# HybRAID: A High-Performance Hybrid RAID Storage Architecture for Write-Intensive Applications in All-Flash Storage Systems

Maryam Karimi, Reza Salkhordeh, André Brinkmann, and Hossein Asadi

**Abstract**—With the ever-increasing demand for higher I/O performance and reliability in data-intensive applications, *solid-state drives* (SSDs) typically configured as *redundant array of independent disks* (RAID) are broadly used in enterprise *all-flash storage systems*. While a mirrored RAID offers higher performance in random access workloads, parity-based RAIDs (e.g., RAID5) provide higher performance in sequential accesses with less cost overhead. Previous studies try to address the poor performance of parity-based RAIDs in small writes (i.e., writes into a single disk) by offering various schemes, including caching or logging small writes. However, such techniques impose a significant performance and/or reliability overheads and are seldom used in the industry. In addition, our empirical analysis shows that partial stripe writes, i.e., writing into a fraction of a full array in parity-based RAIDs, can significantly degrade the I/O performance, which has *not* been addressed in the previous work. In this paper, we first offer an empirical study which reveals partial stripe writes reduce the performance of parity-based RAIDs by up to 6.85× compared to full stripe writes (i.e., writes into entire disks). Then, we propose a high-performance hybrid RAID storage architecture, called *HybRAID*, which is optimized for write-intensive applications. HybRAID exploits the advantages of mirror- and parity-based RAIDs to improve the write performance. HybRAID directs a) aligned full stripe writes to parity-based RAID tier and b) small/partial stripe writes to the RAID1 tier. We propose an online migration scheme, which aims to move small/partial writes from parity-based RAID to RAID1, based on access frequency of updates. As a complement, we further offer offline migration, whose aim is to make room in the fast tier for future references. Experimental results over enterprise SSDs show that HybRAID improves the performance of write-intensive applications by 3.3× and 2.6×, as well as enhancing performance per cost by 3.1× and 3.0× compared to parity-based RAID and RAID10, respectively, at equivalent costs.

**Index Terms**—Solid-State Drives, Performance, Redundant Array of Independent Disks, All-flash Storage Systems.

## 1 Introduction

ALL-FLASH *Storage Systems* (AFS) using *Solid State Drives* (SSD) have been widely employed for enterprise applications, which compared to traditionally used *Hard Disk Drives* (HDD), offer higher read/write performance, particularly for random access workloads [19], [24], [20]. In AFS, multiple SSDs are configured in *Redundant Array of Independent Disks* (RAID) [6] to meet the higher performance, reliability, and capacity requirements of emerging data-intensive applications. A RAID array distributes I/O requests between multiple disks to achieve higher performance and protects original data from disk failures by either mirroring such as RAID1 or parity-based schemes such as RAID4, 5, or 6 [6], [24], [4], [5].

While parity-based RAID configurations are widely employed because of their low cost and high sequential performance, RAID1 provides superior random write performance than parity-based RAIDs. However, the high cost (2× disk overhead) of RAID1 (or RAID10[1]) prevents it from being used in AFS. For example, the cost comparison between RAID5 and RAID10, assuming equal usable capacities indicates that RAID10 is approximately 33.3% to 80% more expensive than RAID5 when the number of usable disks ranges from two to nine. Additionally, RAID5 has a well-known shortcoming called the "small write problem", which hinders its optimal usage in AFS. This shortcoming occurs due to write requests (a.k.a, small writes) that are equal to or smaller than the stripe unit size (data striping unit, a.k.a., chunk, shown in Fig. 1),

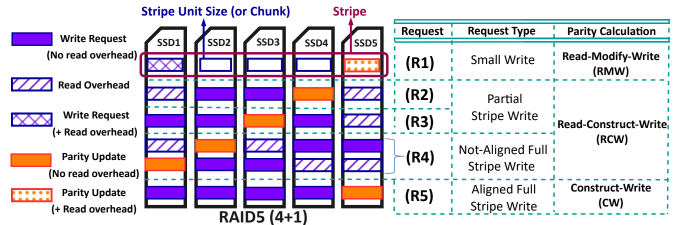



Fig. 1: Read overheads for parity calculation in RAID5 for different sizes of write requests with different parity update schemes.

which utilize the *read-modify-write* parity update scheme (e.g., R1 in Fig. 1) [7], [33], [37]. This parity update scheme necessitates reading both old data and old parity before producing new parity per stripe (i.e., one row of corresponding stripe units, shown in Fig. 1), resulting in a significant performance degradation [8], [33], [9]. In contrast, an aligned full stripe write only writes data in the entire stripe (e.g., R5 in Fig. 1) and does not require any pre-reads in RAID5. It only constructs parity and writes it, resulting in higher performance.

To reduce the number of pre-reads in small writes generated by the read-modify-write parity production scheme in RAID5, studies in [29], [18] suggested using an HDD that sequentially logs the parity between small writes and their old values. Other approaches involve caching either data [21], [40], [16], [22], [36], [30] or parities [11], [10] of small write requests on an SSD [16], [11], [10], main memory [30], [22], or multiple HDDs [21], [40], [36]. These approaches write to the main array after the cache is filled; this way, if small write requests are temporally localized, the number of reads before writes will be reduced. An alternate solution is to shift parity updates to the idle time of the storage

*Maryam Karimi and Hossein Asadi are with the Department of Computer Engineering, Sharif University of Technology, Emails: maryam.karimi@sharif.edu, asadi@sharif.edu. Reza salkhordeh and André Brinkmann are with the Department of Computer Science, Johannes Gutenberg University Mainz, Emails: rsalkhor@uni-mainz.de, brinkman@uni-mainz.de. The corresponding author: Hossein Asadi (Email:asadi@sharif.edu)*

1. RAID10 performs data striping across multiple RAID1 arrays, providing both data redundancy and improved performance.

system, which aims to coalesce small write parity updates [17].

Although previous studies have improved parity-based RAID performance by reducing the number of pre-reads required for small writes, they either *still* require pre-reads or lengthen the recovery process, which can negatively affect reliability. Furthermore, these studies have *only* focused on the performance overhead induced by small writes leveraging read-modify-write parity update scheme, while neglecting the performance overhead of partial stripe writes, i.e., writes into multiple disks, not the entire stripe (e.g., R2, R3, R4 in Fig. 1). Alternatively, *read-construct-write* parity update scheme is utilized for partial stripe writes, which involves reading stripe units that have not been written to while simultaneously writing updated data to regenerate parity [28], [33], [7].

To the best of our knowledge, *none* of the previous studies have targeted the performance overhead of read-construct-write parity update schemes used in partial stripe writes. However, our investigation shows that in SSD-based RAID5 arrays, the performance of partial stripe writes is considerably lower than that of full stripe writes (up to 6.85× slower), even for large-sized requests. In addition, partial stripe writes constitute a significant percentage of write-intensive workloads (on average 30% and up to 50%). Moreover, previous studies have either failed to eliminate pre-reads of small write requests (with a read-modify-write parity update scheme), or suffer from high overheads.

This paper proposes a high-performance hybrid RAID architecture, called ***HybRAID***, which optimizes I/O performance of write-intensive applications in AFS. Our results show that RAID5 write performance is considerably low in partial stripe writes (just as low as small writes). On the other hand, RAID1 has a lower write performance than RAID5 in full stripe writes. Hence, neither RAID5 nor RAID1 can *singly* provide optimal write performance for various request sizes. Based on these observations, we propose a hybrid RAID storage architecture that provides a tiered storage array between RAID5 and RAID1. In our proposed architecture, write requests for *aligned* full stripes (e.g., R5 in Fig. 1 which starts at the beginning of the stripe and fills the whole stripe) are sent to RAID5, while small writes (e.g., R1 in Fig. 1) and partial stripe writes (e.g., R2, R3, R4 in Fig. 1) are sent to RAID1 based on the access frequency of updates.

To mitigate the processing-induced performance overhead, we further propose a novel multi-thread scheme to simultaneously issue stripes of write requests to the underlying RAID levels. HybRAID performs the allocation and migration process online and requires migration *only* from the RAID5 tier to RAID1, resulting in low overhead for online migrations. We further offer an offline migration process, running in the background, which is initiated to make a room when a tier is full. We implement the mapping table for connecting logical addresses to physical addresses using both SSD and main memory to create a persistent and high-speed mapping structure. Write requests that modify the mapping between physical and logical addresses are sent to both SSD and main memory, while other requests are directed to main memory only. Due to the small size of the table entries we utilize a portion of the RAID1 tier to store the mapping table, which provides higher performance for small write requests. Moreover, the low migration overhead of HybRAID (≈6.3%) ensures that the performance of the architecture remains largely unaffected.

We implement HybRAID at the user level on a Gen9 HP server running Linux 5.15.0 kernel with Ubuntu Server 18.04.5 LTS operating system. We test HybRAID on more than thirty workloads from *SNIA MSR Cambridge Traces* [25]. The experimental results show that HybRAID increases the performance of write-intensive applications by an average of 3.3× and 2.6× compared to RAID5 and RAID10 with equal cost. Additionally, HybRAID offers an average of 12.3% reduction in read overhead compared to RAID5 while imposing 14.4% read overhead compared to RAID10 with equal raw capacity (cost). We also propose using heterogeneous SSDs per tier to provide the required endurance at an efficient cost. Accordingly, the cost-overhead is at most 6.4% compared to RAID5, and the cost-benefit is at least 28.9% compared to RAID10 with equal usable capacity. Also, by using heterogeneous SSDs, HybRAID results in an average of 3.1× and 3.0× performance per cost improvement over RAID5 and RAID10, respectively.

To our knowledge, the **main contributions** of this work are as follows:

- For the first time, we reveal that partial stripe writes with read-construct-write parity update scheme also exist in large requests and constitute considerable portion of write-intensive real-world workloads that can significantly degrade performance in parity-based RAIDs. We quantify this overhead and compare mirrored- and parity-based RAIDs in various types of write requests.
- To the best of our knowledge, this work is the first to consider the alignment of large requests in parity-based RAID systems, taking into account the significant performance difference between *aligned* (e.g., R5 in Fig. 1) full stripe writes and *not-aligned* full stripe writes (e.g., R4 in Fig. 1, which do not fill the entire stripe and are considered as multiple partial stripe writes). We propose HybRAID, which combines RAID1 and RAID5 to create a high-performance RAID configuration suitable for all types of write access patterns.
- We propose a low overhead migration policy to move the data between two RAID tiers when the data stripes are accessed with different patterns. Our migration policy imposes only 6.3% performance overhead.
- We implement HybRAID on a real platform using enterprise SSDs and utilize Linux available RAID software module (*mdadm*) to create RAID tiers. HybRAID is designed in such a way that it can seamlessly integrate into the *mdadm* kernel module. The results from testing HybRAID with over thirty real workloads indicate an average performance increase of 3.3× and 2.6×, as well as an average performance per cost improvement of 3.1× and 3.0× compared to RAID5 and RAID10 with equal usable capacity, respectively.

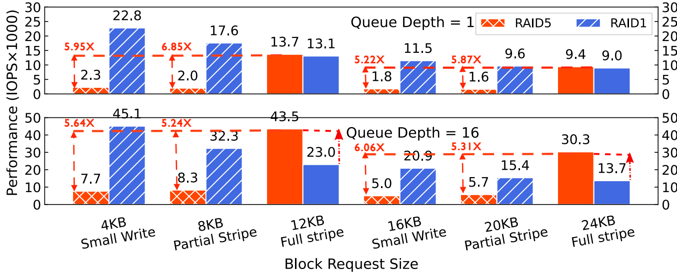
The rest of the paper is organized as follows: Section 2 illustrates the motivations behind HybRAID. Section 3 explains the proposed architecture, including the hardware architecture and algorithm. Section 4 demonstrates the experimental setup and presents the experimental results. Section 5 discusses previous studies. Finally, Section 6 concludes the paper.

