consumer-grade SSD, software overhead already takes over as the major component of write latency, while it

	Model	Capacity (GB)	Read/Write latency (µs)	Read/Write bandwidth (MB/s)	Release year	Price ($/GB)
A	Intel 545s	240	50/60	550/500	2017	0.18
B	Intel 535	256	80/80	540/490	2015	0.15
C	Toshiba q200	240	73/36*	550/510	2017	0.20
D	Sandisk plus	240	44/193*	530/440	2017	0.17
E	Intel D3-S4510	240	36/37	560/280	2018	0.33
F	Intel DC S4500	240	36/36	500/190	2017	0.29

Table 3: Evaluated SSDs (all latency vendor specified except those marked with *)

is almost invisible with the HDD. On average, the SSD RAID tested spends 2.9× time on software overhead than on writing. This software overhead, however, involves synchronization interleaving with I/Os and is not independent of the read/write cost: with read/write cost nearly trimmed, software overhead dominates the RAM disk RAID latency, but its absolute cost is nearly two orders of magnitude smaller than on the SSD RAID. Software overhead also makes the slowest 1% requests suffer 10× the average latency, due to factors such as thread context switching and request queuing (at the block layer and the host-side dispatch queue).

As a side note, though software overhead remains the leading category, for the slowest 1% requests shown in Figure 3, SSD writes also contribute significantly to the SSD RAID tail latency, costing 20.7× the average write overhead. Section 2.3 gives a detailed discussion on this issue.

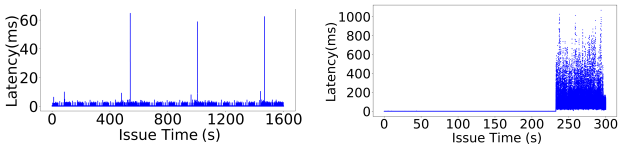
Implications Unlike an HDD array, an SSD RAID has I/O latency dominated by software overhead. Unlike a RAM disk array, it sees such overhead prolonged by actual I/Os and their coordination across disks. This suggests that a shorter write path, with fewer dependencies, may greatly reduce SSD RAID latency, both under average and worst-case scenarios.

2.3 Pathological Latency Spikes of SSDs

SSDs are known to have performance anomaly due to background I/O activities obscure to users [17, 23]. Major sources of performance variability include (1) background maintenance activities inside SSDs, in particular, garbage collection [38, 66, 69], and (2) on-SSD DRAM write buffer flushing [1]. While the existence of performance jitters is well-known [17, 23], in this paper, we quantify the distribution of their magnitude as well as duration, on multiple consumer- and datacenter-grade SSDs.

Table 3 lists basic specifications of the six commercial off-the-shelf SSDs used in our experiments, the first four being consumer-grade and the last two datacenters (DC) drives. Due to space limit, below we summarize our chief findings.

Consumer SSDs First, when running sequential writes on clean drives, all consumer models tested possess clearly periodic latency spikes with a duration between 3ms and 12ms, reaching 5-20 times of their average latency. With GC excluded under this strictly sequential workload, we attribute the regular latency spikes to on-disk DRAM buffer flushes [12]. We found these flushes block read requests as well.



(a) Clean SSD-A, seq., limited space (b) Aged SSD-E, random

Figure 4: Sample spike behavior in write tests on two SSDs (write request size at 64KB)

To repeatedly incur GC, we repeat the sequential write test but write within a 2GB logical space. We find GC produces spikes both much taller (height around 21×) and long-lasting (duration around 18×) than those incurred by buffer flushes, as illustrated in Figure 4(a). Our zoom-in analysis of I/O behaviour shows that requests are blocked during GC inside SSDs and completed immediately in a batch once GC is over. Meanwhile, there are obvious intervals (10× spike duration) between spikes due to enough spare blocks after a GC. Finally, aged SSDs see more severe and frequent spikes. The reason is likely that when the spare blocks inside SSDs run out, GC incurs more data migration and block erasures [33].

DC SSDs Datacenter SSDs behave differently. First, periodic latency spikes are not seen under sequential write workloads, unlike with consumer SSD. This is likely because DC SSDs usually have multiple cores and an optimized FTL for better coordination between background and foreground I/Os. Note such optimization may be achieved at the cost of lowering write throughput [54]: Table 3 does show the write bandwidth of the two DC SSDs at around half of that offered by cheaper consumer models.

Under heavy write workloads, however, DC SSDs cannot hide the impact of GC. Figure 4(b) shows spikes observed with random 64KB writes on SSD-E, which incurs GC faster. Under such a constant write workload, once GC is triggered, the I/O latency goes far beyond 100ms, incuring performance degradation lasting around 60s. Furthermore, spikes cannot be divided strictly, instead a new spike often arrives before the previous one ends, differing from consumer SSDs.

Implications Our profiling confirms that both consumer and datacenter SSDs suffer from severe latency spikes. Coupling such duration with an amplitude that significantly deviates from the norm, these spikes can and should be detected at runtime, to redirect incoming requests to other devices in the SSD pool. Moreover, spikes from consumer and DC SSDs behave differently, which should be handled carefully.

3 Approach Overview

We propose a new SSD RAID architecture, FusionRAID. It reduces *both average-case and tail latencies*, with solutions targeting the three problems observed in Section 2.

Design Rationale To ease innate request bursts in workloads,

FusionRAID *spreads requests to all disks* in a storage pool (such as a large commodity SSD enclosure) hosting multiple RAID volumes. Though most of their individual I/O requests can be answered by a small subset of disks, applications often have severe load bursts that directly lead to tail latency. FusionRAID trims such workload-induced tail latency by smoothing the bursts to all disks in an SSD enclosure using RAID declustering [3, 25, 46, 71]. In multi-tenant settings, this automatically lends resource elasticity to individual workloads' varying intensity.

To reduce the software overhead and inter-disk dependence in RAID writes, FusionRAID employs *replicated writes as a prelude to RAID writes*, with data lazily converted later to the more space-efficient RAID organization for long-term storage. Before such conversion, block replicas ensure the same level of fault-tolerance as the specified RAID level. *E.g.*, FusionRAID writes two copies of a block for a RAID5 volume, and three for RAID6. Doing so shortens the critical path in writes by postponing and in some cases even avoiding the long and interference-prone parity updating process. Consequently, the simpler, more independent operations of such replicated writes deliver lower (and *far more consistent*) latency. To further reduce the conversion overhead, FusionRAID places replicated blocks in a "stripe-ready" manner, in positions where sets of blocks already compose stripes (minus parity data) according to the RAID declustering algorithm adopted, to minimize data copying.

