# Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays

Jaeho Kim, *Virginia Tech;* Kwanghyun Lim, *Cornell University;* Youngdon Jung and Sungjin Lee, *DGIST;* Changwoo Min, *Virginia Tech;* Sam H. Noh, *UNIST*

https://www.usenix.org/conference/atc19/presentation/kim-jaeho

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.
July 10–12, 2019 • Renton, WA, USA
ISBN 978-1-939133-03-8

# Alleviating Garbage Collection Interference through Spatial Separation in All Flash Arrays*

Jaeho Kim  Kwanghyun Lim[‡]  Young-Don Jung[†]  Sungjin Lee[†]  Changwoo Min  Sam H. Noh[⋆]

*Virginia Tech*  [‡]*Cornell University*  [†]*DGIST*  [⋆]*UNIST*

## Abstract

We present SWAN, a novel All Flash Array (AFA) management scheme. Recent flash SSDs provide high I/O bandwidth (e.g., 3-10GB/s) so the storage bandwidth can easily surpass the network bandwidth by aggregating a few SSDs. However, it is still challenging to unlock the full performance of SSDs. The main source of performance degradation is garbage collection (GC). We find that existing AFA designs are susceptible to GC at SSD-level and AFA software-level. In designing SWAN, we aim to alleviate the performance interference caused by GC at both levels. Unlike the commonly-used *temporal separation approach* that performs GC at idle time, we take a *spatial separation approach* that partitions SSDs into the front-end SSDs dedicated to serve write requests and the back-end SSDs where GC is performed. Compared to temporal separation of GC and application I/O, which is hard to be controlled by AFA software, our approach guarantees that the storage bandwidth always matches the full network performance without being interfered by AFA-level GC. Our analytical model confirms this if the size of front-end SSDs and the back-end SSDs are properly configured. We provide extensive evaluations that show SWAN is effective for a variety of workloads.

## 1 Introduction

With the advent of the IoT and big data era, the amount of data generated, manufactured, and processed is expected to grow at rates that were previously unimaginable [20, 25, 36, 44, 52]. Such explosive growth of data will impose considerable stress on storage systems in data centers. All Flash Array (AFA) storage, which solely uses an array of SSDs, seems to be a viable storage solution that is capable of satisfying such a high demand. AFA has been recently receiving a lot of attention because of its high performance, low power consumption, and high capacity per volume.

While flash SSD is relatively new and its characteristics are different from HDD, the overall architecture of an AFA system is not much different from traditional HDD-based storage servers [9, 18, 21, 30, 35, 58]. This is because, instead of architecting a new SSD-based storage server from scratch, existing HDD-based storage servers have evolved to embrace high-speed SSDs. For example, an array of SSDs inside an AFA are grouped by variants of RAID architectures (e.g., RAID4, RAID5, or Log-RAID, which is based on log-structured writing that we describe in more detail in Section 2).

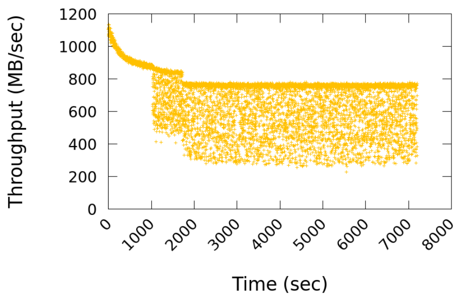


Figure 1: Performance of AFA consisting of eight SSDs under random write workloads. Performance fluctuation starts to occur (at around 1000 seconds) when the size of user write request approaches the capacity of AFA, roughly 1 TB in this configuration.

This naive AFA design, which replaces HDDs with SSDs, is not adequate to take full advantage of high-speed SSDs in two reasons. First, we observe significant performance drop as we run the FIO tool [6] that generates 8 KB random writes. Figure 1 shows the throughput of AFA with eight SATA SSDs (Samsung's 850 PRO) grouped as Log-RAID [9, 10, 21], where each SSD exhibits an effective write throughput of 140 MB/s. We find that garbage collection (GC) of AFA interfere with user I/O. Specifically, for the first 1,000 seconds, the system maintains high throughput that is close to the accumulated throughput of the eight SSDs. However, after 1,000 seconds, owing to interference with GC, the throughput drops considerably and oscillates between 300 MB/s and 750 MB/s, which is much lower than its full performance for 8 KB random I/Os.

*This work was initiated while Jaeho Kim was a postdoc at UNIST.

Table 1: Comparison of All-Flash-Array products

| | | **EMC XtremIO [1]** | **NetApp SolidFire [4]** | **HPE 3PAR [2]** | **Hynix AFA [22]** |
|---|---|---|---|---|---|
| SSD Array | Capacity | 36∼144TB | 46TB | 750TB | 552TB |
| | # of SSDs | 18∼72 | 12 | 120 (max) | 576 |
| Network | Network Ports | 4∼8×10Gb iSCSI | 2×25Gb iSCSI | 4∼12×16Gb FC | 3×Gen3 PCIe |
| | Aggr. Network Throughput | 5∼10 GB/s | 6.25 GB/s | 8∼24 GB/s | 48 GB/s |

Second, we find that storage bandwidth and network bandwidth are unbalanced. Typically, AFA is composed of multiple bricks, which is a 1U storage node to scale out capacity. As shown in Table 1, each brick is composed of a large number of SSDs and multiple network ports. Given the storage capacity and the number of installable SSDs, the aggregated write throughput of SSDs easily surpasses the aggregated network throughput. Taking EMC's XtremIO in Table 1 as an example, its 10GB/s network throughput can be easily saturated with four high-end SSDs with 2.5GB/s write throughput [48] out of the 18∼72 SSDs. This matches with observations of other recent work [12, 15, 37, 40].

The above two observations lead us to propose a new architecture for AFA systems. Given the maximum network or user-required throughput, our goal is to derive a balanced AFA system that *satisfies* the required throughput all the time without being interfered with foreground GC. To this end, we present *SWAN*, which stands for Spatial separation Within an Array of SSDs on a Network. In contrast to RAID that manages *all* the SSDs in parallel, SWAN manages the SSDs through several spatially separated groups. That is, only *some* spatially segregated SSDs out of all the SSDs, are in use at a single point in time to serve write requests over the network.

The rationale behind such segregated management is that, even if a large number of SSDs are available in the system, all the SSDs are not necessary to fully saturate the network bandwidth of the AFA. Using more SSDs in parallel does not help the clients in terms of performance. However, even with such a small number of SSDs being used, providing ideal, consistent performance is impossible with GC interference. The only way to achieve such ideal performance is hiding the GC effect. To hide the GC effect in SWAN, we partition the SSDs in the array into two *spatially* exclusive groups, that is, the front-end and the back-end SSDs, of which the front-end is composed of enough SSDs to saturate the bandwidth specification of an AFA. Then, SWAN manages these SSDs such that all writes are sequentially written to the front-end SSDs and those SSDs are never involved in GC. While the front-end SSDs are busy handling the user writes, the back-end SSDs perform GC in the background. Once the front-end SSDs become full with user data, the cleaned back-end SSDs become the new front-end to keep serving the user writes without foreground GC. By so doing, performance interference due to GC can be hidden.

This unique organization and operational behavior of SWAN gives us insight in deriving its balanced design. The number of SSDs belonging to the front-end SSDs should be large enough to fully saturate the required throughput. If not, users' performance demand cannot be satisfied. To give enough time for back-end SSDs to be completely cleaned up before they become frond-end SSDs, we should provision enough SSDs in the front-end and back-end SSD pool. Otherwise, foreground GC becomes unavoidable. The number of SSDs in the frond-end and back-end groups can be estimated by referring to the required throughput and worst-case GC cost models [31, 38, 53].