## 2 Motivation

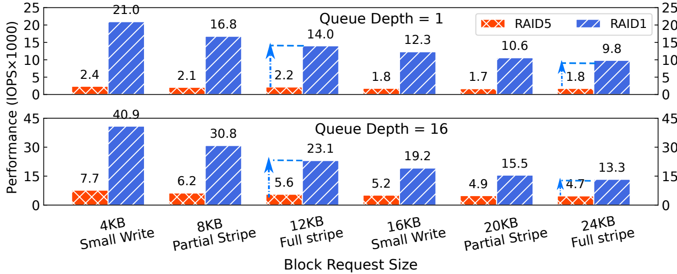
Here we begin by comparing the write performance of RAID5 and RAID1 in terms of write request size for aligned and non-aligned addresses based on the stripe size. Then we provide real-world workload characterization demonstrating the percentage of small, partial, and full stripe writes in write-intensive workloads.

### 2.1 Performance characterization of RAID5 and RAID1 in write-only workloads: request size

Here, we demonstrate the challenges of using RAID5 in terms of write performance, which have *not* been addressed in the previous




(a) With aligned addresses based on the stripe size




(b) With not-aligned addresses based on the stripe size

Fig. 2: Performance of RAID5 and RAID1 in IOPS (I/O per Second) unit with random access write-only workload at QD=1,16 for small, partial and full stripe request sizes where addresses are a) aligned, and b) not aligned based on the stripe size (test configurations are in Table 1).

work. We then provide the main motivation of HybRAID by comparing the write performance between RAID1 and RAID5 with different request sizes and address alignments. Table 1 reports the experimental setup used in our motivational tests. The employed workloads are synthetic random write, generated by *fio* module. Fig. 2 depicts write performance results for different request sizes. In Fig. 2a, the addresses are aligned based on the stripe size of the RAID5 array (in this case, 12KB). Hence, 4KB, 8KB, and 12KB request sizes are declared as small, partial stripe, and full stripe writes, respectively. Also, a 16KB request is composed of one full stripe write and one small write, a 20KB request comprises one full stripe and one partial stripe write, and a 24KB request includes two full stripe writes. Similar results are shown in Fig. 2b when addresses are not aligned based on the stripe size.

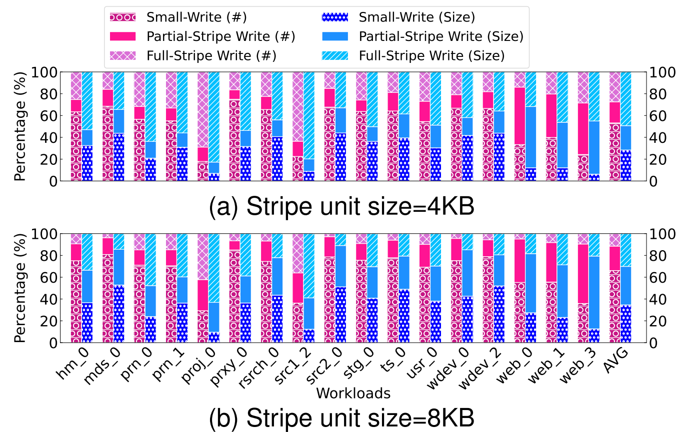
TABLE 1: RAID and workload specification for motivational tests.

| **RAID Specification** | |
|---|---|
| Stripe unit size | 4KB |
| Type of RAID | Software RAID (*mdadm* module) |
| #data disks + #redundant disks in RAID5 | 3+1 |
| #data disks + #redundant disks in RAID1 | 1+1 |
| Scheduler | *noop* |
| **Workload Specification** | |
| Address access pattern | Random write |
| Workload generator | *fio* module (synthetic) |
| Queue depth (numjobs-iodepth) | 1-1 , 16-16 |
| workload size | 10GB each job |
| **Disks Specification** | |
| SSD model | 256GB *Samsung 860 PRO* |
| **Operating System Specification** | |
| OS | Ubuntu Server 18.04.5 LTS |
| Kernel Version | 5.15.0 |

The performance results in Fig. 2a show that RAID5 acts poorly in *both* small and *partial* stripe writes. Small write performance is on average 5.7× *lower* than full stripe write performance with various *Queue Depths (QD)*[2] (QD=1, 16). In addition, partial stripe write performance is on average 5.8× *lower* than full stripe writes. This is because both small writes and partial stripe writes require *pre-reads* (read-modify-write in small writes and read-construct-write in partial stripe writes) to compute the updated parity. Hence, RAID5 write performance *not only* drops at small writes *but also* at partial stripe writes to an almost equal extent, which previous studies fail to address or mitigate such performance degradation.

Fig. 2a shows that when requests have no concurrency (QD=1), RAID1 performance for small and partial stripe writes is on average 7.7× *higher* than RAID5. This is due to that RAID1 does not require pre-reads and also needs fewer writes to disks. On the contrary, in full stripe writes, RAID5 performance is slightly (4%) higher than RAID1 for QD of 1. Similarly, the results for QD of 16 demonstrate that small and partial stripe write performance in RAID1 is on average 4.2× *still higher* than RAID5. Therefore, more disks in RAID5 (compared to RAID1) and greater queue depth (QD=16) *cannot* eliminate performance reduction caused by small and partial stripe writes in RAID5. Also, in QD of 16, the ratio between full stripe write performance of RAID5 to RAID1 becomes more remarkable than that of QD of 1; in particular, for 12KB and 24KB full stripe write requests, RAID5 to RAID1 performance ratio reaches 1.9× and 2.2×, respectively (Compare it to the mere 4% increase observed at a QD of 1). This is because full stripe writes do not require any read operations and increasing the queue depth leads to more simultaneous disk accesses, resulting in greater performance improvement. The characteristic of RAID10 compared to RAID5 is similar to that of RAID1 compared to RAID5, in such a way that for small and partial stripe writes, RAID10 outperforms RAID5, while for full stripe writes, RAID5 outperforms RAID10. Overall, we conclude that RAID5 performance is higher than RAID1/RAID10 at full stripe writes, while RAID1/RAID10 performance for small and partial stripe writes is greater than RAID5 even at high queue depths or when using less disks in RAID1/RAID10 compared to RAID5.

2. Queue depth is the number of I/O requests that can be issued in parallel to the underlying storage subsystem, which can help increase the number of I/O operations per second.




(b) Stripe unit size=8KB

Fig. 3: Write-intensive *SNIA* [25] workload characterization based on request size in RAID5 (3+1) with a) 4KB, b) 8KB stripe unit size.

Our empirical study here also shows that *address alignment* of write requests plays an important role in the performance characteristics of RAID5. Fig. 2b depicts similar results with a difference that the addresses are *not* aligned based on stripe size. In this case, RAID5 performance *only* differs greatly in full stripe writes compared to the results shown in Fig. 2a, where addresses are aligned. As shown in Fig. 2b, the full stripe write performance of RAID5 reaches the same low performance of small and partial stripe writes in RAID5. This is because when addresses are not aligned by the stripe size, full stripe write requests are likely divided into few small and partial stripe writes and are not entirely fitted in one (or multiple) stripes. Therefore, their performance is *as low as* the small and partial stripe requests. As a result, considering the address alignment of write requests based on stripe size is vital in attaining the *peak* performance of RAID5.

### 2.2 Workload characterization: write request size and number of write requests

In this section, we describe write-intensive workloads in terms of write request size grouped into small, partial, and full stripe writes. As illustrated in Fig. 1, the starting address of a request and the request size will classify each request into either small, partial, or full stripe. Additionally, each request can consist of multiple stripes. For instance, in Fig. 1, R4 includes two partial stripe writes. In our characterization study, we treat this request type as two distinct partial stripe writes. Fig. 3 addresses the *SNIA* [25] write-intensive workloads that have on average 79.5% write requests. This figure shows that small writes are dominant in number, while full and partial stripe requests are dominant in size. Hence, to architect a high-performance storage array, one should consider improving the performance for all types of the requests including small, partial, and full stripe; however, no single existing RAID configuration can provide high performance for all types (this is shown in Fig. 2a). To our knowledge, previous works *only* consider improving performance drop caused by small writes (with read-modify-write parity production), but they have not addressed the partial stripe writes (with read-construct-write parity production). This is while our characterization shows that the percentage of partial stripe writes in write-intensive workloads is, on average 29% and 35% for stripe unit size of 4KB (Fig. 3a) and 8KB (Fig. 3b), respectively. In some workloads such as `web_0`, `web_1` and `web_3`, the percentage of partial stripe writes is close to 50%. Such partial stripe write requests significantly decrease the performance of parity-based RAID (such as RAID5), as confirmed by the results shown in Fig. 2a.

## 3 Proposed Architecture

The motivational results presented earlier indicate that small and partial stripe writes constitute a significant proportion of write-intensive real-world workloads. This can adversely affect the performance of parity-based RAID storage systems, particularly RAID5, which demonstrates sub-optimal performance when handling small and partial stripe writes. Full stripe write requests are found to be optimal in terms of performance in RAID5 when the addresses are *aligned* relative to the stripe size. However, small and partial stripe write requests show higher performance in RAID1 than in RAID5. Our proposed architecture, called HybRAID, uses a tiering method that includes two tiers of SSD-based RAID1 and RAID5. In the following, we will cover: 1) how to assign requests to different tiers in HybRAID, 2) the types of data migrations required in HybRAID, 3) the proposed HybRAID architecture and its implementation details, and finally 4) the proposed algorithm used in HybRAID.

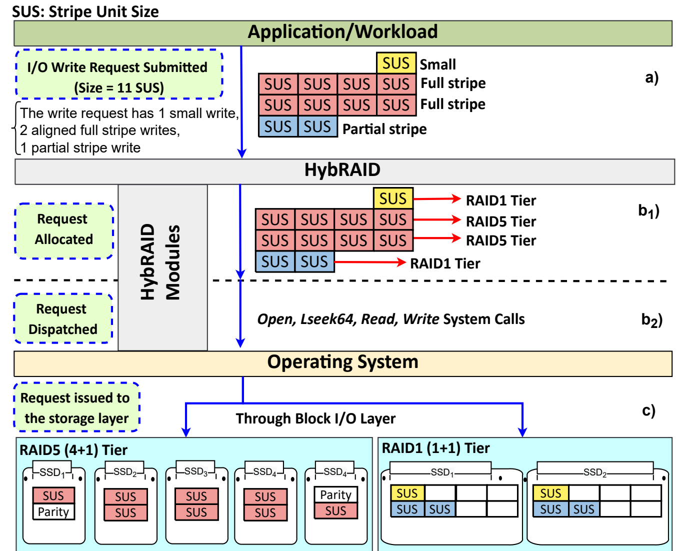



Fig. 4: Example of data allocations in two tiers of HybRAID (`SUS: Stripe Unit Size`)

### 3.1 Data Allocation

In the proposed architecture, we use two tiers: RAID1 and RAID5. Requests are allocated to each tier based on their size. Fig. 4 illustrates the data allocation in each tier. In this figure, RAID5 has five disks, while RAID1 contains two disks. For example, one issued write request includes 11 *Stripe Unit Sizes* (SUS), which are arranged and issued from the application, as shown in Fig. 4a. Then, in Fig. $4b_1$, by HybRAID modules, the requests are divided into two full stripes, one small stripe, and one partial stripe size. Because RAID5 has higher performance for full stripe requests, full stripe write requests should be placed in RAID5. On the other hand, write requests that are less than full stripe sizes (i.e., small and partial stripe writes) should be placed in RAID1 to achieve higher performance. When the allocations of all stripes are completed, HybRAID dispatches all stripes simultaneously to the underlying storage space (Fig. $4b_2$) by calling system call functions such as *open*, *lseek*, and *write* to the corresponding tiers. Afterwards, the block I/O layer in the operating system takes control of dispatching requests to the storage layer (Fig. 4c). As shown in Fig. 4c, two aligned full stripe writes are allocated in the RAID5 tier of HybRAID, while small and partial stripe writes are allocated in RAID1. Only *aligned* (based on stripe size) full stripe size requests should be placed in RAID5 (according to the results in Fig. 2). Hence, addressing, migration, and allocation unit size in HybRAID are determined based on the array stripe size. This approach helps to simplify data placement and reduce overall overhead. Additionally, the allocation decision in HybRAID is not solely dependent on the write request size but also on the update frequency, which will be discussed in detail in Section 3.4.