Finally, to sidestep SSDs undergoing temporary performance degradation, FusionRAID constantly watches each SSD's performance behavior to *detect temporarily unresponsive SSDs*. To this end, it uses a lightweight spike detection mechanism that issues no extra I/Os and requires no SSD internal knowledge. Uniquely enabled by RAID declustering on large SSD pools, FusionRAID easily *redirects* writes to unaffected drives, which likely remain in the majority at any given time. For reads, it could also select the less affected replica, or use existing approaches proposed by systems like ToleRAID [23] to compute a block hosted by an unresponsive SSD using parity data.

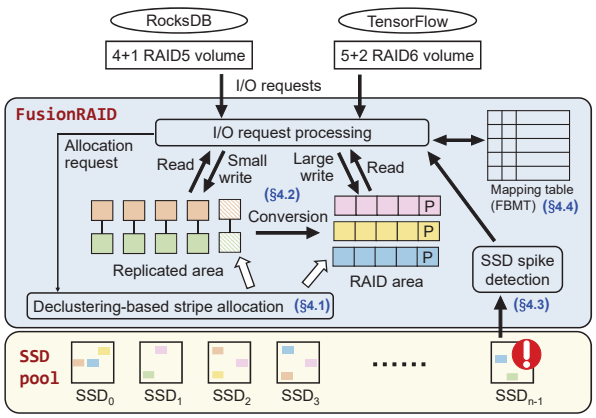


Figure 5: FusionRAID architecture

FusionRAID architecture Figure 5 illustrates the FusionRAID architecture. Multiple virtual RAID arrays (of different RAID configurations) share the same underlying pool containing dozens of commodity SSDs or more. The aggregate logical space of this SSD pool is partitioned into the *RAID* and *replicated areas*, intended for long-term, space-efficient storage and fast absorption of small, random writes, respectively. Note that there is no physical partitioning between these two *virtual* areas: actually, they are intentionally intermixed for fast conversion from the replicated to RAID storage (detailed discussion in Section 4.2). Note that though our discussion/evaluation uses RAID5 in this paper, FusionRAID applies to other RAID organizations, *e.g.*, by increasing the replication degree in the replicated area to 3 for RAID6.

Internally, FusionRAID employs a mapping table (FBMT in Figure 5) for each virtual RAID volume, to maintain the mapping of a per-volume logical block number to a FusionRAID internal logical block number (§4.4). The latter can subsequently be mapped to a logical block on a certain SSD using a deterministic RAID declustering strategy based on MOLS [71] (§4.1). For replicated writes, FusionRAID builds a list of available block pairs from a pair of stripes, enabling low-cost replicated-to-RAID conversion (§4.2).

In addition, FusionRAID performs real-time SSD latency spike detection by monitoring SSD latencies and includes the results in its decision making. In Figure 5, the last SSD is marked unresponsive and will be avoided in both reads and writes whenever possible (§4.3).

4 FusionRAID Design

4.1 Storage Organization

FusionRAID organizes the space across dozens or more SSDs, to serve bursty I/Os from concurrent workloads and provide ample alternative choices among drives to avoid using those under transient performance degradation. It does so by utilizing RAID declustering [3, 25, 46], distributing RAID stripes in a balanced way to larger arrays. The novel challenges here, unaddressed by existing RAID declustering techniques, are to design efficient and flexible support for *partial-stripe writes* to enable fast absorption of small, random writes, as well as to detour around temporarily slow drives.

To this end, FusionRAID employs a storage organization with explicit block mapping, seamlessly manages two logical areas (for replicated and striped writes), and requires low metadata space overhead. Even the transient block replicas are stored in a "stripe-aware" manner, ready for the eventual RAID storage. This way, the two logical areas are fused together, further facilitating efficient conversion from replicated to striped storage, as well as request redirection.

FusionRAID introduces *Fusion logical address space*, an internal logical block layer between the user-perceived logical and the SSD logical block address spaces, as illustrated

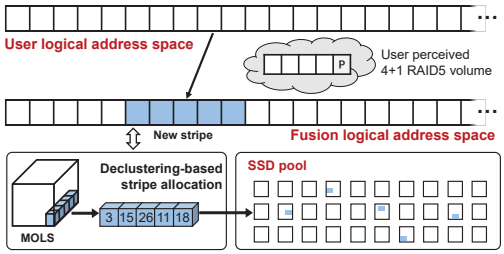


Figure 6: FusionRAID stripe allocation

in Figure 6. The mapping from user-perceived to this internal space is maintained by a block mapping table (to be detailed in Section 4.4), one per user RAID volume, which supports dynamic mapping for out-of-place writes. The mapping from a Fusion logical block to a logical SSD block is done by RAID declustering, involving a static *mapping function* instead of mapping tables. Here different declustering strategies/functions could be plugged in, such as RAID50 and pseudo-random RAID [62].

In our implementation, FusionRAID adopts the deterministic mapping proposed by RAID+ [71]. As shown in Figure 6, each FusionRAID volume will follow a 3D template based on *Mutually Orthogonal Latin Squares (MOLS)*, which resembles a large Rubik's Cube filled with disk IDs. The mapping function uses simple calculation to map a certain stripe within a user RAID volume (such as a 4+1 RAID5 configured in cloud storage, shown in the figure) to a certain subset of disks in the pool, $\langle 3, 15, 26, 11, 18 \rangle$ in this example. The advantage of MOLS-based declustering lies in its *deterministic* and *guaranteed uniform* distribution of each volume's data to all n disks within a pool, as well as low metadata overhead.

4.2 Two-phase Write Operations

Now spreading each RAID volume to the entire SSD pool to better prepare for bursty I/Os, FusionRAID further simplifies the critical path for writes, to deliver lower and more consistent latency. This is done by its *two-phase writes*, using replication as a prelude to RAID storage. While large writes are directly written in RAID stripes, smaller requests are directed to the replicated area, where instead of RAID writes with parity computation and update, we simply write two copies of data. Doing so avoids the dependency brought by the read-compute-write process, and involves fewer SSDs (hence a lower chance of running into a spike). Replicated data can be converted in the background to the more space-efficient RAID stripes. This can be carried out lazily, delayed when space is abundant, when data are hot and updated often, or when the disk pool is busy handling requests.

Two-phase write itself is not new and has been adopted by multiple systems, such as AutoRAID [65], DiskReduce [21], and LDM [67]. FusionRAID differs from them with two key innovations targeting all-flash environments, to reduce both write amplification and conversion-induced background I/Os.

First, FusionRAID performs its two-phase writes *selectively*, saving larger write requests the detour as for them the benefit does not justify the cost: their relative software overhead in RAID writes is lower, while the extra write volume generated by their replication is higher.

For larger requests, FusionRAID includes additional optimizations to lower the software overhead in RAID writes. For partial-stripe writes, it allocates a new stripe and pads the blocks untouched by the request with zero, which allows it to write both data and parity (plus appropriate metadata updates) without performing reads. Hence with requests writing x out of $(n-1)$ data blocks in a RAID stripe, going with our RAID writes as described above would require I/O of roughly n blocks, while replication requires $2x$-block writes. Therefore we set the "break-even point", $\lfloor \frac{n}{2} \rfloor$ blocks, as a size threshold to classify write requests: requests larger than this threshold do not benefit from replicated writes and would follow the aforementioned RAID write workflow.