In summary, this paper makes the following contributions:

- We present a two-dimensional SSD organization as a new AFA architecture, which we refer to as SWAN. We show that such an organization allows for spatial separation of arrays of SSDs so that consistent throughput that is not influenced by GC can be provided.
- We provide an analytic model that decides the best number of SSDs in the frond-end SSD group and in the back-end SSD group. This provides guidance on deriving a balanced system when designing SWAN-based AFAs.
- We conduct comprehensive evaluations using various workloads, including both synthetic and realistic workloads, and demonstrate that SWAN outperforms existing storage management techniques such as RAID0, RAID4, RAID5, and Log-RAID [9, 10, 21]

The remainder of this paper is organized as follows. In the next section, we review existing AFA systems. Then, in Section 3, we present the design of SWAN in detail and an analytic model to completely hide the performance interference by GC. After discussing the implementation issues of SWAN in Section 4, we describe our experimental setup and present detailed results with various workloads in Section 5. We discuss the influence on SWAN on SSD design and other relevant issues in Section 6, . We review prior studies related to this work, comparing them with SWAN in Section 7. Finally, we conclude the paper with a summary in Section 8.

## 2 Background: All Flash Array

Existing AFA systems can be roughly categorized into two types depending on their write strategies, in-place write AFAs and log write AFAs, as summarized in Table 2.

Table 2: Comparison of existing approaches for managing All-Flash-Array storage

| | **Write Strategy** | **GC Interference** | **Media type** | **Modification** | **Disk Organization** |
|---|---|---|---|---|---|
| Harmonia [30] | In-place write | ● | SSDs | Array controller | RAID-0 |
| HPDA [35] | In-place write | ● | SSDs & HDDs | Host layer | RAID-4 |
| GC-Steering [58] | In-place write | ● | SSDs | Host layer | RAID-4/5 |
| SOFA [9] | Log write | ● | SSDs | Host layer | Log-RAID |
| SALSA [21] | Log write | ● | SSDs & SMR | Host layer | Log-RAID |
| Purity [10] | Log write | ● | SSDs | Host layer | Log-RAID |
| **SWAN (proposed)** | Log write | ▲ | SSDs | Host layer | 2D Array |

**GC Interference**: Interference between GC of SSD/AFA-level and user's I/O ●: Heavy interference ▲: Alleviated interference

**In-place Write AFAs.** Approaches with the in-place write strategy heavily rely on the traditional RAID architecture, but attempts have been made to improve performance by modifying data placement and the GC mechanism. The most well-known ones are Harmonia [30], HPDA [35], and GC-Steering [58].

Harmonia is based on the RAID0 architecture that groups all SSDs in an array in parallel without any parity disks [30]. To minimize high I/O fluctuation caused by SSD-level GC that independently happens in individual SSDs, it proposes a globally-coordinated GC algorithm that synchronizes the GC invocations across all the SSDs in the array. If GC is invoked in one SSD, it intentionally triggers GC in all the others, so that all the SSDs in the array are garbage collecting. This approach minimizes user-perceived I/O fluctuation through frequent GC invocations, but cannot completely eliminate performance interference by GC.

HPDA (Hybrid Parity-based Disk Array) takes an SSD-HDD hybrid architecture based on RAID4 [35]. By using a few HDDs as temporary storage to keep parity information, it mitigates performance and lifetime degradations of SSDs caused by frequent updates of parity data. While it is effective in lowering parity overhead, it does not propose any technique to hide GC interference.

GC-Steering is similar to SWAN [58]. It spatially dedicates few SSDs, called staging disks, to absorb the host writes while the other SSDs are busy performing GC. Since host writes can be served by staging disks, GC-Steering can prevent clients from being influenced by GC. However, once the staging disks become full and run out of free space to service host writes, foreground GC becomes unavoidable. To be more specific, it differs from SWAN in that it inherits the limitation of a cache. The staging space is divided into read and write regions, and hot data needs to be migrated to the space for read requests. Space constraints in the staging space not only make it impossible for all reads to avoid collision with GC, but it also causes migration overhead.

**Log Write AFAs.** While using the traditional RAID architecture (e.g., RAID4 or 5) for the purpose of fault-tolerance, some studies adopt log-structured writing, fundamentally changing its storage management policy, so as to generate sequential write requests that are more suitable for flash-based SSDs. To distinguish them from traditional ones, in this paper, we call AFA systems with log writing a log-structured RAID (Log-RAID).

SOFA is one of the first attempts to use the log-structured approach for AFAs [9]. SOFA integrates volume management, flash translation layer (FTL), and RAID logic together and runs them all in the host level. This integration enables global management of GC and wear-leveling, resulting in overhead associated with storage maintenance tasks being considerably reduced.

SALSA is similar to SOFA in that it offloads almost all of the functions that are typically performed inside storage devices to the host level [21]. However, unlike SOFA, SALSA's primary aim is in building a general-purpose storage platform that supports various types of storage media that are incapable of supporting in-place updates such as SSD and SMR.

Purity is an AFA appliance developed by Pure Storage [10]. Purity adopts log-structured indexes and data layouts that are based on the LSM-tree algorithm [43] to ensure that data is written in large sequential chunks. For better utilization of disk capacity, it also incorporates compression and deduplication algorithms into their system.

SWAN shares the same advantages of the aforementioned techniques as it runs its log-structured storage management logic at the host level. However, SWAN takes it one step further by minimizing the performance interference caused by GC while considering an AFA design that balances storage media and network interface performance.

## 3 Design of SWAN

### 3.1 Design Goal and Approach

The primary design goal of SWAN is to provide sustainable high performance for All Flash Array (AFA). More specifically, so that storage does not become the bottleneck, we aim to guarantee AFA storage performance to always be higher than or equal to the network interface bandwidth of AFA.

At a glance, this looks easy to achieve by simply using a RAID of multiple SSDs because even consumer-grade SSDs provide more than 1 GB/s bandwidth. As shown in Figure 2(a), RAID can be used to improve performance and reliability of AFA by composing multiple SSDs in parallel.

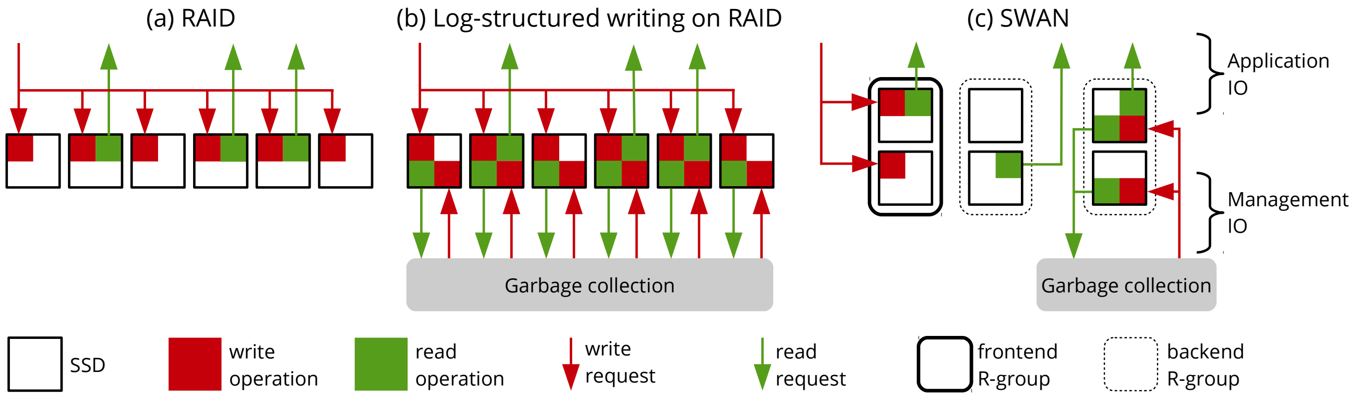


Figure 2: Comparison of All Flash Array (AFA) design. Existing AFA roughly follow either (a) RAID or (b) log-structured writing on RAID (Log-RAID) approaches. We propose (c) SWAN, a spatial separation approach to AFA.