There are two different scenarios that illustrate possible data migrations. The first scenario involves the system response when an access is granted to update a single stripe unit size that was previously located in RAID5 (such as the red full stripe SUSes shown in Fig. 4a). Given that the current write request is a small write, it is optimal to write it in RAID1. However, since this address is already assigned in RAID5, the question is whether it is beneficial to migrate it to RAID1. The second scenario pertains to situations where full stripe update access is requested for an address that is already located in RAID1 (such as the blue partial stripe SUSes shown in Fig. 4a). In such cases, writing the full stripe in RAID5 is optimal. Here it is not immediately clear

whether it is beneficial to migrate to RAID5. These two questions will be answered in the next subsection.

### 3.2 Data Migration

This section elaborates the data migrations that would help the overall performance. Fig. 5 shows the situations in which the data tier may need to be changed. Fig. 5a shows the state where the update request is less than the full stripe size, while it was previously accessed and allocated in the RAID5 tier (as indicated by the red full stripe SUSes shown in Fig. 4). As the request size is less than a full stripe, RAID1 offers higher performance. Therefore, the requested data is assigned without data movement by placing it in the corresponding address in the RAID1 tier. However, because addressing is based on the stripe size, it is necessary to migrate the rest of the stripe from RAID5 to the RAID1 tier. This migration is performed *online*, in which the requested data is directly written to RAID1 while other chunks within the stripe are simultaneously migrated from RAID5 to RAID1. The request acknowledgment is issued after all chunks are migrated successfully and the corresponding stripes in RAID5 are marked as freed for subsequent requests. In Fig. 5b, the requested update data is a full stripe write that was previously written into RAID1 (as indicated by the blue partial stripe SUSes shown in Fig. 4). HybRAID optimizes performance by placing aligned full stripe size requests in the RAID5 tier. Since all the chunks within the stripe need to be updated, there are no remaining chunks that need to be moved from RAID1 to RAID5. Therefore, if the decision is to migrate data from RAID1 to RAID5, there is no need to transfer data between the two tiers. Instead, only the corresponding stripes in RAID1 are marked as freed for subsequent requests.

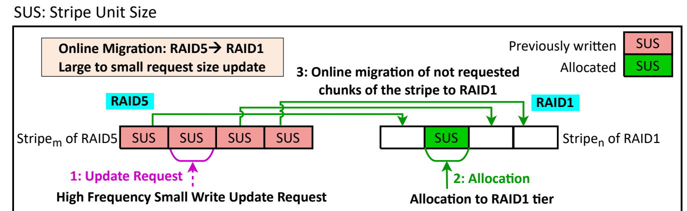
Fig. 5c shows a situation where all physical addresses in the RAID1 tier are filled. Therefore, it is necessary to free up space for subsequent requests. In this case, since the RAID1 tier is usually smaller in space than the RAID5 tier, it is necessary to migrate data blocks from RAID1 to the RAID5 tier *offline* based on the adopted policies. Offline migration allows for more critical data to be placed in the RAID1 tier and can be performed in the background, without affecting servicing of original requests.

### 3.3 Architecture

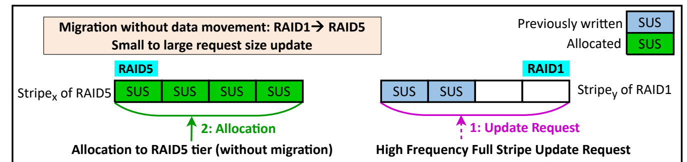
Here, we illustrate the architecture, modules, and data structures used to implement HybRAID. HybRAID consists of three parts (Fig. 6): the user, the operating system (kernel), and the data storage space. We implement HybRAID in the user space, which sends requests to the block I/O layer after performing a pre-processing operation. The requests are then issued to the data storage layer, which composed of two tiers of RAID1 and RAID5. Each part includes specific arrangements for implementing the proposed architecture, which are detailed next.

There are four basic units in the user space section, including the following: 1) I/O parser unit, 2) tier selection decision-making unit, 3) dispatcher (thread generator) unit, and 4) tier reclamation unit (as shown in the top part of Fig. 6). In the I/O parser unit, requests are read from a workload trace. As explored in the motivational results in Section 2.1, 3.1, the addressing of HybRAID is based on the stripe size to achieve the highest performance of the RAID5 tier. Hence, the requests are divided into stripe sizes and then the start and end offsets of each stripe size are determined. In the tier selection decision-making unit, the data placement and migration procedures are applied. Based on the size (and also update frequency, further detail in Section 3.4) of the divided stripes of the request, if the divided stripe is an aligned full stripe write, it is prepared to be sent to the RAID5 tier; otherwise, if it is smaller than the full stripe size, it is prepared to be sent to the RAID1 tier. If any stripes of the request require online migration, it will be performed by the online migration planner. Concurrent with this procedure, the offline migration planner (in tier reclamation unit) monitors the used capacity of the RAID1 tier. It will migrate the RAID1 stripes to RAID5 when the RAID1 tier is full and requires reclamation. The offline migration planner decides which stripes of the RAID1 tier would be more suitable to migrate to the RAID5 tier. The entire process of offline migration planner is managed in the background. Finally, in the dispatcher unit, separate stripes of a request are concurrently sent by the block

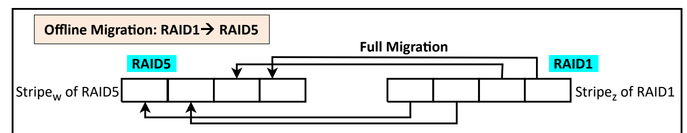
SUS: Stripe Unit Size




(a) Online migration of a stripe from RAID5 to RAID1 tier (requires online data movement)




(b) RAID1 to RAID5 migration of a stripe (without data movement)




(c) Offline migration of a stripe from RAID1 to RAID5 tier (requires offline data movement)

Fig. 5: Examples of tier migrations of an LBA (Logical Block Address) triggered by a,b) update requests for previously written stripes shown in Fig. 4, and c) reaching the full capacity of RAID1 in HybRAID.

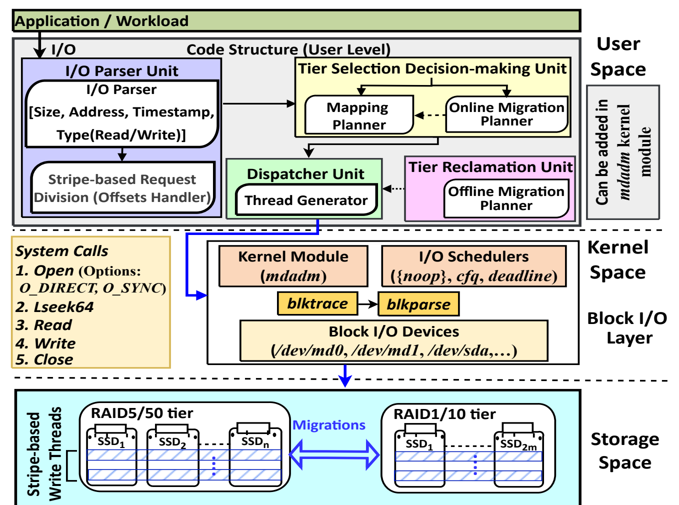



Fig. 6: Architecture of HybRAID in Linux OS stack diagram.

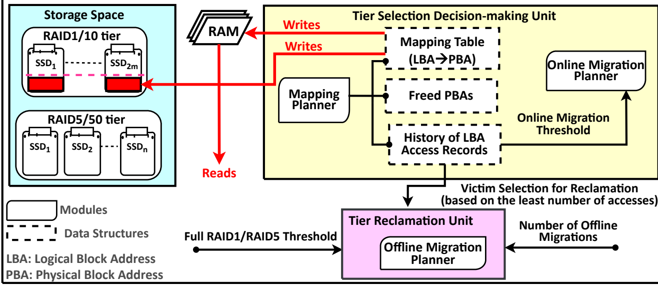



Fig. 7: Modules and data structures of HybRAID with used thresholds for migrations. (`RAM: Random Access Memory`)

I/O layer using a multi-threaded function to enhance performance.

We utilize the *mdadm* software RAID module available in Linux to implement RAID1 and RAID5. Additionally, we employ the *blktrace* and *blkparse* modules to monitor and verify the workloads running on top of the storage devices. We opt for the *noop* scheduler since it offers the best performance for SSDs [15]. The requests are issued directly to the underlying layer of storage devices using the *O_DIRECT* and *O_SYNC* options of the *open* system call function to directly issue requests to the storage layer devices. We utilize the *read* and *write* system calls to issue read and write requests to the block I/O layer of the operating system. The *lseek64* library function is exploited for addressing a device. The *valloc, memset*, and *free* library functions are utilized for allocating and deallocating a buffer in memory, which is employed either in write requests as the content to be written or in read requests as the read content. In the storage layer, we have two tiers of RAID1 and RAID5 consist of enterprise SSDs, which are seen as distinct storage devices. The mapping table is responsible for connecting logical addresses to physical ones. We implement the mapping table in a structure using both SSD and main memory to provide a persistent and high-speed mapping structure (shown in Fig. 7). Write requests that change the mapping between *Physical Block Address* (PBA) and *Logical Block Address* (LBA) will be issued to both SSD and main memory, whereas other requests will only be issued to main memory. Since the mapping table entries are small (4 bytes), we use a portion of the RAID1 tier (shown in Fig. 7) to maintain the mapping table, which provides higher performance for small write requests and also ensures protection against disk failures. Since the migration overhead of HybRAID is quite low (averaging at 6.3%, with further details in Sec. 4.4.1), and the mapping table entries are small in size, the performance of the proposed architecture is not significantly affected.

Fig. 7 illustrates the detailed data and software structures of the HybRAID architecture. The request is first divided into stripe sizes. For each stripe size, the mapping module is executed to determine the physical stripe address and required migrations for the request logical stripes. Before assigning a physical address, each logical address is checked for an entry in the mapping table. If an entry is found, it indicates an update request; otherwise, the request is a new assignment. If it is a new assignment, the corresponding tier is determined from the tiering placement unit, and a physical stripe address is assigned in the mapping table for further access. To allocate physical addresses in each tier, a data structure is used to store freed physical addresses during migrations. Initially, physical address allocation is done linearly up to the size of each tier, and then the data structure of freed physical addresses is utilized. If the requested address already exists in the mapping table, the online migration assessment unit checks whether online migration is necessary. If it is required, the destination physical address is determined and placed in the mapping table. The physical address of the source tier is freed and placed in the data structure of freed physical addresses for subsequent allocations.

In the online migration planner, if the number of write accesses to an address is greater than the specified threshold (*online_migration_threshold)*, online migration (from RAID5 to RAID1) is performed; otherwise, it remains in its current tier. This is determined through a data structure named the history of LBA access records, which dynamically creates a counter for each accessed logical address (in stripe size unit), increments its value after each write access, and removes it when the stripe is migrated. We determine the optimal value for the online migration threshold through empirical experiments with write-intensive workloads. In our experiments, a very low threshold increases migrations with diminished performance gain, as it results in migrating chunks with negligible benefits lower than the migration cost, while a very large threshold reduces the performance gain by neglecting the migration of frequently updated stripes. Hence, the appropriate online migration threshold lies between these values, practically determined to move only chunks where the benefit outweighs the migration cost. This empirically determined value of the online migration threshold is four, which we use in the experiments section.