Second, rather than following the common practice of migrating data from the replicated to the RAID area (writing the full stripe, including both data and parity blocks), we carefully create the replicated blocks "in position", so that they already *form two valid stripes* according to FusionRAID's MOLS-based template. Parity blocks are also allocated and reserved in advance, thus upon conversion, one group of the replicated blocks can be directly recorded as RAID stripe data blocks after the pre-allocated parity block is properly filled. The other group can simply be discarded.

As described in Section 4.1, FusionRAID allocates stripes using RAID declustering for both RAID and replicated writes. While the former consumes one stripe at a time, the latter consumes a pair of blocks at a time, from a pair of stripes, to store two copies of the same block. For fault tolerance, these two blocks need to sit on two different SSDs.

To this end, FusionRAID performs best-effort pairing among available stripes, using the power-of-two-choices scheme [45]. It randomly picks 4 spare stripes, selects a pair with the fewest common disk IDs, and starts block pairing by listing common disk IDs (if any) among the two stripes. It then cycles through the list diagonally, producing non-conflicting pairs. *E.g.*, if two stripes both involve disks A, B, and C, we create block pairs on A-B, B-C, and C-A. The rest of the blocks, on non-overlapping disks, are trivial to pair. This flexible scheme enables FusionRAID to easily recycle free stripes reclaimed from replicated-to-RAID conversion.

Figure 7: In-position replicated-to-RAID conversion

Figure 7 illustrates the conversion process, showing two stripes (whose blocks form 4 pairs, one of which has since been invalidated by subsequent updates while the rest carry

the RAID volume logical blocks 2, 17, and 95). Here the left stripe is converted to RAID storage simply by calculating and filling the parity block (with the invalid block included), without data movement. This easily extends to more general cases: *e.g.*, for RAID-6 we reclaim two out of the three stripes forming 3-block tuples, with two parity blocks calculated for the remaining stripe. To control the replicated area's physical space consumption, the administrator can easily configure the overall size of the logical replicated area.

Finally, background conversion from the replicated to RAID area starts from the least recently accessed stripe pair. The conversion aggressiveness can be governed by policies set by user preferences or workload characteristics. Conversion may be triggered by many different configurable thresholds, such as the number of spare stripes, the ratio of space consumption between replicated and RAID areas, the current workload level, etc., and their combinations.

4.3 Spike Detection and Request Redirection

Even with perfectly spread-out request loads and short write paths, applications suffer sudden surges in request latency when the underlying SSD devices undergo activities such as GC. FusionRAID sets its final line of defense against such adversity by performing constant SSD responsiveness monitoring and request diversion when latency spikes are detected.

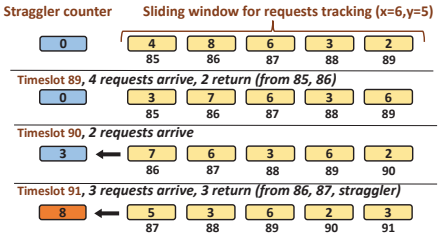


Figure 8: FusionRAID SSD spike detection. The number under each yellow box is the timeslot ID and the one within is the count of in-progress IOs issued in that timeslot.

Motivated by our spike behavior study (Section 2.3), we argue that without SSD internal information, though the timing of spikes is hard to predict, their duration and amplitude make reactive avoidance feasible. FusionRAID tracks the number of *stragglers* (with processing time larger than a preset threshold t) among requests issued to each individual SSD, and can quickly react when a performance anomaly happens.

Light-weight spike detection The challenge, however, lies in the efficient tracking of all active requests in a large SSD pool. After all, spikes are more likely to happen under intensive workloads. Space and time overhead of keeping track of thousands of requests at such peak times is not negligible.

To this end, we propose a lightweight spike detection scheme that records the request dispatching time in coarse granularity. It divides the time-line into equal-sized timeslots. For each SSD in the pool, FusionRAID maintains the number of pending requests dispatched within the latest y timeslots (forming a sliding window with a time length of t). A separate *straggler counter* records the number of requests whose pending time exceeds t. Each request issued increments the request counter for the current timeslot by 1, with an issuing timestamp tagged with the request. Similarly, a request's completion decrements the request counter of its corresponding timeslot. Upon the expiration of the current timeslot, the window slides, and the counter of the oldest timeslot has its value aggregated to the straggler counter. An SSD is identified as unresponsive (under spike) if the straggler counter reaches a preset threshold $\hat{x}$. For an unresponsive SSD, with new traffic guided away and spike-causing internal activities receding, eventually, its straggler counter will fall below $\check{x}$, putting it back to full service. For consumer SSDs, we set $\hat{x}$ equal to $\check{x}$ because requests return in batch once spikes end. However, spikes on DC SSDs often overlap with the previous ones so we set a gap between two thresholds, avoiding oscillations.

Figure 8 illustrates this per-SSD spike detection mechanism, with a sliding window sized at 5. At the last step (timeslot 91), the oldest timeslot (86) has its counter merged with the previous straggler counter, minus 2 requests returning (from "straggler" and timeslot 86). The result ($3+7-2=8$) exceeds the threshold $x=6$, identifying the SSD as unresponsive.

Selective request redirection When an I/O request arrives, FusionRAID judges whether the target SSD is unresponsive by reading its straggler count. When a write is directed onto an unresponsive SSD, FusionRAID searches along the spare block pair or stripe list till it finds one not involving any unresponsive SSD, and performs the update there. The search tends to be short as spikes are relatively rare events. Skipped block pairs or stripes are added to the list tail.

Such redirection also applies to reads. When data requested reside in the RAID area, a slow SSD can be skipped over by data reconstruction from all the remaining blocks in the stripe as in ToleRAID [23]. In the replicated area (which stores more active data and tends to attract more reads), FusionRAID reads from the faster of the blocks (with lower straggler count).

Since our spike detection is reactive, the damage is already done upon successful detection. It might be helpful to adopt existing strategies such as "hedged requests" [17], which resends victim requests to other SSDs when outstanding long enough. In addition, FusionRAID may even proactively trigger GC using the SSD trim mechanism [64] (which informs the SSD to recycle invalid blocks), when it "guesses" that a spike is imminent based on historical monitoring data.

4.4 Metadata Management

Now with major FusionRAID operations explained, we come back to discuss its storage organization, in particular, metadata maintenance necessary to enable partial stripe updates.