However, its design is susceptible to gradual performance degradation due to high GC overhead inside SSDs. In fact, it turns out that such *SSD-level GC* can significantly degrade performance and incur latency spikes in AFA [30, 51].

To mitigate the internal SSD-level GC overhead, log-structured writing on RAID (Log-RAID), as depicted in Figure 2(b), has been widely adopted in AFA [9, 10, 21]. It generates SSD-friendly write requests by transforming small, random write requests to a bulk, sequential write stream thereby reducing the internal GC overhead in SSDs. However, the I/O operations for *AFA-level GC* (not SSD-level) may significantly interfere with application I/O and degrade performance by constantly issuing read/write requests. In particular, if an application tries to read a sector on an SSD where AFA-level GC is in progress, read latency could increase by several orders of magnitude [21, 23, 51]. A common approach to mitigate such interference is to perform AFA-level GC at idle time by *temporally separating* application I/O and AFA-level GC I/O, that is, segregate the two I/Os in terms of time. However, temporal separation of application I/O and AFA-level GC I/O is hard to control in reality because high performance AFAs are designed to handle multiple concurrent clients.

The key idea of our approach is the *spatial separation* of application I/O and AFA-level GC I/O to minimize such interference by organizing the SSDs into a two-dimensional array as depicted in Figure 2(c). Like Log-RAID, we adopt log-structured writing on RAID to minimize the performance degradation caused by heavy SSD-level GC while providing high performance and reliability using RAID. However, unlike Log-RAID, we spatially separate SSDs into two pools, the front-end and back-end pools. The front-end pool will serve write requests from applications in a log-structured manner, while the back-end pool is used for *SWAN-level GC*, which is GC that occurs only at the back-end pool. Thus, SWAN-level GC does not interfere with application write requests. When the front-end pool SSDs become full, a pool of SSDs from the back-end becomes the front-end and the old front-end SSDs return to the back-end. This design also has the advantage that application read requests are less interfered by SWAN-level GC operations. Recall that as SWAN uses commodity SSDs, it does not have direct control over SSD-level GC. However, we take a best effort approach given the conventional SSD interface.

## 3.2 Flash Array Organization

SWAN exposes linear 4KB logical blocks such that upper-level software just considers SWAN as a large block device. The SWAN software module is implemented as a part of the Linux kernel in the block I/O layer, where a logical volume manager or a software RAID layer is implemented. SWAN groups multiple SSDs into a single physical volume, which is then divided into fixed-size segments. It manages each segment in a log-structured manner, with new data always being sequentially appended. A segment is the unit for writing a chunk of data as well as for cleaning of obsolete data. Similar to other log-structured systems, therefore, SWAN manages mapping between logical blocks and segments, and, when necessary, it performs GC to secure free segments.

The overall architecture of SWAN is like a big host-level FTL supporting multiple SSDs, but it is the management mechanism in SWAN that differs from typical systems. Typical RAID systems manage an SSD array in a one-dimensional manner, that is, all the SSDs are arranged horizontally and incoming writes are evenly striped over all of them. Unlike RAID, in SWAN, an array of SSDs is organized as a *two-dimensional array*. Then, SSDs belonging to the same column are grouped in a RAID manner and are used in parallel. This group of SSDs is called a *RAID group (R-group)* as this is where redundancy is manifested to prevent data loss in the event of hardware or power failure. The R-group is also where the size can be set such that its aggregate throughput surpasses the network interface provided by AFA. That is, for AFA providing higher network interface bandwidth, we can increase the number of SSDs within the

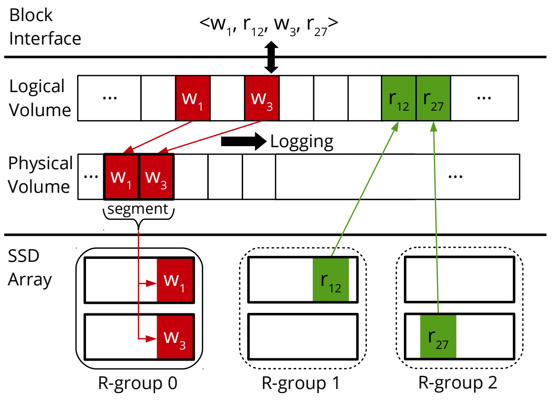


Figure 3: Example of handling read/write requests in SWAN where R-group 0 is the front-end R-group and R-groups 1 and 2 are in the back-end pool. SWAN appends writes to the log and issues write requests to the front-end R-group in segment units. Read requests will be served by any R-group holding the requested blocks.

R-group to match the network bandwidth. We further discuss optimizing the configuration of SWAN in Section 3.5.

## 3.3 Handling Application I/O Requests

As discussed, SWAN organizes SSDs into two or more R-groups, and each R-group is either a front-end or belongs to the back-end pool at a certain point in time. SWAN manages the movement of the R-groups such that each R-group takes turns being the front-end R-group, while the rest belong to the back-end pool.

Only the front-end R-group serves the incoming writes in a log-structured manner, consuming its free space. Once free space of SSDs in the R-group in the front-end pool is exhausted, SWAN moves this R-group to the back-end pool and selects a new R-group in the back-end pool that has enough free space to become the front-end R-group to serve writes coming in from the network. Incoming read requests are served by any SSD holding the requested blocks.

Figure 3 shows an example of how SWAN handles the I/O sequence $< w_1, r_{12}, w_3, r_{27} >$ arriving from the network, where $w_i$ and $r_i$ are the write and read of block $i$, respectively. The writes are appended to a segment, but are actually distributed across SSDs in the front-end R-group and are written in parallel. Reads, in contrast, will be served by any of the three R-groups. To alleviate read delays due to GC, we can employ methods such as RAIN as suggested by Yan et al. [60], which we do not consider in this study and leave for future work. However, as we show later, even without such optimizations, SWAN read performance does not suffer from delays as it is always given highest priority. Thus, read performance is comparable with conventional methods, while write performance is significantly improved.

## 3.4 Garbage Collection in SWAN

SWAN performs GC to secure free segments like other log-structured systems. SWAN chooses victim segments from one of the back-end R-groups and writes valid blocks within the chosen R-group. That is, GC is performed internally within a single back-end R-group. Also, while any victim selection policy could be used [13, 16], in this paper, we use the greedy policy that chooses a segment that has the least number of valid blocks as the victim. Such GC creates a free segment in the chosen R-group. When the front-end R-group becomes full, SWAN chooses an R-group in the back-end to be the next front-end, then moves the old front-end to the back-end group. SWAN completely decouples normal I/O from GC I/O by spatially separating SSDs into the front-end R-group and the back-end R-groups, so as to eliminate interference by GC I/O upon user writes.

## 3.5 Optimizing SWAN Configuration

The key insight behind the SWAN design is that given the many SSDs in AFA, only a portion of these SSDs (which forms the R-group) are sufficient to saturate the network interface bandwidth of an AFA. However, to realize and achieve sustainable high performance in SWAN, it is important to properly decide the configuration knobs: 1) the number of SSDs in an R-group and 2) the minimum number of R-groups in an SSD array.

**Determining the number of SSDs in an R-group.** The aggregated throughput of the SSDs in one R-group must be high enough to fully saturate the AFA network interface bandwidth. Thus, we determine the number of SSDs in an R-group such that the aggregated write throughput[1] of the SSDs in an R-group is higher than the aggregated AFA network interface bandwidth. For example, using an AFA configuration such as the EMC XtremIO in Table 1, given the aggregate network throughput of 10 GB/s and assuming the maximum write throughput of an SSD to be 2.5 GB/s [48], four SSDs (three for data and one for parity) must be assigned to the R-group to support RAID4 in SWAN.