The offline migration module specifies a filling threshold for RAID5 (*full_RAID5_threshold)* and RAID1 tier (*full_RAID1_threshold)*. To be able to respond to new requests, the offline migration is conducted in the background once the number of occupied physical addresses reaches the specified threshold. During this process, stripes are transferred from the full tier to the free tier. It selects victim stripes to migrate from the full tier based on the access history of logical addresses, prioritizing those with the lowest access count. Another threshold is specified in the offline migration module to determine the number of stripes that must be transferred (*number_of_offline_migrations*) to reclaim the full tier. After specifying the final physical address for each request stripe, the thread generator module sends all the request stripes simultaneously to the storage layer via the OS block I/O layer functions (such as read/write/lseek64 system calls). The entire code structure in the user space can be integrated into the *mdadm* module of the Linux OS kernel.

### 3.4 Algorithm Used in HybRAID

As previously stated, RAID5 provides higher performance for aligned full stripe size write requests while RAID1 is better suited for small or partial stripe writes. If a new write access to an LBA is requested and it is an aligned full stripe, it will be directed to the RAID5 tier. Otherwise, it will be directed to the RAID1 tier. When an update access to an LBA is requested, the destination tier depends on three factors: 1) the current tier, 2) the size of the corresponding stripe of the request (full stripe or small/partial stripe), and 3) the access history of the requested LBA in the current tier. Table 2 demonstrates the placement policy for determining the destination tier of update requests based on the current tier and the size of the write request. If the size of the update request is below the full stripe size, RAID1 may be the best destination tier, but whether migration is necessary depends on the current tier. Migrating from RAID5 to RAID1 requires moving some not-updated chunks from the source tier to the destination tier. In contrast, during the migration from RAID1 to RAID5, full stripe write requests will completely fill the stripe and eliminate the need for any data movements. Instead, it only modifies specific metadata such as the write access history, mapping table, and freed physical addresses during the migrations. In other scenarios where

TABLE 2: Placement policy of HybRAID for update write requests based on current tier and write request size.

| Stripe-based divided update request | Current tier | Destination tier | Migration | Data Movement |
|---|---|---|---|---|
| size < full stripe | RAID5 | RAID1 | Yes | Yes |
| size < full stripe | RAID1 | RAID1 | No | – |
| size = full stripe | RAID5 | RAID5 | No | – |
| size = full stripe | RAID1 | RAID5 | Yes | No |

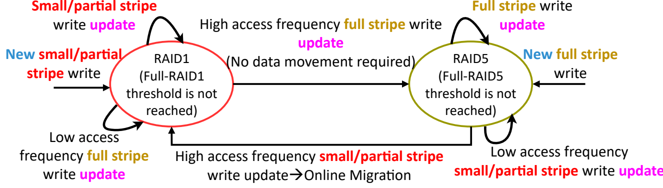



Fig. 8: State diagram of HybRAID.

the destination tier remains unchanged, only the requested chunks will be written, and also certain metadata (such as the history of write accesses) will be updated. Meanwhile, the remaining chunks will remain unchanged without any data movements. Therefore, according to Table 2, only one transition from RAID5 to RAID1 requires data movement.

Fig. 8 illustrates the state diagram of HybRAID, which consists of two states: RAID1 and RAID5. This figure does not include offline migration because both tiers have free space and do not require reclamation. This state diagram shows all possible transitions between the two tiers, which depend on the current tier, the write request size, and the requested LBA access history. For first access write requests, HybRAID directs small/partial stripe sizes to the RAID1 tier, while directing aligned full stripe sizes to the RAID5 tier for higher performance. For second and subsequent write accesses to an LBA (i.e., update requests), HybRAID also considers the LBA write access history. To minimize the number of ineffective migrations, HybRAID changes the LBA tier only when its access history exceeds a predefined threshold called the *online_migration_threshold*. As shown in Fig. 8, migrations (from RAID5 to RAID1 or vice versa) occur only when the access history of the requested LBA reaches this threshold. For instance, if an LBA exists in the RAID5 tier and a small/partial stripe update is requested, it will be migrated to RAID1, provided that its access history in RAID5 reaches the *online_migration_threshold*. Otherwise, it will remain in the same tier (RAID5). It is worth noting that changing the LBA tier in HybRAID does not always result in data movements between the two tiers. For example, if full stripe size requests are issued to RAID1, changing the LBA tier from RAID1 to RAID5 only requires some metadata updates and does not require data movements.

Algorithm 1 depicts the workflow of the tier selection and migration decision, which was presented in Fig. 8. This module triggers online when a write request is dispatched. First, *mapping_table* and *mapping_tier* are set to their initial values (Line 1), which preserve physical addresses and tiers corresponding to LBAs, respectively. Then, for each divided stripe size of the request, the start and end offset within each stripe are determined (Line 4). The assignable physical address is obtained for each tier in Lines 5 through Line 9. If $tier_{(1||5)}_stripe_counter$[3] has not reached the end of the tier space, the assignable physical address will be equal to $tier_{(1||5)}_stripe_counter$. Otherwise, the assignable physical address is extracted from the $tier_{(1||5)}_freed_table$, which is used for keeping freed physical stripes during migrations. When exploiting freed physical addresses in $tier_{(1||5)}_freed_table$, we should erase the used entry since it is currently in use (Line 8). Determining destination tier and online migrations are accomplished from Line 10 through Line 29. This has two states: 1) the first write access (Line 10 through Line 15) and 2) the second (or more) write accesses to the LBA (Line 16 through Line 29). The first access to the LBA is specified such that $Mapping_Table$ has its initial value in that LBA (Line 10). In this state, the divided request size determines its destination tier and the $Mapping_Table[LBA_i]$

3. Here 1||5 represents two variables declared for RAID1 tier:1 and RAID5 tier:5

**Algorithm 1** Tier selection and Migration Decision

$LBA_i$ : Logical block address of $i_{th}$ stripe of $Write_Request$
$Mapping_Table$ / $Mapping_Tier$ : Input is stripe's LBA, output is stripe's PBA / tier
$tier_{(1||5)}_freed_table$ : maintains the freed physical stripe addresses during migrations per tier (RAID1 or RAID5)
$tier_{(1||5)}_history$ : Input is stripe's LBA, output is number of write accesses to the LBA per tier (RAID1 or RAID5)
$migration_i$ : Declares whether $i_{th}$ stripe of request requires migration or not
$tier_{(1||5)}_stripe_counter$ : A counter for traversing on physical stripe address per tier (RAID1 or RAID5)
$tier_{(1||5)}_physical_stripe_add$ : Current assignable physical stripe address per tier (RAID1 or RAID5)

```
1:  Mapping_Table for all entries ← 0;  Mapping_Tier for all entries ← 'N';
    tier(1||5)_stripe_counter ← 0
2:  procedure MAPPING_MODULE (WRITE_REQUEST)
3:    for each stripe size of Write_Request as stripe_i do
4:      Set startoffset and endoffset for stripe_i as stoffset_i and endoffset_i
5:      if tier(1||5)_stripe_counter < tier(1||5)_number_of_stripes then
6:        tier(1||5)_physical_stripe_add ← tier(1||5)_stripe_counter;
    tier(1||5)_stripe_counter += 1
7:      else tier(1||5)_physical_stripe_add ← tier(1||5)_freed_table.begin()
8:        Erase tier(1||5)_freed_table.begin()
9:      end if
10:     if Mapping_Table [LBA_i] = 0 then migration_i ← False
11:       if (stoffset_i + endoffset_i = RAID5_data_disk_number − 1) then
    Mapping_Tier [LBA_i] ← '5'
12:         insert tier5_history[LBA_i] and set it to 1; Mapping_Table [LBA_i] ←
    tier5_physical_stripe_add
13:       else insert tier1_history[LBA_i] and set it to 1;  Mapping_Tier [LBA_i] ←
    '1'
14:         Mapping_Table [LBA_i] ← tier1_physical_stripe_add
15:       end if
16:     else if (Table_Tier[LBA_i] = '5') and (stoffset_i + endoffset_i ≠
    RAID5_data_disk_number − 1) and (tier5_history[LBA_i] ≥
    online_migration_threshold)) then
17:       migration_i ← True;  insert tier1_history[LBA_i] and set it to 1
18:       erase tier5_history[LBA_i];  pre_physical_stripe_add_i ←
    Mapping_Table [LBA_i]
19:       Mapping_Table[LBA_i] ← tier1_physical_stripe_add;
    Mapping_Tier[LBA_i] ← '1'
20:       After receiving the migration ack, insert tier5_freed_table with
    pre_physical_stripe_add_i
21:     else if (Table_Tier[LBA_i] = '1') and (stoffset_i+endoffset_i =
    RAID5_data_disk_number − 1) and (tier1_history[LBA_i] ≥
    online_migration_threshold) then
22:       migration_i ← False;  insert tier5_history[LBA_i] and set it to 1
23:       erase tier1_history[LBA_i];  insert tier1_freed_table with
    Mapping_Table[LBA_i]
24:       Mapping_Table[LBA_i] ← tier5_physical_stripe_add;
    Mapping_Tier[LBA_i] ← '5'
25:     else migration_i ← False
26:       if Table_Tier[LBA_i] = '5' then  tier5_history[LBA_i] += 1
27:       else tier1_history[LBA_i] += 1
28:       end if
29:     end if

    /* Set thread_write_data of all stripes of the request to be dispatched in
    multi-thread manner to the storage layer                                  */
30:     thread_write_data_i.address ← Mapping_Table[LBA_i]
31:     thread_write_data_i.tier ← Mapping_Tier[LBA_i]
32:     set thread_write_data_i.blksize to stripe_i's size in bytes
33:     thread_write_data_i.migration ← migration_i

    /* Set thread_migration_data of migration stripes of the request to be dispatched in
    multi-thread manner to the storage layer                                  */
34:     if migration_i = True then
35:       thread_migration_data_i.pre_address ← pre_physical_stripe_add_i
36:       thread_migration_data_i.new_address ← Mapping_Table[LBA_i]
37:     end if
38:   end for
39: end procedure
```

and $Mapping_Tier[LBA_i]$ will be inserted to their corresponding values (Lines 11 through 14). Additionally, the history of the destination tier for that LBA is inserted and initialized to '1' by $tier_{1||5}_history[LBA_i]$ (Lines 12, 13).

There are three different states in the second (or more) write access to the LBA (i.e., update access). The algorithm for the first state is from Line 16 to Line 20 and is related to migration from RAID5 to RAID1 tier. According to Table 2, if the current tier of the LBA is the RAID5 tier and an ongoing request aims to update to a small/partial stripe size and the write access frequency to that LBA is higher than the *online_migration_threshold*, we migrate it to the RAID1 tier. We also erase the history of the LBA in its previous tier (RAID5). Then, we insert the history of that LBA in the RAID1 tier and initialize it to '1' (Line 17 and Line 18). Since the physical address of the previous tier (RAID5) is no longer required after receiving migration acknowledgment from the thread generator, we put this address in the $tier_5_freed_table$ to be used for future requests (Line 20). The algorithm for the second state is from Line 21 to Line 24 and is related to migrating the tier of a stripe from RAID1 to RAID5. Since the request fill the stripe, there are no residual chunks to migrate from RAID1 and the migration flag (here declares data movement) is set to false (Line 22). For the third state, if it does not fall into the first or second states, the destination tier is the same as before and no migrations are required. We will only increase the LBA access frequency in the corresponding tier history (Line 26 and Line 27). These are the cases where the state diagram in Fig. 8 depicts them as self-loop edges (low access frequency or updating with the request size suited at its current tier). At the end of this algorithm, we will assign the required data (address, tier, and request size) to the *thread_write_data* (Line 30 through Line 33) and *thread_migration_data* (Line 34 through Line 37) data structures to prepare the write and migration threads. The thread generator module in the dispatcher unit will later use these data structures to simultaneously write the stripes of the request to the destination tiers.