Fusion Block Mapping Table (FBMT) This central map-

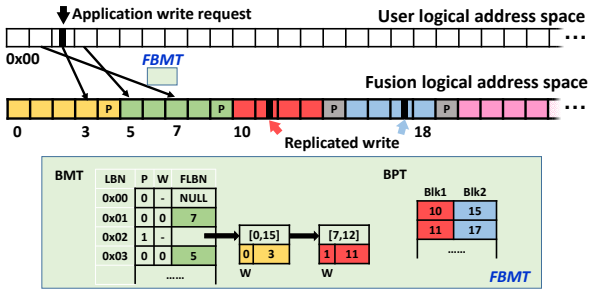


Figure 9: Block mapping and write handling

ping table in FusionRAID contains two components: a *Block Mapping Table (BMT)*, a direct address table storing the block mapping from the user logical block address to the Fusion block address, and a *Block Pairing Table (BPT)*, a hash table storing paired blocks used in replicated writes. Figure 9 gives sample illustrations. To reduce metadata storage, FusionRAID adopts a relatively large block size (64KB by default), which is also its RAID stripe unit size.

FusionRAID may write directly to the RAID area or make replicated writes. This distinction is recorded by the `W` bit in the 40-bit BMT entries: 0 for RAID and 1 for replicated. In the former case, the remaining 38 bits store the Fusion logical block number (FLBN), supporting a 16PB logical space with the default 64KB block size (*i.e.*, 4096 4TB-SSDs within one enclosure). In the latter, the BMT entry instead stores the smaller block number in a block pair, as a key to the BPT.

Recall the in-position replicated-to-RAID conversion shown in Figure 7. For the stripe to be kept (the left one), its three valid blocks will see their BMT entries updated during the conversion, with the `W` bit switched to 0 (from "replicated" to "RAID"), and its FLBN properly recorded. For the one to be discarded (the right one), the entire stripe is reclaimed and returned to the spare stripe list. Finally, the four involved block pairs are removed from the BPT.

Adopting larger blocks reduces metadata storage overhead, but creates more partial block updates. To avoid read-modify-write operations, which not only prolong the write process but incur write amplification, FusionRAID utilizes a patching method to append partial updates to the BMT entry of the updated block. The `P` bit here records the block's partial update history: if set to 1, the FLBN field becomes the head of a linked-list, whose nodes each contains an updated page range, along with the FLBN storing these updates (at their corresponding pages). Again, each element carries a `W` bit to tell whether the corresponding user logical block is in the replicated area.

Figure 9 demonstrates the handling of a write on user logical block address 0x02, which before the write was in the RAID area. This small write updates pages 7-12 of the block and incurs out-of-place updates via replicated writes. As described in Section 4.2, a pair of blocks (11 and 17, as indicated by the red and blue arrows) are allocated, with these pages written to the corresponding locations within these two blocks, without reading the unchanged data. Now, block 0x02 has its valid data distributed in three Fusion logical blocks: the previous location at block 3, plus the block pair 11 and 17. Thus its BMT entry carries a linked-list, with a node for page range 7-12 linking to the ordered block pair (11,17) in the BPT. Such a linked-list can be periodically compacted so that its length does not exceed the number of pages in a block.

Space overhead Out-of-place updates produce holes (invalid blocks). Holes in stripes cannot be reused directly since the invalid data segment is still involved in parity computation. Again the problem can be solved with periodic compaction similar to SSD GC mechanisms [34, 69], performed during our replicated-to-RAID conversion. In our experiments, we found such small partial rewrites quite infrequent across our evaluated traces (only 1.3%) leading to a small portion of BMT entries having such linked-lists.

FusionRAID maintains FBMT in battery-backed DRAM, for both performance and durability. FusionRAID's overall metadata storage overhead is quite modest. Each BMT entry is 5-bytes long, and our optimized BPT entry only takes 6 bytes (exploiting the proximity of logical addresses for paired blocks). Given the ratio of linked-list entries (each 10-bytes long) and a 10% space limit to the replicated area, these data structures altogether take 0.0084% of storage capacity. This means to manage a fully allocated 60TB SSD pool, only 5.2GB battery-backed memory is needed for FusionRAID to store its mapping data structures. In addition, FusionRAID needs to log updates in battery-backed memory, to ensure consistency of in-progress operations.

5 Performance Evaluation

5.1 Experiment Setup

Testbed We use a SuperMicro 4U storage server, with two 12-core Intel XEON E5-2650 V4 processors and 128GB DDR4 memory, running Ubuntu 16.04 with Linux kernel v4.15.0. It has two AOC-S3008L-L8I SAS JBOD adapters, each connected to a 30-bay SAS3 expander backplane via 2 channels.

For RAID evaluation, we select datacenter devices tested in our SSD performance study (§2): SSD E (Intel D3-S4510). We have 15 drives sitting on one backplane, with each test using one 30-drive SSD pool. The I/O channels provide a combined I/O bandwidth of 24GB/s, exceeding the aggregate sequential bandwidth from the 30 SSDs (195MB/s per SSD we measured, 5.71GB/s in total).

Workloads We use both trace-driven and real application tests. For the former, we implemented a trace player in C using *libaio* [7] that issues direct block I/O requests according to given timestamps. We use eight traces mentioned in Section 2.1 with major attributes in Table 4. For the latter, we evaluate FusionRAID with the popular RocksDB KV store [10] running YCSB workloads [16].

Trace	IOPS	Write ratio	Avg. write size
YCSB-Load	409	99%	507.0KB
YCSB-A	1353	64%	500.4KB
YCSB-B	1218	62%	500.2KB
TPC-C	5764	75%	29.6KB
TensorFlow	65	69%	80.1KB
VirtualDesktop	811	42%	23.8KB
Exchange	846	70%	13.1KB
Proxy	307	32%	13.8KB

Table 4: Characteristics of experimented I/O traces

RAID systems setup We implement FusionRAID as a Linux kernel module in v4.15.0 with about 5,400 LoC and evaluate it using 29-SSD pools,[1] with the stripe width of 7 (6+1 RAID-5). We compare with *LogRAID* [37], a log-structured RAID-50 that appends all updates. We used our own implementation[2] based on existing literature [13, 37], In addition, we implement *ToleRAID* [23], designed for cutting read tail latency, following its authors' guidance. We also evaluate two common organizations utilizing disk pools: 4 independent (6+1)-disk RAID-5 arrays (*4-RAID5*) and a RAID-50 system that stripes across them (*RAID50*). Finally, we implement another alternative design that adds an NVRAM write buffer above RAID50, which we label *NV-RAID*. All the above five systems use 28 SSDs in 4 RAID groups and 1 as a hot spare.

Unless otherwise noted, we test with SSDs consistently *aged*: first cleaned with the Linux `hdparm SECURE_ERASE` command, followed by a full-device sequential write, and finally, 6 hours random 16KB writes using `fio` [4], to guarantee each write generates invalid page(s).