**Determining the minimum number of R-groups.** Besides the raw aggregated throughput of an R-group, another important factor to decide the maximum write throughput of SWAN is GC overhead. If consuming a segment in a front-end R-group is faster than generating a free segment in the back-end R-groups, SWAN-level GC will be a performance bottleneck, limiting its performance. Thus, the number of the back-end R-groups should be large enough for SWAN-level GC not to fall behind. We provide an *analytic model* to calculate the minimum number of R-groups, which determines the number of back-end R-groups, to avoid cases where GC falls behind the front-end writing. In our analytic model, we only consider operations that dominate the execution time

[1] We consider only the write throughput of an SSD because read is faster than write.

such as read, write, and garbage collection and do not consider SSD-level optimizations such as the write buffer in the SSD that could further improve SSD performance.

Since SWAN manages the SSD array in a log-structured fashion, it divides the SSD space into fixed-size segments, and all the write and cleaning operations are done in segment units. Let $T_w$ and $T_r$ denote the elapsed times for writing and reading a segment to and from an SSD, respectively. Let $T_e$ be the time for erasing flash blocks in a segment when all the data in it are invalid.

SWAN writes new data over the network to a front-end R-group. Once the front-end R-group fills up, it is moved to the back-end. To perform GC for a segment that has both valid and invalid pages in the back-end, the valid pages must first be read and then written to a free segment. Of course, all flash blocks in the free segment must be erased before writing valid pages. Therefore, time to finish GC of an R-group is $S \cdot (T_e + u \cdot (T_r + T_w))$ where $S$ is the number of segments in an R-group and $u$ is the ratio of valid pages in a segment (i.e., segment utilization). After finishing GC of an R-group, we will have $S \cdot (1-u)$ free space for this R-group. That R-group will be moved to the front-end later at some time. It will take $S \cdot (1-u) \cdot T_w$ to completely consume the free segments in that R-group.

In SWAN, all R-groups are independent of each other; either they service writes and reads as the one in the front-end or they perform GC and reads as ones in the back-end. Once an R-group is moved to the back-end pool, it will not be chosen as a front-end R-group until all previous R-groups in the back-end are consumed. This implies that, after an R-group moves to the back-end pool, it will return as the front-end R-group after $(N-1) \cdot (S \cdot (1-u) \cdot T_w)$ time at the earliest, where $N$ is the number of R-groups in a SWAN array. This is when data are written to the front-end R-group SSDs at maximum throughput; it will take longer to return if the throughput is lower.

This tells us that, if the GC time of an R-group is equal to or shorter than the time that R-group is recycled, SWAN can finish GC of an R-group before moving it to the front-end. Conversely, if the above condition is not met, SSDs in the front-end R-group may need to delay writes as it waits for free segments to become available.

Consequently, the condition

$$S \cdot (T_e + u \cdot (T_r + T_w)) \leq (N-1) \cdot S \cdot (1-u) \cdot T_w$$

must hold to guarantee that SWAN-level GC does not interfere application writing at the front-end. This can be simplified as follows:

$$T_e + u \cdot (T_r + T_w) \leq (N-1) \cdot (1-u) \cdot T_w$$

From here, we get

$$\frac{T_e}{T_w} \cdot \frac{1}{1-u} + \left(\frac{T_r}{T_w} + 1\right) \cdot \frac{u}{1-u} + 1 \leq N \qquad (1)$$

Note that Equation 1 is independent of the number of SSDs per R-group and dependent on the specifications of the SSDs and the utilization. Previous studies [31, 38, 53] have shown that in a log-structured scheme, $u_d = \frac{u-1}{\ln u}$ holds, where $u_d$ is the disk utilization. This tells us that even for heavy loaded storage systems where the disk utilization ($u_d$) is 60% to 70%, $u$ will be below 0.5.

Let us now consider applying the model. Given an array of SSDs, let $T_e$, $T_w = t_w \cdot B$, and $T_r = t_r \cdot B$ be constants, where $T_e$, $t_w$, $t_r$, and $B$ are the time to erase a segment, write a block, read a block, and the number of blocks in a segment, respectively. From our AFA prototype, our measurements show that segment erase time is roughly 4 milliseconds, block read and write time is 15.6 and 19.5 microseconds, respectively, and the number of blocks per segment is 256. Taking these numbers and with a storage device that is ($u_d =$) 60% utilized, which results in roughly $u = 0.33$, then we can calculate $N$ to be roughly 1.89. This tells us that with SWAN composed of two R-groups, we will be able to sustain the full network interface bandwidth performance and see no GC affects throughout its services. We believe that the modeling results based on our measurement-based parameter estimations effectively reflects the underlying system architecture as the impact of realistic factors such as queuing delays and resource contention are being reflected in the measured parameters.

Applying these results to a realistic setting, let us, once again, take a configuration such as EMC's XtremIO in Table 1, assuming an SSD with 2.5 GB/s write bandwidth. If we can configure each R-group to be of four SSDs, which is enough to saturate the network bandwidth, then we have, in the smallest configuration case (18 SSDs), three back-end R-groups (plus two spare SSDs), which will be more than sufficient to allow full sustained write performance.

One factor that we did not consider in our analysis is the bandwidth consumed by read requests to the back-end R-group. However, in reality, as the number of back-end R-groups are sufficiently high, these reads will not have a real effect on GC time needed to return as a front-end R-group.

Our analysis shows that with our SWAN approach, once we have set the number of SSDs within the R-group to match the network bandwidth, the total number of SSDs to maintain high, sustained performance can be determined. Also, extra SSDs for larger capacity will further ensure that SWAN-level GC will not interfere user writes at the front-end.

## 4 Implementation

We implement SWAN and Log-RAID in the block I/O layer, where the I/O requests are redirected from the host to the storage devices, in Linux kernel-3.13.0. For our implementation, we extend the SSD RAID Cache implementation in the Device Mapper (DM) [42] to accommodate AFA storage. To implement RAID0, RAID4, and RAID5, we use `mdadm`, which is a GNU/Linux utility used to manage and monitor software RAID devices [54].

### 4.1 Metadata Management

SWAN and Log-RAID maintain basically the same metadata. They manage two types of metadata: 1) a mapping table from the logical volume to the physical volume mapping table (L2P) for address translation, and 2) the segment summary information. Each entry in the table, which takes up 5 bytes, corresponds to a 4 KB block in an SSD array. Thus, the metadata size for the mapping table is roughly 0.12% of the total storage capacity (i.e., 5 bytes per 4 KB).

The segment summary metadata contains information about each segment such as the segment checksum, sequence number, and the physical to logical mapping (P2L) for GC. It is located in the last block of a segment taking up 4 KB per segment. The metadata overhead for segment summary depends on SWAN's configuration. For example, for a 1 MB segment size, which is the size used in our experiments, segment summary takes up 0.39% of the storage space (i.e., 4 KB per 1 MB).

In the SWAN and Log-RAID prototypes, we maintain the entire metadata structures in DRAM, assuming that their contents are backed up by built-in batteries in the server. Owing to their huge size, however, keeping all of the data structures in DRAM could be burdensome, in terms of cost and energy. To address this, on-demand mapping that only keeps popular mapping entries in DRAM while storing the rest in SSDs can be considered. However, we do not consider this in this study.

### 4.2 Optimizing GC using TRIM

TRIM is used to further optimize GC. Once valid pages in a victim segment are written back to the new segment, then the victim segment is TRIMmed. This is efficient as the writing of the segments occur in a sequential manner and also, as the TRIM unit is large. With large segments being TRIMmed, the SSD firmware will perform erasures in an efficient manner. Thus, it helps SWAN achieve high performance regardless of SSD manufacturer.

## 5 Evaluation

In this section, we first present the evaluation results of micro-benchmarks to see how our design choices affect the behavior of SWAN and help to avoid GC interference. We then present the evaluation results of real-world workloads and compare the performance of SWAN to the traditional RAID0, RAID4, RAID5, and Log-structured management schemes (Log-RAID0 and 4) for an array of SSDs.