## 4 Experimental Results

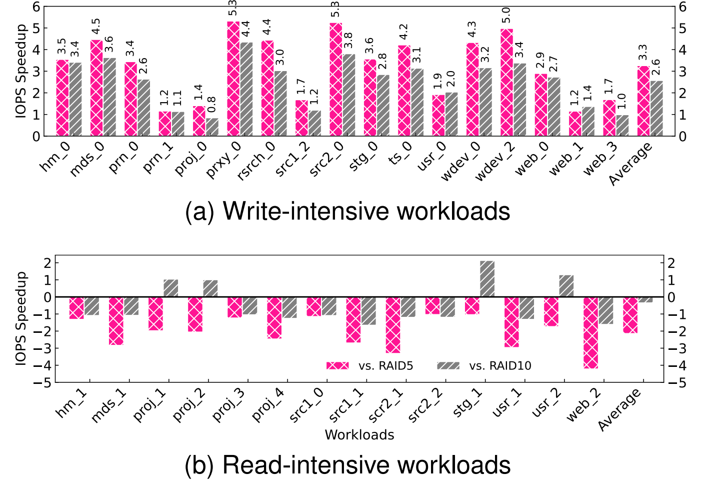
This section presents comprehensive experiments to assess the performance improvements resulting from the proposed architecture. Furthermore, we examine the impact of implementing this architecture on endurance and the associated overheads.

### 4.1 Experimental Setup

To evaluate the proposed architecture, we conducted experiments on a real testbed: an HP ProLiant DL180 Gen9 server with 24 Intel(R) Xeon(R) CPU E5-2650 v4 cores, each running at 2.20GHz. The server is equipped with 4 sets of 16GB DDR4 RAM and five Samsung SSD 860PROs, each with a capacity of 256GB. We construct RAID5, RAID10, and the proposed tiered architecture (HybRAID) using these SSDs. We implement HybRAID at the user level on the server, which runs the Linux operating system with Ubuntu Server 18.04.5 LTS distribution using 5.15.0-46-generic kernel. We set up the RAID using the *mdadm* software RAID Linux kernel module. In all experiments, we use a stripe unit size of 4KB, which provides the highest performance and the least amount of extra writes in stripe-based RAID levels [26]. To compare HybRAID with mirroring and parity-based RAIDs (RAID10 and RAID5), we implement RAID10 and RAID5 using four SSDs. The proposed architecture uses the same SSDs with three SSDs in the RAID5 tier and two SSDs in the RAID1 tier. We partition each disk to 30GB and use the partitioned space in the RAID construction. As a result, the usable capacity of RAID5 and

TABLE 3: Write and read ratio in write- and read-intensive workloads with respect to the number of requests, and also the number of online migrations for write-intensive workloads.

| | Workload | Total request size (GiB) | Write requests (#) | Read requests (#) | Migrations (#) | | Workload | Read ratio (#) |
|---|---|---|---|---|---|---|---|---|
| Write-Intensive Workloads | hm_0 | 44.6 | 2,575,570 (64.5%) | 1,417,750 (35.5%) | 162,261 (6.3%) | Read-Intensive Workloads | hm_1 | 95.3% |
| | mds_0 | 14.9 | 1,067,060 (88.1%) | 143,973 (11.9%) | 53,353 (5%) | | mds_1 | 92.9% |
| | prn_0 | 78.4 | 4,983,410 (89.2%) | 602480 (10.8%) | 303,988 (6.1%) | | proj_1 | 89.4% |
| | proj_0 | 169.2 | 3,697,140 (87.5%) | 527,381 (12.5%) | 366,017 (9.9%) | | proj_2 | 87.6% |
| | prxy_0 | 94.7 | 12,135,400 (96.9%) | 383,524 (3.1%) | 206,302 (1.7%) | | proj_3 | 94.8% |
| | rsrch_0 | 17.5 | 1,300,030 (90.7%) | 133,625 (9.3%) | 75,402 (5.8%) | | proj_4 | 98.5% |
| | src_1_2 | 60.1 | 1,423,690 (74.6%) | 484,079 (25.4%) | 125,285 (8.8%) | | src_1_1 | 95.3% |
| | src_2_0 | 16.2 | 1,381,080 (88.7%) | 176,729 (11.3%)) | 70,435 (5.1%) | | src_2_1 | 97.8% |
| | stg_0 | 29.6 | 1,722,480 (84.81%) | 308,437 (15.19%) | 127,464 (7.4%) | | stg_1 | 63.8% |
| | ts_0 | 22 | 1,485,040 (82.4%) | 316,692 (17.6%) | 66,827 (4.5%) | | usr_1 | 91.5% |
| | usr_0 | 56.8 | 1,333,410 (60%) | 904,483 (40%) | 96,006 (7.2%) | | usr_2 | 81.1% |
| | wdev_0 | 14.1 | 913,732 (79.9%) | 229,529 (20.1%) | 59,393 (6.5%) | | web_2 | 99.2% |
| | wdev_2 | 2.1 | 181,077 (99.9%) | 189 (0.1%) | 15,211 (8.4%) | | src_1_0 | 56.5% |
| | web_0 | 36.4 | 1,423,460 (70.1%) | 606,487 (29.9%) | 6,645 (9%) | | | |
| | web_1 | 5 | 73,833 (45.9%) | 87,058 (54.1%) | 3,544 (4.8%) | | | |
| | web_3 | 1.25 | 21,330 (68%) | 10,050 (32%) | 342 (1.6%) | | | |
| | AVG | | (79.5%) | (20.5%) | (6.3%) | | AVG | 87.9% |




(a) Write-intensive workloads

(b) Read-intensive workloads

Fig. 9: Performance gain of HybRAID compared to flat[4] RAID5 and flat RAID10 in write- and read-intensive workloads.

the proposed architecture is 90GB, while RAID10 with four disks has 60GB. The detailed specification of exploited write- and read-intensive workloads (SNIA block I/O traces [25]) are reported in Table 3. Also, for a comparison with more recent studies on RAID5 optimizations for small writes, indeed CRAIS5 [22], we implement CRAIS5 using OpenCAS (Open Cache Acceleration Software) developed by Intel [3]. This implementation incorporates a RAM cache for effectively caching small writes. We utilize version 20.03.3.0307 of OpenCAS with a cache-line size equal to 4KB on the Ubuntu Server 18.04.5 LTS operating system. A RAM disk in Linux is created to serve as the RAM cache in CRAIS5. To be able to implement CRAIS5 on real SSDs and RAID systems, minor modifications are made to its implementation, ensuring no impact on its performance improvement. Instead of employing a write-through cache between the RAM cache and the main array, a write-only RAID1 cache is implemented. In this configuration, one drive is an SSD and the other is a RAM disk. This setup maintains the same performance and functionality as CRAIS5, caching small writes and protecting the cache value by mirroring it with the SSD inside RAID1 without updating the parity.

### 4.2 Performance Improvement

Fig. 9 shows the performance improvement of the proposed architecture for (a) write- and (b) read-intensive workloads com-

4. "Flat" typically refers to a pure RAID array that has not been modified or adjusted in any way. For example, in a flat RAID5 array with $N$ disks, data blocks from $N-1$ disks is used for storage, while the remaining disk is dedicated to parity.

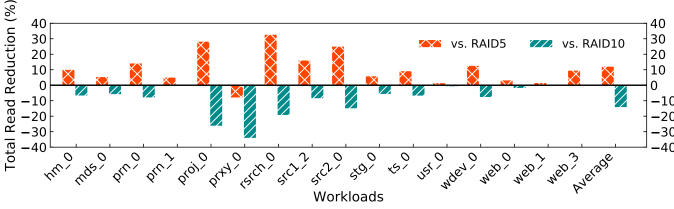



Fig. 10: Percentage of read overhead reduction of HybRAID compared to flat RAID5 and flat RAID10 in write-intensive workloads.

pared to RAID5 and RAID10 with equivalent costs. As can be seen in Fig. 9a, HybRAID with five disks (3+2) (three disks in RAID5 and two disks in RAID1 tier) improves the performance in IOPS (I/O Per Second) on average by 3.3× and 2.6× compared to RAID5 (3+1) and RAID10 (2+2), respectively. Since write-intensive workloads are a mixture of small and large size writes, HybRAID outperforms both RAID10 and RAID5, which are best suited for *only* one kind of write requests, i.e., RAID5 for large writes and RAID10 for smaller size writes. HybRAID offers higher performance than RAID10 because it exploits parity-based RAID for full stripe writes, which have higher performance compared to RAID1. However, the performance improvement obtained by HybRAID over RAID10 is always lower than its improvement against RAID5. This is due to the significant impact of small/partial stripe write performance overhead on parity-based RAIDs, which is addressed and resolved by HybRAID. Fig. 10 illustrates the total percentage reduction in read overhead of HybRAID compared to RAID5 and RAID10 in write-intensive workloads. It shows that the proposed architecture leads to an average reduction of 12.3% in read overhead compared to RAID5. This is because HybRAID redirects small or partial stripe writes to the RAID1 tier, which helps to reduce the number of reads needed for parity updates. On the other hand, since no additional reads are performed in RAID10, as expected, HybRAID induces more reads than RAID10 by 14.4% in write-intensive workloads. This is due to the additional reads required for parity updates within the RAID5 tier of HybRAID. Indeed, HybRAID does not redirect all small and partial stripe write requests to the RAID1 tier unless their update frequency exceeds the online migration threshold. Instead, these requests are rewritten to the RAID5 tier, necessitating extra pre-reads. This contributes to the observed extra 14.4% reads in HybRAID compared to traditional RAID10.

In read-intensive workloads, as depicted in Fig. 9b, HybRAID exhibits lower performance on average compared to RAID5 and RAID10, with reductions of -2.1× and -0.3×, respectively. Since some values are positive, the average reduction factor can fall between -1 and 0, where negative values indicate a decrease in performance. This disparity in performance can be attributed to the fact that parity-based RAIDs outperform RAID1/10 in terms of read performance due to their larger number of disks, enabling a higher level of concurrency. Analyzing read-intensive workloads reveals that the most significant performance decrease is linked to the workloads with a high percentage of read requests (exceeding 92%), where dominant requests (over 74%) have a large size (68 KB). This observation is attributed to the fact that larger-sized read requests typically perform better in traditional RAID5 due to the distribution of requests among more disks.

Also, Fig. 11 shows the performance gain obtained by HybRAID over RAID5 and the performance gain obtained by CRAIS5 over RAID5 with three usable SSDs in each array. As can be seen, in write-intensive workloads, HybRAID results in an average 3.3× performance gain over RAID5, while CRAIS5 leads to an average 1.5× performance gain over RAID5. This

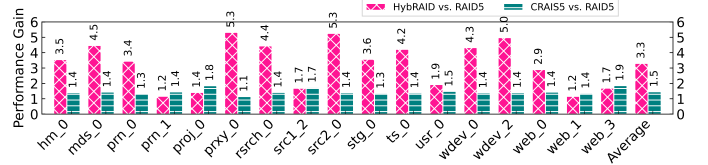
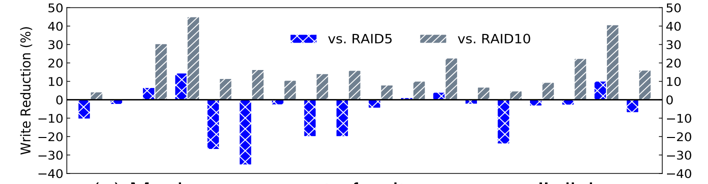
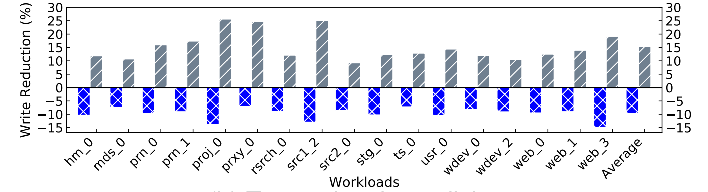



Fig. 11: Performance gain of HybRAID over RAID5 versus performance gain of CRAIS5 over RAID5 with three usable SSDs for write-intensive workloads.