5.2 Overall Performance

Trace-driven, concurrent workloads Considering the common usage of our testbed SSD pool size, we measure overall performance by co-running multiple workloads. More specifically, we select 20 *4-workload mixes* randomly from the aforementioned 8 traces, testing 4-RAID5, RAID50, LogRAID, ToleRAID, and FusionRAID on DC SSDs.

Figure 10 gives the median and tail latencies of 8 test workloads. The bars show the average value among all their executions in the 20 mixes (number of executions ranges between 9 and 13), and error bars mark the min/max values. Note that the *y* axis in the tail latency chart uses a log scale.

4-RAID5 shows comparatively consistent performance under light workloads (`TF` and `Proxy`) due to hardware isolation. However, the median and tail latencies on 4-RAID5 increase obviously under workloads with larger average write size, higher bandwidth or I/O bursts due to limited resources in one RAID-5. Despite spreading work to all 28 SSDs, RAID50 does not reduce median latency and often worsens tail latency. Light workloads show significant performance degradation on RAID50 when they share resources with heavy ones. RAID50's two-level striping adds further complexity and inter-SSD dependency into the I/O path, making average cases more costly. The worst cases, meanwhile, are slowed down by one or two unresponsive SSDs.

LogRAID does not appear to help: by consolidating all writes to the pool into a single log stream, it reduces concurrency and can utilize 1-2 7-SSD arrays at a time (while RAID50 and FusionRAID simultaneously use more disks). Moreover, data writes still experience the underlying RAID write path, thereby enduring higher latency. Compared with 4-RAID5, ToleRAID brings almost identical median latency under all the 8 workloads, and obviously reduces tail latency under four workloads (*i.e.* `YCSB-A`, `YCSB-B`, `VD`, `Proxy`). However, ToleRAID does not reduce tail latency under the other 4 workloads with higher write ratios, as write I/Os cannot benefit from ToleRAID's request redirection.

FusionRAID, on the other hand, significantly reduces *both* median and tail latencies. Compared with 4-RAID5, FusionRAID shows an average reduction of 49% in median latency across the 8 traces and a maximum of 87%. Compared with RAID50, LogRAID and ToleRAID, the average/maximum reductions are 81%/97%, 76%/98% and 45%/85%, respectively. FusionRAID's P99 improvement (Figure 10(b)) is even more impressive, averaging a 15× reduction (up to 32×) from 4-RAID5, 35× (up to 62×) from RAID50, 34× (up to 61×) from LogRAID, and 8.3× (up to 14×) from ToleRAID. Later we give an in-depth breakdown of sources contributing to such dramatic cut in tail latency.

In addition, for both median and tail, FusionRAID achieves shorter error bars than RAID50, LogRAID, and even 4-RAID5 (which offers hardware isolation, with a dedicated RAID-5 array for each workload). This demonstrates that spreading bursts, simplifying writes, and avoiding spikes bring more reliable performance than simply trying to protect workloads from each other. Finally, all tested RAID systems show the same throughput since trace-driven workloads issue I/O requests according to timestamps. This also allows us to observe the median and tail latency of FusionRAID and other systems under the same load intensity, for fair comparison.

Systems	Workloads	Update ratio	Read avg. latency (ms)		Update avg. latency (ms)	
			Slowest 10%	*Slowest 1%*	*Slowest 10%*	*Slowest 1%*
RAID50	YCSB-Load	100%			32.55	232.17
FusionRAID					4.22	29.25
RAID50	YCSB-A	50%	0.92	2.4	9.2	63.13
FusionRAID			0.753	1.924	2.35	11.80
RAID50	YCSB-B	5%	0.66	1.81	5.17	34.34
FusionRAID			0.47	1.55	1.27	5.86

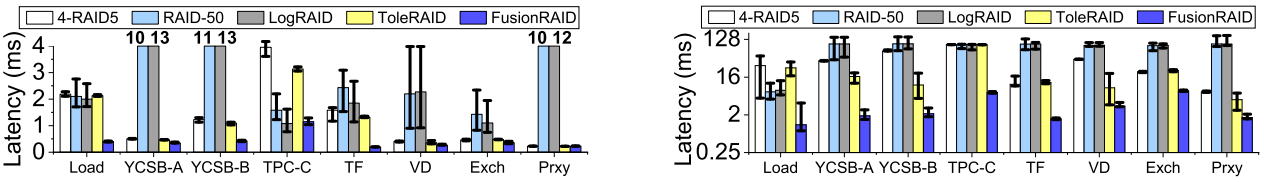
Table 5: RocksDB average and tail latency in running YCSB

Real-application workloads Next, we evaluate with representative YCSB workloads, of varied write intensity levels, running RocksDB on top of ext4 above FusionRAID and RAID50. Table 5 lists workload information and results.

For each workload, Table 5 lists the average latency of the slowest 10% and the slowest 1% of operations, for reads

[1] MOLS requires pool size to be a power of a prime number.

[2] Due to time/resource limit this system is implemented in user space, disabling it from supporting a file system and running applications.



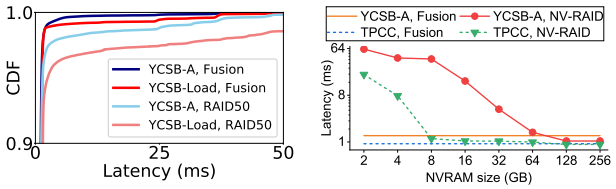
(a) Median latency. Bold numbers above bars denote average median latencies exceeding 4ms.

(b) P99 tail latency, *log scale*.

Figure 10: Overall performance comparison, from 20 randomly selected 4-workload mixes. Error bars show ranges of measured latency from all executions of each workload.

and writes separately. For the least write-intensive workload (YCSB-B), most reads are served from the RocksDB memtable, closing the performance gap between FusionRAID and RAID50. Still, across all three workloads, read tail latency consistently benefits from FusionRAID, due to its *more efficient digestion of request bursts.* For writes, even at an update rate as low as 5%, FusionRAID reduces the average latencies of the slowest 10% and 1% by 4.1× and 5.9×, respectively. As expected, its margin of improvement grows with write intensity, reaching 7.9× for YCSB-Load's slowest 1%.

As for writes, Figure 11(a) plots RocksDB update latency CDFs under the two write-intensive YCSB workloads. FusionRAID reduces the write tail latency significantly, especially with YCSB-Load. A major source of the long tail latency for update operations in RocksDB is the contention between foreground memtable flushing and background compaction [6]. FusionRAID benefits from its higher and more consistent bandwidth, reducing the probability of contention.