We evaluate SWAN on a Dell R730 server equipped with two Xeon E5-2609 CPUs and 64GB DRAM. We use 120GB Samsung 850 PRO SSDs of which measured peak read and write throughput is roughly 500MB/s and 400MB/s, respectively. The number of SSDs used differ from experiment to experiment as we describe later. We measure performance at the host system. Before any experiments for a particular configuration, we go through a systematic cleaning and aging process; each SSD is first cleaned through formatting and TRIMming, and then the SSD is aged by making random writes to roughly 60% of the storage capacity.

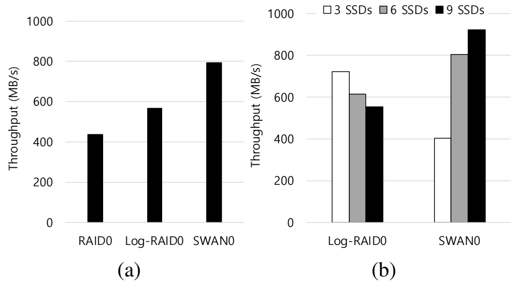


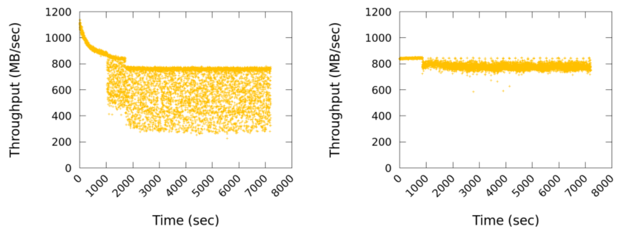
Figure 4: (a) Performance comparison of RAID0, Log-RAID0 and SWAN0 with 8 SSDs with SWAN0 configured as 4R-2SSD and (b) performance trend for Log-RAID0 and SWAN0 with 3, 6, 9 SSDs with SWAN configured as 3 R-groups with 1, 2, and 3 SSDs.

### 5.1 Micro-benchmarks

We compare the performance of SWAN with two other AFA schemes, RAID and Log-RAID, all with RAID0 configurations. We denote each of these configurations as SWAN0, RAID0, and Log-RAID0, and the convention of attaching the suffix number representing the RAID type to the configuration will be used throughout hereafter. To understand their behavior especially under heavy GC, we make use of the FIO [6] benchmark issuing 8 KB random write (only) requests. To observe the raw performance, we disable any caching layers and directly issue writes to each scheme. Each experiment is conducted for two hours and its total footprint is roughly 12 TBs. Each experiment is performed more than 3 times and all results are within 6% of each other.

Figure 4(a) shows the results with 8 SSDs and SWAN configured as 4 R-groups of 2 SSDs each, which we denote as 4R-2SSD. Hereafter, this numbering convention will be used to represent SWAN configurations. The results show that RAID0 shows worst performance because it generates random writes and incurs high GC overhead inside the SSD. While Log-RAID0 transforms random writes to bulk, sequential writes, its performance is slower than SWAN0. The reason is Log-RAID0 requires GC to reuse log space, which issues read and write operations to all SSDs as illustrated in Figure 2. These GC related operations significantly degrade normal I/O operations. In contrast, SWAN0 shows close to full SSD throughput.

We then compare the performance of Log-RAID0 and SWAN0 with varying number of SSDs to understand how our partial aggregation of SSDs affects performance. We make use of 3, 6, 9 SSDs to configure Log-RAID0 and SWAN0, which, in turn, is configured as 3R-1SSD, 3R-2SSDs, and 3R-3SSDs using 3, 6, and 9 SSDs, respectively. As Figure 4(b) shows, surprisingly, the throughput of Log-



(a) Log-RAID0 (8-SSD) (b) SWAN0 (4R-2SSD)

Figure 5: Throughput of Log-RAID0 and SWAN0 over time

RAID0 degrades as more SSDs are used, while performance of SWAN0 improves. The reason behind this can be explained by the results shown in Figure 5, which shows (a) 8 SSDs configured as Log-RAID0 and (b) SWAN0 as 4R-2SSD. The results in this figure show that even with only 2 SSDs for SWAN, performance is actually better than Log-RAID0 with 8 SSDs. Furthermore, it shows that the throughput of Log-RAID0 fluctuates significantly, while SWAN0 shows high, sustained write performance that is proportional to the 2 SSDs in the R-group. The reason for throughput degradation with Log-RAID is that normal I/O and GC I/O interfere with each other. When normal I/O and GC I/O requests are being served by the same SSD, the latency of each I/O operation increases. As more SSDs get involved, the throughput of Log-RAID0 degrades because GC performance is limited by the slowest SSD [18].

## 5.2 Analysis of GC Behavior

We further analyze the performance degradation caused by AFA-level GC. For the random write workload of Figure 5, we plot the read/write throughput of each SSD for Log-RAID0 and SWAN0 in Figure 6. Note that all reads here are those issued for GC, and thus, we can observe the negative effect on overall performance due to such read operations.

Figure 6(a) shows the results for Log-RAID0. We see that all SSDs are involved in write operations throughout its execution. GC operations of Log-RAID begins from ⓐ in the figure, and performance starts to fluctuate from that point. This is also the point where read maintains a steady bandwidth overhead. (Though there is a line for read before this point, the value is 0.) The total amount of write requests up to this point closely coincides with the total RAID capacity. Once disk space becomes exhausted, GCs are triggered, and we observe performance deterioration and fluctuation. This observation lasts until the end of our experiment.

Figure 6(b) shows the performance for SWAN0. Here, in contrast to those of Figure 6(a), we see that the throughput of SSDs 1 and 2 that comprise the front-end R-group is close to 400MB/s, the maximum throughput of the SSD, as they are the ones receiving the write requests. The performance offered to the user is the aggregate of the two SSDs, which is roughly 800MB/s. Once the free segments in the first two SSDs are exhausted, SSD 3 and 4 become the front-end R-group and the old front-end R-group becomes the back-end. When SSD 7 and 8 become the front-end R-group, SWAN starts performing GC by selecting the SSDs with victim segments based on the greedy policy. In the figure, the R-group denoted by ⓑ is the front-end and the one denoted ⓒ is the selected back-end R-group that is performing GC. Note from ⓒ (and the magnified circle) that only SSDs performing garbage collection is incurring reads. All other SSDs neither incur reads or writes (except the ones of the front-end R-group). These front-end and back-end transitions are repeated throughout the experiments.

To quantitatively understand how SWAN GC behaves in runtime, we analyze the utilization of victim segments (i.e., the ratio of valid pages in a victim segment) and the number of free segments in the SWAN array for 80 minutes. Recall that as our workload continuously writes to the front-end R-group and GC on the back-end must continuously be performed in the background to maintain stable performance. From Figure 6(b), we observe that starting from around 800 seconds, GC starts to occur. Figure 7 shows that initially the utilization of the selected victim segments are 0, but then start to increase. The results show that, eventually, the utilization of the victim segment and the number of free segments are converging. This is because data is being overwritten and thus, the back-end R-group is likely to have many obsolete data. Such convergence shows that free segment generation through GC in SWAN is stable and does not interfere with the writes occurring in the front-end R-group.

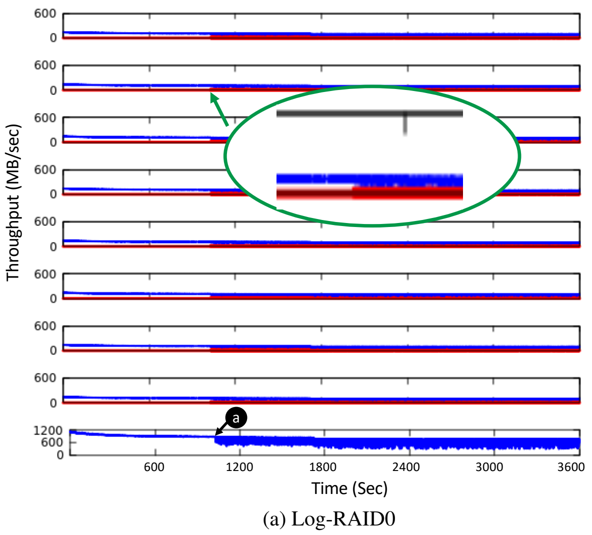
## 5.3 Real-world Workload

To see how effective SWAN is in a real-world setting, we experiment with the YCSB benchmark [11] on RocksDB [5]. For these experiments, we use RAID4 and 5 configurations, which is different from previous sections, to test SWAN in a more realistic setting. In particular, we use 9 SSDs with RAID4, RAID5, Log-RAID4, and SWAN4, which is configured as 3R-3SSD with 2 data SSDs and 1 parity SSD per R-group.