(a) Maximum amount of writes among all disks



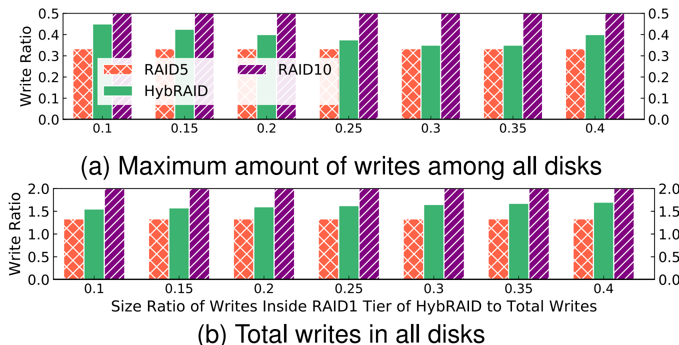

(b) Total writes in all disks

Fig. 12: Write reduction percentage of HybRAID compared to flat RAID5 and flat RAID10 in write-intensive workloads.

is because CRAIS5 still directs partial stripe writes to RAID5 without enhancing their performance, leading to additional read requests. Moreover, partial stripe writes may contain small writes within them, which CRAIS5 does not consider for caching. Furthermore, HybRAID incorporates a multi-threading mechanism for large requests to further enhance the performance, which is not implemented here. Lastly, the improvement of CRAIS5 relies on the locality of the workload to reduce the small writes flushed to the main array, a restriction that HybRAID does not have.

### 4.3 Endurance Results

We obtained endurance results as shown in Fig. 12 by measuring the maximum of writes among all disks (Fig. 12a) and the total of the writes across all disks (Fig. 12b) for RAID5, RAID10, and HybRAID under write-intensive workloads. On average, HybRAID results in a 16.1% reduction in maximum of writes compared to RAID10 and a 7% increase in maximum of writes compared to RAID5 (Fig. 12a). These results are based on the final implementation of HybRAID, incorporating all aspects of the architecture, including the migration policies. In a few workloads (such as `prn_0`, `prn_1`, and `web_3`), however, HybRAID resulted in a lower maximum write than RAID5. To analyze this result, we conducted a theoretical estimation of the maximum writes for RAID5, RAID10, and our proposed architecture (Fig. 13). We assume that write requests are evenly distributed among disks, so the portion of writes per disk for RAID5 is $\frac{1}{N-1}$ and for RAID10 is $\frac{1}{N/2}$, where $N$ is the total number of disks. To obtain the theoretical maximum writes for HybRAID, we calculated the maximum amount of writes associated with RAID5 and RAID1 tiers. We consider the write portion of the small/partial stripe writes for the workloads, denoted as $Ratio_{SmallSizes}$, which are redirected to the RAID1 tier. Assuming an even distribution of writes between disks, the write portion in the RAID1 tier is equal to $Ratio_{SmallSizes}$, and in the RAID5 tier, it is equal to




(a) Maximum amount of writes among all disks

(b) Total writes in all disks

Fig. 13: Theoretical write portion in flat RAID5, HybRAID, and flat RAID10 for different size ratios of workloads that are directed to the RAID1 tier in HybRAID (i.e., small/partial stripe writes).

$\frac{(1-Ratio_{SmallSizes})}{N-2}$. Thus, the maximum write portion is obtained from the maximum of these two values. Fig. 13a shows the maximum write in each architecture versus $Ratio_{SmallSizes}$ (up to 40% of write workload size) when $N = 4$. As can be seen, the maximum write portion of RAID5 and HybRAID is almost equal for $Ratio_{SmallSizes}$ ranging from 0.25 through 0.35. Hence, the write portion of HybRAID could be lower than RAID5 in the practical results shown in Fig. 12a. This is because migrations are not counted in the theoretical measurements. Additionally, the distributions of writes between disks in real-world workloads depend on the workload request addresses and additionally, parity writes could also affect the distributions, which are not considered in the theoretical estimations. In Fig. 13a, the maximum amount of writes in RAID10 is always higher than that in HybRAID, which was also seen in the practical results shown in Fig. 12a (on average, 16.1% write reduction).

As shown in Fig. 12b, HybRAID results in a 15.3% reduction in the total of writes compared to RAID10 and 9.8% increase in the total of writes compared to RAID5. The theoretical estimations presented in Fig. 13b validate the practical results shown in Fig. 12b, where the total write portion of HybRAID always falls between RAID5 and RAID10. This is because the proposed architecture uses parity-based RAID alongside RAID1, which writes *only* a parity for the whole stripe. In contrast, RAID10 repeats all data in the corresponding redundant disks, producing more writes than HybRAID. On average, HybRAID produces 9.8% more total writes than RAID5. This endurance overhead of HybRAID is considerably lower compared to its performance gain. Nevertheless, we will use heterogeneous SSDs in each tier to cost-effectively compensate this write increase of HybRAID, as we will describe in Section 4.4.2.

## 4.4 Overheads

In this section, we quantify the overheads associated with HybRAID, including migration overhead, processing overhead, and cost overhead. Additionally, we present the performance-per-cost results of HybRAID compared to flat RAID5 and RAID10. Furthermore, we explore potential applications of the proposed architecture and outline future research directions in this area.

### 4.4.1 *Migration and Processing Overhead*

Figure 14 illustrates the migration percentage relative to the total number of writes for write-intensive workloads. In the case of HybRAID, online migration occurs exclusively when updating a small or partial stripe write request that was previously located in the RAID5 tier. On average, this type of migration accounts for only 6.3% of the total writes in HybRAID, where the number

Fig. 14: Percentage of I/O requests migrated by HybRAID in write-intensive workloads.

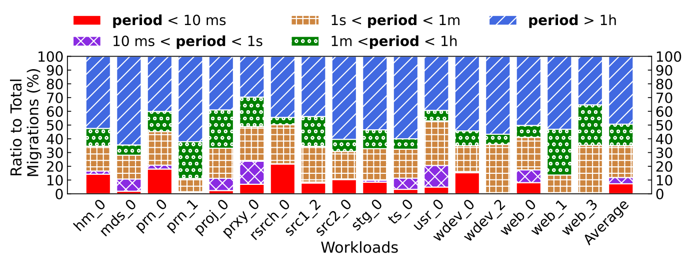



Fig. 15: Time period between migrations in write-intensive workloads (s: second, ms: millisecond, m: minute, h: hour.).

of migrations per workload is demonstrated in Table 3. This low percentage is due to the use of access history for write accesses in each tier, which sets the threshold for migrating data from RAID5 to RAID1. Fig. 15 illustrates the periods between migrations for write-intensive workloads, which depends on the workload characteristics. These results show that the total of 6.3% migrations (shown in Fig. 14) are, on average, more than ten milliseconds apart from each other for 92.7% of the time, and only 7.3% of them are less than ten milliseconds apart. Since we are using high-performance SSDs, which can handle multiple requests simultaneously in less than one millisecond, the overhead caused by migrations in HybRAID is negligible. Therefore, the proposed architecture can achieve performance gains even when migration overheads are included.

The processing overhead of using HybRAID is only 75% of a processor core compared to conventional RAIDs. This information is obtained using the user command *htop* available in Linux. Obviously, with the prevalence of multi-processor systems with hundreds of cores, this overhead becomes negligible.

### 4.4.2 *Cost Overhead*

The cost overhead of HybRAID includes the cost of a mapping table and an extra disk compared to conventional RAID5. In HybRAID, addressing is based on the stripe size, and each entry in the mapping table links a physical stripe address to a logical stripe one. The size of each entry must be four bytes in order to address the stripes. Table 4 shows the mapping table overhead based on the storage size, number of disks in the RAID5 tier, and various stripe unit sizes. For example, suppose there are three disks in the RAID5 tier (plus two disks in the RAID1 tier) and a 4KB stripe unit size. In this case, the stripe size is 8KB (8192 bytes), and the overhead is $\frac{4}{8192} = 0.05\%$ of the storage size. As can be seen from the table, the maximum amount of mapping table overhead is only 0.05% of the storage size and it decreases with an increase in the stripe unit size or the number of disks in the RAID5 tier.

The extra storage cost of HybRAID is due to the exploitation of the RAID1 tier, which requires one additional SSD compared to the flat (baseline) RAID5 with equal usable capacity. Fig. 16 illustrates the cost overhead/cost-benefit of HybRAID compared to flat RAID5 (baseline) and flat RAID10, respectively, both with equal usable capacity. When using homogeneous SSDs such as flat (baseline) RAID5, the cost overhead of HybRAID with $N+1$

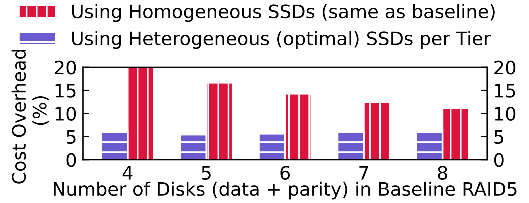
TABLE 4: Mapping table overhead regarding storage size for various numbers of RAID5 disks and stripe unit sizes.

| # of Total disks in RAID5 tier | Stripe unit size | Stripe size | Overhead |
|---|---|---|---|
| 3 | 4KB | 8KB | 4B / 8KB = 0.05% |
| 5 | | 16KB | 4B / 16KB = 0.02% |
| 3 | 8KB | 16KB | 4B / 16KB = 0.02% |
| 5 | | 32KB | 4B / 32KB = 0.01% |

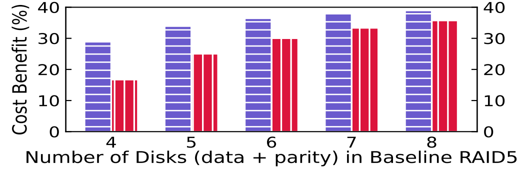
disks over flat RAID5 with $N$ disks is equal to $\frac{(N+1)-N}{N+1} = \frac{1}{N+1}$, where $N$ is the total number of disks in flat RAID5. Additionally, the cost-benefit of HybRAID over flat RAID10 is determined by the number of disks used in each architecture with equal usable capacity (for $N = 4$, RAID10 requires six disks equal to $(2 \times (N-1))$ while HybRAID requires five disks equal to $(N+1)$). Hence, the cost-benefit of HybRAID over flat RAID10 is $\frac{2\times(N-1)-(N+1)}{2\times(N-1)} = \frac{N-3}{2\times(N-1)}$. So, as shown in Fig. 16a when using homogeneous SSDs, the cost overhead of HybRAID compared to flat RAID5 decreases from 20% to 11.1% as $N$ grows from 4 to 8 disks, while the cost-benefit over flat RAID10 increases from 16.7% to 35.7% (Fig. 16b).