(a) RocksDB latency CDF

(b) P99 latency

Figure 11: Detailed latency comparison: (a) FusionRAID vs. RAID50 with YCSB-Load and YCSB-A, and (b) FusionRAID vs. NV-RAID with varying NVRAM size, *log scale*

Comparison with NV-RAID Intuitively, one can tame the tail latency of SSD arrays easily with an NVRAM buffer. To address this concern, we compare FusionRAID with NV-RAID. Since we cannot use the Intel Optane NVDIMM without a processor upgrade, we emulate an NVRAM buffer using DRAM and set a 100ns delay for each cacheline flush operation [32, 43]. To focus on the effect of such a buffer, we only add delays for data operations and not metadata ones.

Figure 11(b) illustrates the P99 latency of NV-RAID with different NVRAM sizes under YCSB-A and TPC-C, and that of FusionRAID (two straight lines) for reference. For YCSB-A, the P99 latency remains over 40ms, 30× higher than the FusionRAID result, with an NVRAM buffer size under 8GB. Here such small buffers do not sufficiently ease write pressure, incurring SSD GC activities that reduce effective SSD array bandwidth and delaying buffer flushes, which in turn results in further NVRAM buffer space shortage. As the buffer size increases, NV-RAID's tail latency improves, nearly catching up with FusionRAID at 64GB and leveling off over 128GB.

For TPC-C, as the NVRAM buffer grows, the P99 latency of NV-RAID decreases more quickly. The inter-workload difference here is due to TPC-C's smaller requests and higher issuing rates, while YCSB-A contains larger (512KB) I/Os. Therefore NV-RAID's tail latency gets fairly close to FusionRAID's, with an 8GB write buffer. This also leads to better median latency of NV-RAID than FusionRAID with TPC-C (28µs vs. 96µs) with the former using a 256GB buffer. With YCSB-A, on the other hand, even at this full NVRAM buffer capacity NV-RAID delivers a median latency of 391µs (vs. FusionRAID's 320µs).

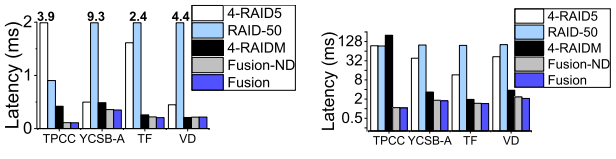
One sees that FusionRAID delivers similar or better tail latency performance as adopting an NVRAM buffer, while the median performance is more workload-dependent. Meanwhile, FusionRAID is designed to support multiple workloads simultaneously. Under such scenarios, concurrent applications need to share the precious NVRAM space, and may suffer write bandwidth scalability problems when more concurrent threads access the NVRAM, as found by a recent study [70]. Finally, NVRAM is far more expensive than SSDs: one 256GB NVDIMM costs $2,595 [2], equal to the price of our 30-SSD pool, even without considering the cost of the required server upgrade. Also, in the case of the Intel Optane NVDIMM, unlike most storage devices, a larger NVRAM device (256GB vs. 128GB) actually costs more per GB.

5.3 Impact of Individual Techniques

For the rest of the section, due to space limit, we focus on the mix including four different types of workloads (TPC-C, YCSB-A, TF and VD). Here, to isolate the improvement brought by key FusionRAID techniques, we evaluate its two intermediate versions: *4-RAIDM*, with 4 independent 7-disk volumes like 4-RAID5 but with all write requests conducted by performing mirrored log-structure writes in a round-robin way,

and *Fusion-ND*, same as FusionRAID but with SSD spike detection and request redirection removed.

Figure 12 shows the median and P99 latencies of different systems. From 4-RAID5 to RAID50, as explained earlier, building a single, shared volume on the entire SSD pool does not necessarily help. Replacing the costly RAID write with replicated write (4-RAIDM), in contrast, brings about the most significant improvement. Compared with 4-RAID5, 4-RAIDM reduces median latency by 2.1%-89% and shrinks P99 by 5.9-12×. However, the P99 latency of 4-RAIDM under TPC-C reaches 202ms, caused by the limited number of SSDs to serve intensive writes.



(a) Median latency (bold numbers giving values over 2)

(b) P99 latency, *log scale*

Figure 12: Incremental impact of proposed techniques

Fusion-ND enjoys the same benefits of simplified writes but spreading work to the entire pool rather than having 4 physically isolated arrays. Now with a shorter and decoupled write path, involving more disks improves processing power without propagating spikes. This produces a significant reduction from the most write-intensive workload (TPC-C) in tail latency (again plotted in log scale).

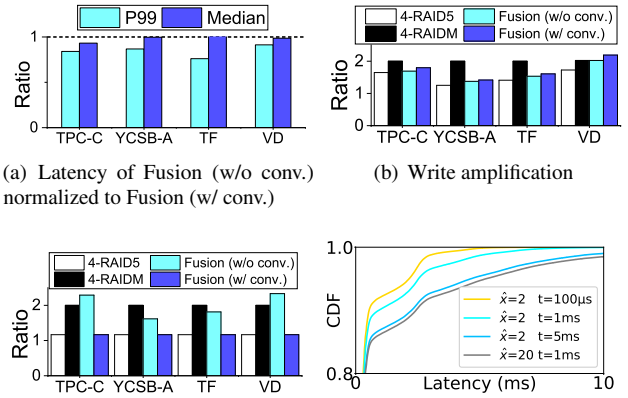
	Slowest 1% avg. latency (ms)		Slowest 0.1% avg. latency (ms)	
	Fusion-ND	*Fusion*	*Fusion-ND*	*Fusion*
TPC-C	1.90	1.47	6.34	2.79
YCSB-A	3.04	2.41	15.37	9.71
TF	2.31	2.20	52.16	49.83
VD	6.96	2.97	45.49	11.50

Table 6: Average latency at tails: Fusion-ND vs. Fusion

Finally, from Fusion-ND to FusionRAID, we add SSD spike detection and request redirection. Their difference may not seem significant from Figure 12, as it only comes into play when requests run into device-side spikes. As shown in Table 6, FusionRAID reduces the average latency of the slowest 1% requests by 1.1-2.3×, redirecting 0.73% of write requests from the Fusion-ND baseline. We also measure the frequency and duration of spikes on Fusion-ND with spike detection on and request redirection off. On average one SSD experiences 4.85 spikes/minute, each lasting for 3.79ms. Although these spikes appear quite short, bursty requests suffer performance degradation in batch once encountering them.

5.4 FusionRAID Overhead and Sensitivity

FusionRAID's major internal I/O activity is its replicated-to-RAID data conversion. Our tests presented earlier all have



(a) Latency of Fusion (w/o conv.) normalized to Fusion (w/ conv.)

(b) Write amplification

(c) Space overhead

(d) Latency CDF of Fusion with varying thresholds in spike detection

Figure 13: FusionRAID overhead and sensitivity

conversion turned on (only temporarily suspended when foreground throughput on a disk exceeds 40MB/s, our observed per-drive average). To examine its performance impact, Figure 13(a) shows performance with and without such background conversion. One sees that even with our current brute-force conversion policy (with no foreground-aware optimization such as avoiding application busy bursts), its performance overhead is quite small, thanks to our in-position stripe allocation during replicated writes.