Table 3: I/O characteristics of YCSB benchmark

| YCSB | Load | Run | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| Read | - | 32GB | 60GB | 64GB | 60GB |
| Update | - | 32GB | 3GB | - | - |
| Insert | 64GB | - | - | - | 3GB |
| R:W ratio | 0:100 | 50:50 | 95:5 | 100:0 | 95:5 |

The workload characteristics of YCSB is summarized in Table 3, which includes the amount of data accessed by three different operations, Read, Update, and Insert, as well as the read/write ratios. Note that all the YCSB benchmarks consist of two phases: load and run phases. The load phase is responsible for creating the entire data set in the database, thus involves a large number of Insert operations. The run phase



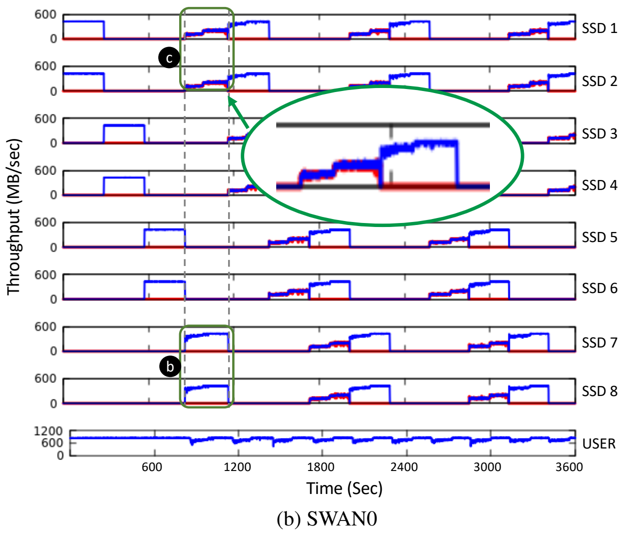
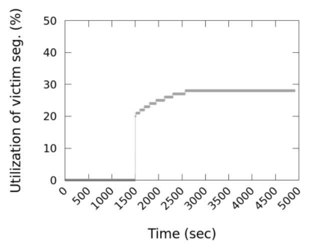
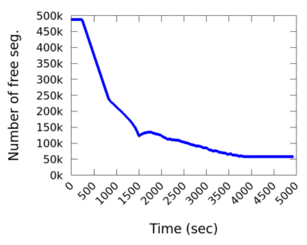


Figure 6: Throughput of Log-RAID0 and SWAN0 for random write workload used in Figure 5. Top eight rows are the write throughput for each SSD and they include not only user requests but also GC incurred by each scheme. The bottom row shows the aggregate throughput of each scheme. The blue (upper) line denotes write throughput and the red (lower) line denotes the read incurred by GC. The SWAN configuration here is the same as that of Figure 5.



(a) Utilization of a victim segment



(b) Number of free segments

Figure 7: Utilization of a victim segment and the number of free segments for SWAN0 with 8KB size random write workload for 80 minutes

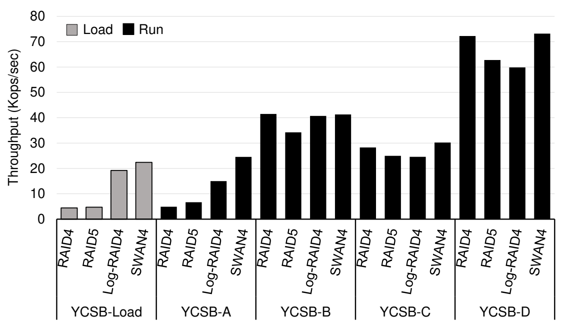


Figure 8: Throughput comparison for RAID4, RAID5, Log-RAID4, and SWAN4 for YCSB benchmark.

executes a specific workload (YCSB-A through YCSB-D) with different I/O patterns on the created data set.

**Overall Performance:** Figure 8 shows the overall throughput results. The results show that SWAN4 outperforms RAID4, RAID5, and Log-RAID for all the workloads. In the Load phase where almost all of the requests are writes, SWAN exhibits over 4× higher throughput compared to RAID-4/5 and even performs 17% better than Log-RAID. This is due to the fact that SWAN4 maintains sufficient free space to serve incoming writes immediately without interference by GC. In particular, for the YCSB-A workload where the workload is composed of reads and updates, SWAN4 performs significantly better than the other schemes, including Log-RAID4. Even in the other workloads, which are read dominant, SWAN4 performs slightly better or similar compared to the other schemes. As read is more latency-sensitive, we now further analyze read latency.

**Read Latency:** We now consider the effect of SWAN on read latency. As shown in Figure 9, the average read latency of SWAN is similar to or better than the other schemes. Moreover, as illustrated in Figure 10, SWAN exhibits much shorter tail latency compared to others across all of the YCSB benchmarks. This is because, in RAID4/5 and Log-RAID4, read requests are often blocked by AFA-level or SSD-level GC. In particular, we find that even under read-dominant workloads (YCSB-C and YCSB-D), SWAN4 exhibits shorter read latency. The reason for this is due to background GC. More specifically, recall that in all our experiments we include a systematic cleaning and aging process for the array of SSDs. We find that AFA-level GC (for Log-RAID4 and SWAN4) continues for a considerable length of time (roughly 15 minutes), while SSD-level GC [55] contin-

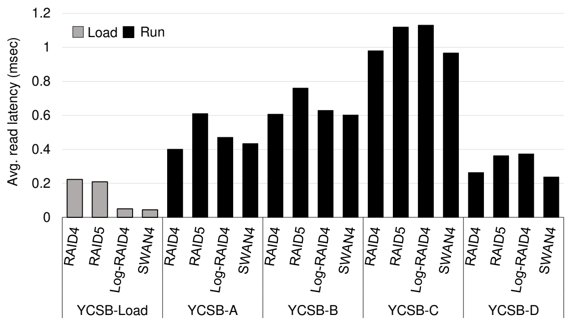


Figure 9: Average read latency comparison for RAID4, RAID5, Log-RAID4 and SWAN4 for YCSB benchmark

Table 4: Read requests in SWAN for YCSB-A workload

| | # of requests (million) | Avg. latency (usec) |
|---|---|---|
| Front-end | 20.3 | 98 |
| Back-end (Idle) | 25.9 | 88 |
| Back-end (GC) | 3.93 | 103 |

ues even further, which interfere with read requests. Fortunately, SWAN spatially separates GC so that it occurs only in one R-group, which enables us to effectively hide interference by GC. This argument is supported by Figure 11, which depicts the latency distribution of read requests for YCSB-C. Unlike the other schemes where we observe high latency spikes, SWAN shows fairly stable and consistent read latency.

To further understand SWAN's impact on read latency in more detail, we measure the latency of reads served by three different types of R-groups in SWAN: the frond-end, the idle back-end (that does not perform GC), and the busy back-end (that is performing GC), as shown in Table 4. As expected, the idle back-end provides the shortest read latency. The frond-end, on the other hand, is responsible for handling user writes, and thus it provides longer read latency than the idle back-end. Finally, the read latency on the busy back-end shows worst performance as it is more likely to be delayed by the erase and the write operations incurred by GC.