Since the amount of writes among the two tiers of HybRAID is different, we suggest using heterogeneous SSDs in each tier to provide the best-tuned endurance per tier at an efficient cost. Assuming a completely even distribution of requests between data disks, the writes inside flat RAID5 are equal to $\frac{S}{N-1}$ per disk (flat RAID5 has $(N-1)$ **data** disks), where $S$ is the write workload size. Considering that 70% of the write workload is issued to the RAID5 tier in HybRAID (as described in the workload characterization in Fig. 3), the writes inside the RAID5 and RAID1 tiers of HybRAID are equal to $\frac{S\times0.7}{N-2}$ (the RAID5 tier has $(N-2)$ **data** disks) and $0.3 \times S$, respectively. Hence, the writes inside the RAID5 tier are always lower than flat RAID5, while the RAID1 tier always has more writes than flat RAID5 per disk. Therefore, we can use higher and lower endurance SSDs than flat RAID5 for the RAID1 and RAID5 tiers in HybRAID, respectively, which are optimal in terms of endurance and cost while maintaining performance. To measure the cost of optimal SSDs in the two tiers, we utilize the ratio between endurance and cost of real enterprise SSDs with the same performance. We observe that by doubling the endurance, the cost becomes 1.7× higher (Samsung 970 PRO (higher endurance) [2] vs. Samsung 970 EVO (lower endurance) [1] with almost the same performance). Hence, the endurance-to-cost ratio will be $\frac{2\times}{1.7\times} = 1.17\times$. Therefore, considering this endurance-to-cost ratio, we calculate the optimal SSD costs for RAID5 and RAID1 tiers of HybRAID compared to flat RAID5. Equation 1 declares the total cost ratio of HybRAID compared to flat RAID5 using optimal heterogeneous SSDs per tier. Using this equation, Fig. 16a demonstrates the cost of HybRAID compared to flat RAID5 using optimal SSDs in each tier. As can be seen, the cost-overhead is at most only 6.4%, which is considerably reduced compared to using homogeneous SSDs (resulting in 1.7× up to 3.2× lower cost-overhead than using homogeneous SSDs). Furthermore, as depicted in Figure 16b, utilizing heterogeneous SSDs yields a cost-benefit advantage of at least 28.9% over flat RAID10. This advantage translates into more cost saving benefits compared to using homogeneous SSDs, resulting in a 1.1× up to 1.7× improved cost-benefit. Moreover, this method leads to improved endurance, as each tier requires different endurance levels relevant to their writes.

$$Total\ Cost\ Ratio = \frac{(N-1)\times[\frac{(N-1)\times0.7}{N-2}]+2\times[0.3\times(N-1)]}{Endurance-to-Cost\ Ratio=(1.17)} \quad (1)$$




(a) Cost overhead over flat RAID5




(b) Cost benefit over flat RAID10

Fig. 16: Cost overhead/cost-benefit of HybRAID with heterogeneous (optimal) or homogeneous (same as flat RAID5) SSDs compared to flat RAID5/flat RAID10 with equal usable capacity, respectively.

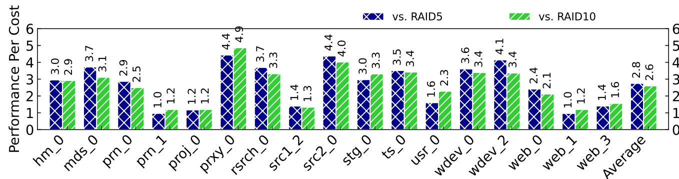
### 4.5 Performance Per Cost Results

Fig. 17 illustrates the comparison of performance per cost of HybRAID compared to flat RAID5 and flat RAID10. The comparison is done using a) homogeneous SSDs same as flat RAID5 as depicted in Fig. 17a and b) heterogeneous SSDs per tier (optimal) as shown in Fig. 17b. The usable capacity in HybRAID, RAID5, and RAID10 is equivalent to three SSDs, with RAID5, RAID10, and HybRAID having a total number of four, six, and five SSDs, respectively. The performance per cost metric is calculated by dividing the performance improvement (as described in Section 4.2) by the cost-overhead (as discussed in Section 4.4.2) of HybRAID over flat RAID5 and flat RAID10. Performance per cost exceeding "1" indicates superior performance when considering the cost factor. With homogeneous SSDs, HybRAID shows an average of 2.8× and 2.6× improvement in performance per cost compared to flat RAID5 and flat RAID10, respectively. Using heterogeneous SSDs, on average, HybRAID results in a 3.1× and 3.0× improvement in performance per cost compared to flat RAID5 and flat RAID10, respectively.

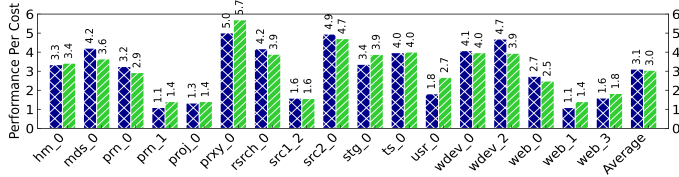
### 4.6 Discussion

In this section, we present several essential capabilities of HybRAID that can be leveraged in future research or serve as a basis for discussions on the versatility of the architecture.

**Contributing factors to the overall performance gain:** Comparing tests without migration to scenarios including online migration implementation, we found that implementing migration results in an average performance improvement of 15.4% (up to 24.5% in the results). This improvement is contingent upon workload characteristics, with online migration accounting for an average of 6.25% in write-intensive workloads (shown in Fig. 14), contributing to the overall performance improvement of 15.4%. The most significant performance increase factor over RAID5 is attributed to the tiering between RAID5 and RAID1, delivering improvements from both arrays with stripe-based address mapping, averaging 66.7%. Another noteworthy factor is the multi-threading module in HybRAID, affecting performance by an average of 17.9%. This approach is crucial for handling large requests, as HybRAID divides original requests into stripe-size segments, optimizing performance gain by managing each segment simultaneously.




(a) Performance per cost of HybRAID using homogeneous SSDs




(b) Performance per cost of HybRAID using heterogeneous (optimal) SSDs per tier

Fig. 17: Performance per cost results of HybRAID compared to flat RAID5 and flat RAID10 with equal usable capacity (three SSDs) using a) homogeneous (same as flat RAID5) and b) heterogeneous (optimal) SSDs.

**Scalability of HybRAID when the number of disks scales:** Firstly, as data volume and disk count increase, migration tends to rise, but this is counterbalanced by the improved performance of RAIDs with more disks. Secondly, the simplicity of our migration policies, avoiding complex computations, ensures that scaling the number of disks does not introduce additional complexity or overhead to migrations. Thirdly, the direct placement and retrieval of metadata with an O(1) complexity minimizes the impact of expanding storage capacity on metadata access complexity. Finally, we partition the total disks into two tiers based on the results gathered in Fig. 3, which shows that the average percentage of small writes in real workloads is 30%. Accordingly, we allocate 30% of the total disks to the RAID10 tier. For example, if the total number of available disks is 12, then 30% (4 disks) will be allocated to the RAID10 tier, and the remaining 70% (8 disks) will be allocated to the RAID50 tier. We also recommend using RAID5(3+1) as the base RAID5 configuration for RAID50 due to its better reliability and acceptable performance. For this example, in the RAID50 tier, we have RAID0 across two RAID5(3+1) arrays, and in the RAID10 tier, we have RAID0 across two RAID1(1+1) arrays. Consequently, the performance improvement remains consistent.

**Compatibility to read-intensive workloads:** Read requests exhibit excellent performance in both RAID5 and RAID10, responding from all disks simultaneously without additional operations or overheads. Consequently, read-intensive workloads are not the focus of this work, as normal RAIDs are more suitable for them. This is while, write requests can experience performance degradation in both RAID5 and RAID10, for small or partial stripe writes and full stripe writes, respectively. Moreover, write requests are generally slower than read requests in SSDs due to inherent characteristics of flash memories and write amplification of SSDs. Hence, the objective of this paper is to address the write performance degradation in RAID levels and propose optimizations to reduce the write overheads. However, for applications generating a substantial number of read requests, a suggested future enhancement involves introducing a migration policy. This policy would entail relocating frequently accessed read-intensive requests, irrespective of their size, to the RAID5 tier. The aim is to leverage the larger number of disks in RAID5 for enhanced parallel data access, ultimately boosting read performance.

## 5 Related Works

The proposed solutions for reducing the overheads of parity updates in parity-based RAIDs commonly fall into two main categories.

*1) Parity logging:* In HDD-based RAID5, [29] suggests using an additional HDD that logs the parity instead of writing them directly to the main array. This log disk generates the parity between the old and new values of the updated data chunk and sequentially issues them to the log disk. Although it reduces the number of extra reads and writes from two to one, it fails to completely eliminate the read overhead because it still needs to read the old value of the updated data. Another research initiative, *EPLOG*, [18] proposes a parity logging scheme for SSD-based RAID5. It takes advantage of the out-of-place update characteristics of SSDs, where invalid data remains in the SSD until Garbage Collection (GC)[5] process [31], [27], [20] is established. *EPLOG* uses this invalid data so as not to read the old value, which is already set as invalid (unlike [29]). The old value is also utilized to recover the operational data within a stripe in case of disk failure. However, although *EPLOG* eliminates the read-before-write process, it requires a new garbage collector that constructs new parity for each stripe. This GC operation imposes considerable time overhead and cannot be used for commodity SSDs with built-in controllers. In addition, if a disk fails, the controller must also recover the old data, which can extend the recovery process and reduce the reliability. To address the limitations of *EPLOG*, *RFPL* [38] uses workload skewness to reduce the number of old data per stripe. However, parity logging solutions still fail to match the sequential performance of the log disk in the steady-state. Furthermore, exploiting HDDs for all-flash storage systems will significantly reduce the overall performance.

*2) Parity or data caching:* The *Leavo Cache* [16] utilizes an SSD cache for HDD-based RAID5, which sends the updated data to both RAID and cache while also storing the old data in the cache. This ensures that updated data is protected in mirroring form, while parity protects the other chunks within a stripe. However, this solution needs to pre-read the old data, so the read-before-write problem for small writes still exists. The *CRAIS5* [22] is similar to *Leavo Cache*, but it is suggested for SSD-based RAID5. Due to the out-of-place update characteristic of flash memories, *CRAIS5* does not require pre-reading the old data, and it only stores the updated data in the cache. However, the limitation of this work, like *EPLOG*, is related to the overheads imposed on GC operation. The *PPC* [11], [10] is similar to *CRAIS5* and suggested for SSD-based RAID5. The difference is that it reduces the size of cached data by storing the parity of updated data instead of pure data. To efficiently leverage the cache presented on *PPC*, another study [39] proposes demoting highly correlated accessed chunks of a stripe from the cache to the RAID5. Other studies [35], [12] enhance existing caching/tiering methods in all-flash data storage systems, which are orthogonal to our work.

*Other related works:* Small random writes can also fragment the physical space of SSDs and lead to more GC operations. To address this issue, *HPDA* [21] uses an HDD-based RAID1 for SSD-based RAID4, which absorbs small random writes and demotes them to RAID4 during idle time. Additionally, a portion of one HDD is used for storing parity to mitigate the endurance problem of SSDs. However, exploiting HDDs for AFS will significantly

5. "GC" involves the identification and reclamation of SSD blocks that contain invalid data in order to optimize storage efficiency and maintain performance. This process is similar to defragmentation in traditional HDDs.

decrease the performance. Moreover, they work for high workload skewness; otherwise, the parity update overheads remain. *LDM* [36]/*HRAID6ML* [40] proposes an HDD log disk mirroring for SSD-based RAID5/RAID6, respectively, as a solution similar to *HPDA*. So at the same way, using HDDs for AFS will significantly decrease performance. Besides, they work for high workload skewness; otherwise, the parity update overheads remain. *RAFS* [30] proposes a new file system for SSD-based RAIDs, which allocates space based on stripe groups (instead of file) at the block level and then absorbs the small writes in one RAM buffer. *FRA* [17] delays the parity update to the idle time of storage array to mitigate the parity update overheads. However, this scheme will have data loss between idle times in case of disk failure. Wilkes et al. [34] aim to achieve a cost-performance benefit through the use of a tiered-based structure in HDD-based RAID. Their approach involves employing a mirroring array as the performance tier and a parity-based array as the capacity tier to optimize cost and performance. They embed mirrored stripes in a parity-based RAID to implement both in one structure, but it wastes the usable capacity in mirrored stripes because other requests cannot use free chunks of these stripes. They assign frequent and infrequent access data to mirroring and parity-based portions, respectively. However, it is assumed that mirroring RAIDs consistently outperform parity-based RAIDs in all cases while the impact of request size and alignment on allocations and migrations has been neglected. This is while our investigations reveal that parity-based RAIDs exhibit poor performance compared to mirroring RAIDs for *both* small (using the read-modify-write parity update scheme) and partial stripe writes (using the read-construct-write parity update scheme) to a similar extent.