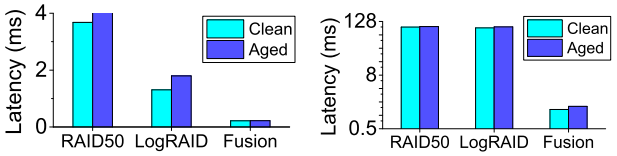
Next, Figure 13(b) reports the ratio of write amplification across all systems. Compared with 4-RAIDM, FusionRAID reduces the ratio across most of the workloads since it performs RAID writes directly for large requests. An exception is that FusionRAID brings a higher ratio under VD. This is because numerous replicated writes generated by dominant small writes in VD (see Table 4), plus parity updates in conversion, increase write amplification more than mirroring does. On average, its in-position conversion only brings around a 5.6% increase in write amplification.

Space consumption, meanwhile, depends on the aggressiveness of the background conversion policy adopted. Figure 13(c) gives the overall space consumption (vs. user data size), for all data written during the trace run. As expected, 4-RAID5's extra space overhead comes from the single parity block in its 6+1 stripe. 4-RAIDM has a constant space ratio of 2 as it performs simple mirroring. FusionRAID, with conversion turned off, has a slightly varying space ratio across workloads, due to their different write patterns. With conversion fully performed, one of the replicas is recycled (the other reclaimed) and multiple writes are compacted, returning the space consumption to the same as the 4-RAID5 level.

In addition, we examine the sensitivity of parameters in spike detection (§4.3). We first set y (determining the counting precision) at 10 and $\check{x}$ (used to speculate the end of latency spikes) at 0 empirically. Figure 13(d) shows latency CDFs under Exchange with different values of $\hat{x}$ and t. Smaller $\hat{x}$ and t help FusionRAID to detect spikes and react earlier while more

SSDs are identified as unresponsive, leaving fewer choices for incoming I/Os. Fortunately, FusionRAID performs well with smaller thresholds, thanks to the large 30-SSD pool.

Finally, we discuss how SSD aging affects the systems' performance. Figure 14 compares the performance of RAID50, LogRAID, and FusionRAID on clean and aged SSDs under the same workload combination as in §5.3, showing median and P99 latency on average. Aside from its capability to significantly reduce the median/tail latency in all cases, FusionRAID shows different behavior across clean and aged SSDs than the baseline systems. While RAID50 and LogRAID have higher median latency on aged SSDs due to increased GC activities, FusionRAID works as well there due to its spike aversion. For tail latency, however, FusionRAID does perform better on clean SSDs, where it has more alternatives to redirect requests. RAID50 and LogRAID, without such a mechanism, show no differences across aged and clean SSDs.



(a) Median latency (b) P99 latency, *log scale*

Figure 14: Performance on clean vs. aged SSDs

6 Related Work

SSD RAID designs SSD RAID systems have been extensively studied, with approaches roughly in three groups: 1) improving reliability by distributing parity unevenly across the array [5] or through wear leveling across member SSDs [61], 2) enhancing average-case performance and/or reliability by mitigating the parity update problem [14,15,28,36,42], and 3) taming tail-latency by alleviating GC impact [23,37,38,66,69]. Approaches in the first two groups do not address tail latency, and most in the third group [38, 66, 69] rely on host-managed/open-channel SSDs. One exception, ToleRAID [23], focuses on cutting read tails under a full-stripe read workload. In contrast, FusionRAID works on off-the-shelf SSDs and aims to reduce *both average-case and tail latencies*, while maintaining the same fault tolerance level.

A highly related system is SWAN [37], which eliminates GC impact on commodity SSD arrays (Rails [54] is similar but focuses on read protection from GC). It partitions drives into groups, which rotate in handling foreground and internal writes, shielding user writes from GC traffic. It targets settings where SSD RAIDs are built for capacity, but with aggregate bandwidth bound by network, while FusionRAID targets shared storage serving latency-critical applications. Compared with SWAN, FusionRAID is reactive rather than proactive, but protects and redirects both reads and writes.

RAID data layout optimization Parity declustering [46] utilizes as many disks as possible to serve application requests in data reconstruction. It has been extended and optimized by many [3, 25, 26, 48, 63]. It is also widely adopted in industry, by products such as PanFS [63], the IBM GPFS Native RAID [19] and Spectrum Scale RAID [27], HPE 3PAR [60], and Huawei's RAID2.0+ [44]. FusionRAID's design leverages existing Latin-square based deterministic addressing of RAID+ [71] but augments it with explicit block mapping to enable two-phase writes and out-of-place updates.

RAID write optimization Purity [15], Flash-Aware RAID [28], and PPC [14] use NVRAM to buffer the incoming data and/or parity information and delay parity updates, so as to conduct full-stripe writes and reduce the reads involved in parity updates [15], or to reduce the number of parity commits to SSDs [14, 28]. However, they require large amounts of NVRAM for storing data and/or parity information. ESAP-RAID [36] and RAID-Z [9] organize the incoming data in elastic-width stripes to reduce parity-induced read overhead, at the cost of increased stripe management complexity.

Two-phase writing was used by existing systems: AutoRAID [65], DiskReduce [21], and Log Disk Mirroring (LDM) [67] all write data to a replicated zone, with future background conversion to the RAID zone. However, FusionRAID is unique in its in-place conversion by replicating in a stripe-ready manner. Note that it specifically targets SSD RAIDs, where massive data migration may cause frequent GC and consume more SSD write cycles.

Alleviating GC Interference Harmonia [39] and Global Garbage Collection (GGC) [38] synchronize GC across a set of SSDs, thereby reducing overall performance variability. Application-managed Flash [41] and LightNVM [8] eliminate GC overhead by letting the host software manage the exposed flash channels. Several other systems cut read tail latency by issuing an extra read to parity block and rebuild the "late" data [23, 66, 69], and write one by enforcing at most one active GC in every RAID group and writing data to a no-GC member [66, 69]. Unlike the above, FusionRAID works without assuming SSD internal information/control, by observing SSDs' performance behavior to detect the onset of degradation and steer away if possible.

7 Conclusion

With FusionRAID, we argue that SSD RAID systems can be much faster *and* more consistent, by eagerly spreading application load, lazily performing parity writes (instead trading space temporarily for simple replicated writes), and carefully watching individual SSD's performance and waiting out their transient latency spikes. Large SSD enclosures, once not constrained with the rigid routines of traditional RAID arrays, simultaneously provide high concurrency to serve co-executing applications' bursty I/Os, and high flexibility in avoiding drives under transient performance degradation.