Table 4 also shows the number of reads handled by the three R-groups with the YCSB-A workload. We observe that 92% of the read requests are serviced by the idle back-end (52%) and the front-end (40%). Only 8% of the reads are destined for the busy back-end group that is performing GC. This skewed data access is due to YCSB's I/O pattern model that is based on the Zipf distribution, which is typically observed in many data-center applications [8, 14]. We find that, under the Zipf distribution with temporal locality, the front-end and the idle back-end R-groups are likely to receive more reads because they hold recently written data as we delay GC of R-groups as much as possible (as depicted in Figure 6). The frond-end receives a relatively smaller number of read requests than the idle back-end because many read requests are hit and served by the OS page cache holding data that were recently written but have not yet been evicted to the front-end R-group. The busy back-end contains old data, so only few read requests are directed to that R-group.

## 5.4 Analysis with an open-channel SSD

In AFA systems with RAID, I/O requests can be unexpectedly delayed if SSD-level GC is triggered. In particular, GC-blocked read I/Os are considered to be the root cause of long tail latency [60]. Unlike existing AFA systems, SWAN suffers less from SSD-level GC because it writes all the data in an append-only manner, thereby avoiding valid page copies for GC inside an SSD.

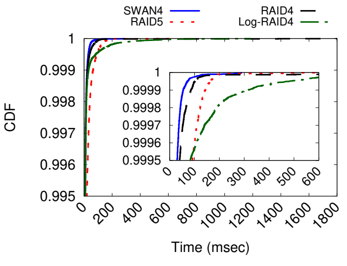
In this section, we quantitatively analyze the benefits of SWAN on individual SSDs, in terms of tail latency. Since we cannot modify and analyze the internals of off-the-shelf SSDs, we implement a custom page-level FTL scheme on an open-channel SSD [34]. From two different settings, SWAN0 and RAID0 with six SSDs, we collect block I/O traces of FIO random read/write workloads, and then replay the traces atop our open-channel SSD. We integrate a performance profiler with the custom FTL and monitor and collect detailed FTL activity statistics including page reads/writes, block erasures, as well as elapsed times for serving host reads and writes.

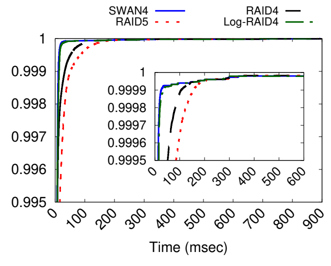
Figure 12 depicts the latency CDF measured in the open-channel SSD. The read and write latencies of NAND chips in the open-channel SSD is around $100us$ and $500us$, respectively. SWAN0 shows shorter latency and shorter tail compared to RAID0 throughout its execution. This indicates that I/O performance of RAID0 is deteriorated by the extra page copies for internal GC. Consequently, the results confirms that SWAN is effective in reducing SSD-level GC overhead.

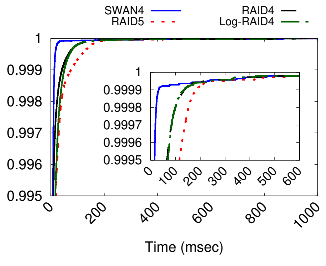
## 6 Discussion

**Benefits with simpler SSDs:** The main design principle of SWAN is minimizing the performance interference caused by SSD-level GC and AFA-level GC. We think this opens opportunities to save cost and power consumption without compromising performance by adopting SSDs with simpler design. We expect that the main benefits of the simpler SSD will come from 1) smaller DRAM size, 2) FTL implemented on a low-power ARM core or hardware, and 3) smaller over-provisioning space (OPS).

A high-end modern SSD today requires an FTL (SSD firmware), a large amount of DRAM (e.g., 0.5-16 GB for mapping tables [45, 49]), high-end processors to run its space management and garbage collection algorithm (e.g., multi-core ARM processor [19]) along with additional over-provisioning space (OPS) (e.g., extra 6.7% to 28% of flash capacity just for OPS [46, 47]) to reduce garbage collection overhead. However, SWAN does not rely on such a sophisticated, powerful FTL. SWAN sequentially writes data to segments and TRIMs a large chunk of data in the same segment at once. This implies that an SSD receives sequential write


(a) YCSB-A read


(b) YCSB-B read


(c) YCSB-C read

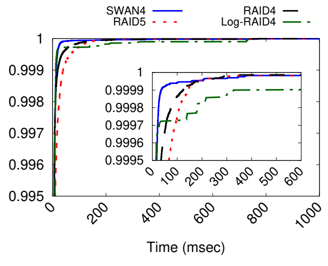
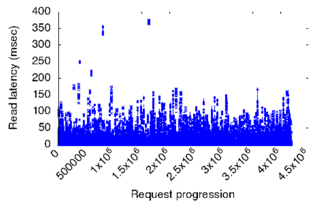
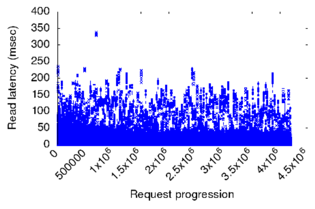
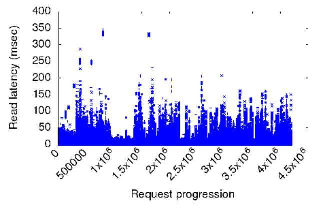

(d) YCSB-D read

Figure 10: CDF of read latency for YCSB benchmark. The tail latency of SWAN is shortest in all workloads. In particular, at 99.9th or higher latency, SWAN shows much shorter latency than others.


(a) RAID4


(b) RAID5


(c) Log-RAID4

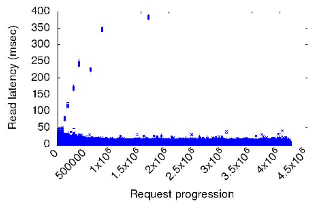
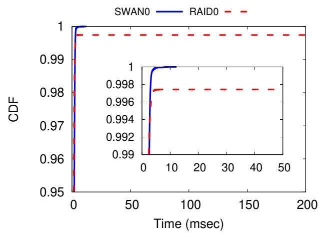

(d) SWAN4

Figure 11: Read latency distribution of YCSB-C


(a) Read

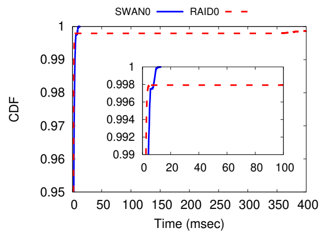

(b) Write

Figure 12: Latency CDF of FIO random read/write workloads measured in open-channel SSD for RAID0 and SWAN0

streams all the time from the host, which will be obsolete together later. Under such workloads, it is only necessary for an SSD to carry out block erasures to reclaim fully invalidated flash blocks, and thus complicated media management algorithms like address remapping and garbage collection are not needed. On the SSD side, actually, a simple block-level FTL is sufficient to support SWAN's workloads. By making the design of FTL simpler, we can reduce cost for DRAM and the processor inside the SSD and save power consumption as well. For example, a page-level FTL scheme requires roughly 1 GB of memory for a 1 TB SSD to manage the mapping information [45]. However, in our experience of implementing the block-level FTL for SWAN, only 8 MB of DRAM is required for address mapping. Also, for SSDs deployed with SWAN, they do not require a powerful processor to run sophisticated FTL algorithms such as hot-cold separation and multi-streamed I/O management [27] that are designed to reduce GC overhead. We expect a single low-power ARM core or even hardware logic to be enough to manage NAND flash with SWAN. Finally, SSDs used in SWAN do not need to reserve large OPS, which is critical to reduce GC overhead. This has the benefit of improving the effective storage capacity provided to users.