Another group of studies suggests new parity generators, such as those presented in [13], [14], [23], which change the normal parity production in a RAID5 array to reduce the parity update overheads. Another group try to modify the architecture of RAID levels to improve their performance in real time [9], [32] or in the reconstruction process [41]. *StRAID* [32] employs a one-worker-per-stripe model, which reduces the number of stripe-states and their lock-based checks, and a fine-grained stripe-level lock to mitigate contentions on shared data structures. Gu et al. [9] explore the implementation of RAID over SSDs with built-in transparent compression, allowing for the possibility of elastically mixing RAID levels to improve performance without compromising storage capacity. The proposed elastic RAID can dynamically adjust the mixture of RAID5 and RAID10, as well as retain drive failure protection during RAID level conversion. However, the practical implementation of elastic RAID is non-trivial due to differences in data mappings and storage capacity between RAID5 and RAID10, and the effectiveness of elastic RAID depends on the runtime compressibility of user data.

## 6 Conclusion

In this paper, we demonstrated that parity- and mirroring-based RAIDs alone cannot offer high performance for write workloads consisting of various write request sizes. Previous studies only targeted the small write performance problem of parity-based RAIDs and did not address the poor performance of their partial stripe writes. Furthermore, RAID1 write performance at full stripe writes is much lower than parity-based RAID and is also more expensive. To mitigate these performance problems, we proposed *HybRAID*, a high-performance hybrid RAID architecture for write-intensive applications in all-flash storage systems. HybRAID is a tiered architecture consisting of two tiers, RAID5 and RAID1. By directing full stripe writes to parity-based RAID and small/partial stripe writes to RAID1, it aims to improve performance. Compared to flat RAID5 and RAID10 with equal cost, HybRAID achieves, on average, $3.3\times$ and $2.6\times$ higher performance. Additionally, it provides an average performance per cost improvement of $3.1\times$ and $3.0\times$ compared to flat RAID5 and RAID10, respectively.

## References


[1] 970EVO. 2018. SAMSUNG SSD. [Online]. Available: https://semiconductor.samsung.com/consumer-storage/internal-ssd/970evo/. Accessed: Jul. 2023.

[2] 970PRO. 2018. SAMSUNG SSD. [Online]. Available: https://semiconductor.samsung.com/consumer-storage/internal-ssd/970pro/. Accessed: Jul. 2023.

[3] Open-CAS. [Online]. Available: https://open-cas.github.io/. Accessed: Feb. 2024.

[4] Mohammadamin Ajdari, Pouria Peykani Sani, Amirhossein Moradi, Masoud Khanalizadeh Imani, Amir Hossein Bazkhanei, and Hossein Asadi. Re-architecting i/o caches for emerging fast storage devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 542–555, 2023.

[5] Mohammadamin Ajdari, Patrick Raaf, Mostafa Kishani, Reza Salkhordeh, Hossein Asadi, and André Brinkmann. An enterprise-grade open-source data reduction architecture for all-flash storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–27, 2022.

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

[7] Haiwei Deng, Ranhao Jia, and Chentao Wu. A graph-assisted out-of-place update scheme for erasure coded storage systems. In *International Conference on Parallel Processing*, pages 1–10, 2021.

[8] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. Delayed parity update for bridging the gap between replication and erasure coding in server-based storage. In *ADMS@ VLDB*, pages 1–9, 2021.

[9] Zheng Gu, Jiangpeng Li, Yong Peng, Yang Liu, and Tong Zhang. Elastic RAID: implementing raid over SSDs with built-in transparent compression. In *ACM International Conference on Systems and Storage*, pages 83–93, 2023.

[10] Soojun Im and Dongkun Shin. Delayed partial parity scheme for reliable and high-performance flash memory SSD. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST), Nevada, USA, May*, pages 1–6. IEEE Computer Society, 2010.

[11] 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.

[12] 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 *USENIX Conference on File and Storage Technologies (FAST), February*, pages 355–370, 2021.

[13] Jaeho Kim, Eunjae Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability. *IEEE Trans. Computers*, 65(4):1116–1130, 2016.

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

[15] Jieun Kim, Dohyun Kim, and Youjip Won. Fair I/O scheduler for alleviating read/write interference by forced unit access in flash memory. In *ACM Workshop on Hot Topics in Storage and File Systems*, pages 86–92, 2022.

[16] Eunjae Lee, Yongseok Oh, and Donghee Lee. SSD caching to overcome small write problem of disk-based RAID in enterprise environments. In *ACM Symposium on Applied Computing, Salamanca, Spain, April*, pages 2047–2053, 2015.

[17] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. FRA: a flash-aware redundancy array of flash storage devices. In *International Conference on Hardware/Software Codesign and System Synthesis, Grenoble, France, October*, pages 163–172. ACM, 2009.

[18] Yongkun Li, Helen H. W. Chan, Patrick P. C. Lee, and Yinlong Xu. Elastic parity logging for SSD RAID arrays. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, June*, pages 49–60, 2016.

[19] Jakob Lüttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian M. Kunkel, and Thomas Ludwig. Survey of storage systems for high-performance computing. *Supercomput. Front. Innov.*, 5(1):31–58, 2018.

[20] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 165–180, 2022.

[21] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Trans. Storage*, 8(1):4:1–4:20, 2012.

[22] Linjun Mei, Dan Feng, Jianxi Chen, Lingfang Zeng, and Jingning Liu. A write-through cache method to improve small write performance of ssd-based RAID. In *International Conference on Networking, Architecture, and Storage (NAS), Shenzhen, China, August*, pages 1–6, 2017.

[23] Linjun Mei, Dan Feng, Lingfang Zeng, Jianxi Chen, and Jingning Liu. A high-performance and high-reliability RAIS5 storage architecture with adaptive stripe. In *Algorithms and Architectures for Parallel Processing. Guangzhou, China*, pages 562–577. Springer, 2018.

[24] Rino Micheloni, Alessia Marelli, and Kam Eshghi. *Inside solid state drives (SSDs)*. Springer, 2013.

[25] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces. In *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, 2007.

[26] Farzaneh Rajaei Salmasi, Hossein Asadi, and Majid GhasemiGol. Impact of stripe unit size on performance and endurance of SSD-based RAID arrays. *Scientia Iranica*, 20(6):1978–1998, 2013.

[27] Reza Salkhordeh, Kevin Kremer, Lars Nagel, Dennis Maisenbacher, Hans Holmberg, Matias Bjørling, and André Brinkmann. Constant time garbage collection in ssds. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–9. IEEE, 2021.

[28] EMC Education Services. *Information storage and management: Storing, managing, and protecting digital information in classic, virtualized, and cloud environments*. Wiley Publishing, 2012.

[29] Daniel Stodolsky, Garth A. Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *International Symposium on Computer Architecture, San Diego, USA, May*, pages 64–75. ACM, 1993.

[30] Chenlei Tang, Jiguang Wan, Yifeng Zhu, Zhiyuan Liu, Peng Xu, Fei Wu, and Changsheng Xie. RAFS: A RAID-Aware file system to reduce the parity update overhead for SSD RAID. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, March*, pages 1373–1378, 2019.

[31] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 429–444, 2022.

[32] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. StRAID: stripe-threaded architecture for parity-based raids with ultra-fast SSDs. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 915–932, 2022.

[33] Shucheng Wang, Qiang Cao, Ziyi Lu, and Jie Yao. Mlog: multi-log write buffer upon ultra-fast ssd raid. In *International Conference on Parallel Processing*, pages 1–11, 2022.

[34] John Wilkes, Richard A. Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, 1996.

[35] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *USENIX Conference on File and Storage Technologies (FAST), February*, pages 307–323, 2021.

[36] 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.

[37] Ping Xie, Zhu Yuan, and Yu Hu. Nscale: an efficient raid-6 online scaling via optimizing data migration. *The Journal of Supercomputing*, 79(3):2383–2403, 2023.

[38] 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 *International Conference on Parallel Processing (ICPP), Kyoto, Japan*. ACM, 2019.

[39] 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 *IEEE International Conference on Computer Design (ICCD)*, pages 59–66. IEEE, 2018.

[40] Lingfang Zeng, Dan Feng, Jianxi Chen, Qingsong Wei, Bharadwaj Veeravalli, and Wenguo Liu. HRAID6ML: A hybrid RAID6 storage architecture with mirrored logging. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST), April, CA, USA*, pages 1–6, 2012.

[41] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: deterministic and balanced data distribution for large disk enclosures. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 279–294, 2018.

**Maryam Karimi** received the BSc degree in computer engineering from Shahid Beheshti University (SBU), in 2014 and the MSc degree in computer architecture from the Sharif University of Technology, in 2016. She is working toward the PhD degree with the Data Storage, Networks, and Processing (DSN) Lab, Sharif University of Technology. Her current research focuses on the performance and endurance of all-flash data storage systems.

**Reza Salkhordeh** received the BSc degree in computer engineering from the Ferdowsi University of Mashhad, in 2011, the MSc degree in computer engineering from the Sharif University of Technology, in 2013, and the PhD degree from Data Storage, Networks, and Processing (DSN) Lab, SUT, in 2018. He is currently a postdoctoral researcher with Efficient Computing and Storage Group, Johannes Gutenberg University Mainz under supervision of Prof. André Brinkmann. His research interests include operating systems, solid-state drives, and data storage systems.

**André Brinkmann** received the PhD degree in electrical engineering from the Paderborn University, in 2004. He is a full professor with the Computer Science Department of Johannes Gutenberg University Mainz (JGU)(since 2011). He has been the head of the data center with the JGU from 2011 until 2021. He has been an assistant professor with the Computer Science Department, Paderborn University from 2008 to 2011. Furthermore, he has been the managing director of the Paderborn Center for Parallel Computing $PC^2$ during this time frame. His research interests focus on the application of algorithm engineering techniques in the area of data center management, cloud computing, and storage systems. He has published more than 150 papers in renowned conferences and journals and is a senior associated editor of the *ACM Transactions on Storage*.

**Hossein Asadi** (Senior Member, IEEE) received the BSc and MSc degrees in computer engineering from the Sharif University of Technology, Tehran, Iran, in 2000 and 2002, respectively, and the PhD degree in computer engineering from Northeastern University, Boston, MA, USA, in 2007. He is currently a full professor with the Department of Computer engineering, SUT. He is the founder and director of the Data Storage, Networks, and Processing (DSN) Lab. and the director of Sharif HPC Center. He was a recipient of the Distinguished Lecturer Award from SUT in 2010, the Distinguished Researcher Award and the Distinguished Research Institute Award from SUT in 2016, the Distinguished Technology Award from SUT in 2017, and the Distinguished Research Lab Award from SUT in 2019. He also received the Best Paper Award at IEEE/ACM Design, Automation, and Test in Europe (DATE) in 2019. More recently, he received Distinguished National Technology Award in 2022 by Ministry of Science & Technology. His current research interests include data storage systems, SSDs, operating systems, and HPC.