Acknowledgment

We thank all reviewers for their insightful comments and helpful suggestions. We are especially grateful to our shepherd, Keith Smith, for his thorough, detailed, and patient guidance during our camera-ready preparation. This work was supported by the National key R&D Program of China under Grant 2018YFB0203902, and the National Natural Science Foundation of China under Grants 61672315 and 62025203.

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, and Rina Panigrahy. Design tradeoffs for SSD performance. In *the 2008 USENIX Annual Technical Conference (USENIX'08)*, pages 57–70, 2008.

[2] Paul Alcorn. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2019.

[3] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of 25th International Symposium on Computer Architecture (ISCA'98)*, pages 109–120, 1998.

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

[5] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *ACM Transactions on Storage (TOS)*, 6(2):4, 2010.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 753–766, 2019.

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

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association.

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

[10] Dhruba Borthakur. RocksDB: A persistent key-value store. https://rocksdb.org/, 2014.

[11] Eric Burgener. Justifying investment in all-flash arrays. https://www.emc.com/collateral/analyst-reports/justifying-investments-in-all-flash-arrays.pdf, 2019.

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

[13] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software orchestrated flash array. In *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*, pages 1–11. ACM, 2014.

[14] 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 Transactions on Very Large Scale Integration (VLSI'14) Systems*, 22(7):1470–1480, 2014.

[15] 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 (SIGMOD'15)*, pages 1683–1694. ACM, 2015.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154. ACM, 2010.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[19] Veera Deenadhayalan. GPFS Native RAID for 100,000-Disk Petascale Systems. In *25th Large Installation System Administration Conference (LISA'11)*, 2011.

[20] DELL EMC. VMAX All Flash Family, https://www.dellemc.com/en-us/collaterals/unauth/data-sheets/products/storage-2/h16051-vmax-all-flash-250f-950f-ss.pdf. 2020.

[21] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. DiskReduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *SC'11*, 2011.

[22] FUJITSU. FUJITSU Storage ETERNUS AF650 S3. https://www.fujitsu.com/global/products/computing/storage/all-flash-arrays/eternus-af650-s3/index.html, 2020.

[23] Mingzhe Hao, Gokul Soundararajan, Deepak 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'16)*, pages 263–276, Santa Clara, CA, February 2016. USENIX Association.

[24] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM SIGPLAN Notices*, volume 50, pages 161–175. ACM, 2015.

[25] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 23–35, 1992.

[26] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS'93)*, pages 422–431, 1993.

[27] IBM. IBM Spectrum Scale RAID. https://www.ibm.com/support/knowledgecenter/en/SSYSP8_5.3.1/raid_adm.pdf, 2017.

[28] Soojun Im and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers*, 60:80–92, 01 2011.

[29] Intel. Intel Solid State Drives. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives.html. 2020.

[30] I/O Umass Trace Repository. OLTP Application I/O and Search Engine I/O. http://traces.cs.umass.edu/index.php/Storage/Storage.

[31] Itnews. ANZ Bank goes all-flash for storage. https://www.itnews.com.au/news/anz-bank-goes-all-flash-for-storage-490262. 2020.

[32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[33] Dawoon Jung, Jeong Uk Kang, Heeseung Jo, Jin Soo Kim, and Joonwon Lee. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, 2010.

[34] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, 2014.

[35] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards SLO complying SSDs through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 183–189, 2015.

[36] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H Noh. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pages 1–12. IEEE, 2013.

[37] 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'19)*, pages 799–812, Renton, WA, July 2019. USENIX Association.

[38] Youngjae Kim, Junghee Lee, Sarp Oral, David A Dillow, Feiyi Wang, and Galen M Shipman. Coordinating garbage collection for arrays of solid-state drives. *IEEE Transactions on Computers*, 63(4):888–901, 2014.

[39] Youngjae Kim, Sarp Oral, Galen M Shipman, Junghee Lee, David A Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*, pages 1–12. IEEE, 2011.

[40] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of*

the 10th ACM International Systems and Storage Conference, pages 1–11, 2017.

[41] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 339–353, Santa Clara, CA, February 2016. USENIX Association.

[42] Yongkun Li, Helen HW Chan, Patrick PC Lee, and Yinlong Xu. Elastic parity logging for SSD RAID arrays. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, pages 49–60. IEEE, 2016.

[43] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

[44] Huawei Technologies Co., Ltd. RAID 2.0+ Technical White Paper. https://actfornet.com/HUAWEI_STORAGE_DOCS/Storage_All2/Enterprise%20Unified%20Storage%20RAID%202.0+%20Technology-HUAWEI%20OceanStor%20Technical%20White%20Paper.pdf, 2014.

[45] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[46] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 162–173, 1990.

[47] Sampann N. Amazon found every 100ms of latency cost them 1% in sales. https://www.linkedin.com/pulse/amazon-found-every-100ms-latency-cost-them-1-sales-sampann/, 2016.

[48] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)*, pages 1–10. IEEE Computer Society, 2004.

[49] NetApp. AFF A-Series All Flash Arrays. https://www.netapp.com/us/products/storage-systems/all-flash-array/aff-a-series.aspx#technical-specifications. 2020.

[50] 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 International Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[51] PureStorage. FlashArray//X. https://www.purestorage.com/products/nvme/flasharray-x.html. 2020.

[52] Sandisk. SanDisk InfiniFlash System. https://www.solidstateworks.com/InfiniFlash.asp. 2020.

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

[54] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent flash performance through redundancy. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 463–474, Philadelphia, PA, June 2014. USENIX Association.

[55] SNIA. Microsoft Enterprise Traces. http://iotta.snia.org/traces/130, 2007.

[56] SNIA. Microsoft Production Server Traces. http://iotta.snia.org/traces/158, 2007.

[57] SNIA. MSR Cambridge Traces. http://iotta.snia.org/traces/388, 2007.

[58] SNIA. SNIA Block I/O Traces. http://iotta.snia.org/tracetypes/3, 2017.

[59] P Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 513–527. USENIX Association, 2015.

[60] Karl L. Swartz. 3PAR Fast RAID: High performance without compromise. http://www.kls2.com/~karl/papers/raid-wp-10.0.pdf, 2010.

[61] Wei Wang, Tao Xie, and Abhinav Sharma. SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays. *ACM Transactions on Storage (TOS)*, 12(3):10, 2016.

[62] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, pages 1–12. IEEE, 2006.

[63] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *6th Usenix Conference on File and Storage Technologies (FAST'08)*, pages 17–33, 2008.

[64] Wikipedia. Trim (computing). https://en.wikipedia.org/wiki/Trim_(computing). 2020.

[65] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[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*, 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 Transactions on Storage (TOS)*, 12(4):22, 2016.

[68] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 543–557, 2015.

[69] 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. *ACM Transactions on Storage (TOS)*, 13(3):22, 2017.

[70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 169–182, 2020.

[71] 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'18)*, pages 279–294, Oakland, CA, 2018. USENIX Association.