**Effect of NVRAM:** As a remedy for GC overhead, one might argue that NVRAM could be used as a write buffer to accommodate incoming writes while the underlying SSDs are busy doing GC. Using NVRAM, however, is costly because it requires expensive battery-backed DRAM. Thus, even high-end AFA controllers have only few GBs of NVRAM (e.g., 8-64 GB [3]) and use it to improve data persistence and consistency, for example, by keeping user data for a few seconds before a new consistency point starts [56]. However, considering various factors such as cost of NVRAM, the high bandwidth of the AFA network, the ever-increasing working-set size, as well as the limited DIMM slots, we think using NVRAM to buffer large amounts of user data for an extended period of time to maintain high throughput and hide SSD-level GC is not an easily acceptable solution.

## 7 Related Work

**Reducing GC overhead:** Considerable effort have been made at various layers in the storage stack to reduce GC overhead in flash storage, including at the file system [32, 39, 41], the I/O scheduler [24, 29], buffer replacement [26], and the SSD itself [17, 27, 28]. These efforts, likewise, alleviate GC overhead through reduced write amplification, but do not completely remove or hide GC overhead.

The recently proposed TTFlash almost eliminates GC overhead by exploiting a combination of current SSD technologies including a powerful flash controller, the Redundant Array of Independent NAND (RAIN) scheme, and the large RAM buffer in SSD internals [60]. By making use of the timely technologies in SSD, Tiny-tail handles I/O requests with almost no-GC scenario, with the caveat that the copy-back operation must be supported. While Tiny-tail is an SSD internal approach, it is different from what we propose as, first, we target an array of SSDs and second, we can make use of any commodity SSD, though a SWAN optimized, simpler SSDs would be most efficient.

**GC preemption:** Preemption is another way to decouple GC impact from user requests. GC preemption is a means of virtually postponing GC to avoid conflicts between GC and user requests. A number of studies, including an industry standard, have been conducted in this direction [33, 57, 59]. However, GC preemption is prone to failure for various reasons such as excessive write requests or ill-chosen GC policies [60].

**Array of Flash/SSDs:** There have been studies to address GC impact in arrays of SSDs. Flash on Rails [51] and Harmonia [30] are SSD-based array schemes suggested to resolve the GC problem. Flash on Rails separates read and write requests on different physical disks to separate the read request handling SSD from the GC handling SSD. This is a similar approach as our work, with the difference being that we consider a large scale, network connected storage system while Flash on Rails maintain at least one replica SSD for servicing read requests. It basically differs from SWAN in physical data placement and redundancy level. In large capacity storage devices such as an AFA system, this doubling of space is subject to deployment constraints. In contrast, in Harmonia, the host OS synchronizes the GC of all SSDs to prevent request blocking from unaligned GC for an array of SSDs. This approach does not remove or hide GC, but synchronizes GC to reduce its negative effect.

**Gecko:** The work most similar to ours is Gecko, which was designed for an array of HDDs [50]. Gecko is similar to SWAN in that it views the chain of HDDs as a log with new writes being made to the tail of the log to reduce disk contention by GC. SWAN advances this idea especially in the context of AFA, which is SSD-based. The key differences between Gecko and SWAN in terms of storage media can be summarized are as follows. 1) SWAN provides a guide to organizing of an array of SSDs based on the analytical model that reflects the characteristics of commercial SSD devices. 2) SWAN introduce the most efficient way to use SSDs in AFA through writing large amount of data sequentially and trimming, which is an SSD-only feature. 3) GC preemption is employed for serving read requests. 4) SWAN provides implications for a cost effective SSD design for AFA.

In terms of system organization, unlike Gecko, which uses a one-dimensional array of HDDs, SWAN manages SSDs in two dimensions to spatially separate GC writes from first-class writes and to achieve higher aggregated storage throughput than the network throughput. Also, Gecko has to prevent interference by read operations because it targets HDDs, where a read operation can also move the disk head.

**Exposing flash to host:** LightNVM [7] and Application-managed Flash [34] attempt to eliminate GC overhead by letting the host software manage the exposed flash channel. These approaches are similar to our method in that GC is being managed by the host, but they are different in that they do not decouple the I/Os for GC and those requested by user applications. Hence, even though these approaches reduce GC impact by directly controlling the flash devices from the host, GC is required in managing the flash device. SWAN, on the other hand, hides GC overhead through host controlled GC in an array of SSDs.

## 8 Conclusion

We presented a novel all flash array management scheme, named SWAN (Spatial separation Within an Array of SSDs on a Network). Our work was motivated by key observations that aggregating a number of SSDs is sufficient to surpass the network bandwidth. However, burdensome garbage collection together with all flash array software prevented us from realizing optimal performance by making it difficult to fully saturate the peak network bandwidth. In an attempt to overcome this problem, SWAN decoupled GC I/Os from normal ones by partitioning the SSD array into two mutually exclusive groups and by using them for different purposes in a serial manner: 1) serving incoming writes or 2) performing GC in the background. This spatial separation of SSDs enabled us to hide costly GC overheads, providing GC free performance to the applications. Moreover, using an analytical model, we confirmed that SWAN guaranteed no GC interference I/Os at all times if two mutually exclusive groups were properly partitioned. Our evaluation results showed that SWAN offered consistent I/O throughput at close to the maximum network bandwidth and that read latency also improved.

## Acknowledgment

We would like to thank our shepherd Patrick P. C. Lee and the anonymous reviewers for their constructive comments. This work was supported by in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2019R1A2C2009476), by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1E1A1A01077410). Sam H. Noh is the corresponding author.

## References

[1] EMC XtremIO X2 Specification. https://www.dellemc.com/resources/en-us/asset/data-sheets/products/storage-2/h16094-xtremio-x2-specification-sheet-ss.pdf.

[2] HPE 3PAR StoreServ Specification. https://h20195.www2.hpe.com/V2/GetDocument.aspx?docname=4AA3-2542ENW.

[3] NetApp All Flash FAS. https://goo.gl/1D9dmT.

[4] NetApp SolidFire Specification. `https://www.netapp.com/us/media/ds-3773.pdf`.

[5] RocksDB: A persistent key-value store. https://rocksdb.org/.

[6] Axboe, J. FIO: Flexible I/O Tester. `https://github.com/axboe/fio`.

[7] BjÃŽrling, M., Gonzalez, J., and Bonnet, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2017), pp. 339–353.

[8] Chen, Y., Alspaugh, S., and Katz, R. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment 5*, 12 (Aug. 2012), 1802–1813.

[9] Chiueh, T.-c., Tsao, W., Sun, H.-C., Chien, T.-F., Chang, A.-N., and Chen, C.-D. Software Orchestrated Flash Array. In *Proceedings of International Conference on Systems and Storage (SYSTOR)* (2014), pp. 14:1–14:11.

[10] Colgrove, J., Davis, J. D., Hayes, J., Miller, E. L., Sandvig, C., Sears, R., Tamches, A., Vachharajani, N., and Wang, F. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2015), pp. 1683–1694.

[11] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2010), pp. 143–154.

[12] Davis, R. The Network is the New Storage Bottleneck. `https://www.datanami.com/2016/11/10/network-new-storage-bottleneck/`, 2016.

[13] Desnoyers, P. Analytic Models of SSD Write Performance. *ACM Transactions on Storage 10*, 2 (Mar. 2014), 8:1–8:25.

[14] Di, S., Kondo, D., and Cappello, F. Characterizing Cloud Applications on a Google Data Center. In *Proceedings of International Conference on Parallel Processing (ICPP)* (2013), pp. 468–473.

[15] Edsall, T., Kaser, R., Meyer, D., Sequeira, A., and Warfield, A. Networking is Fast Becoming the Bottleneck for Storage and Compute, How Do We Fix It? `https://www.onug.net/town-hall-meeting-\networking-is-fast-becoming-the-bottleneck-for-\storage-and-compute-how-do-we-fix-it/`, Open Network User Group, 2016.

[16] Gal, E., and Toledo, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Survey 37*, 2 (2005), 138–163.

[17] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: a Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 229–240.

[18] Hao, M., Soundararajan, G., Kenchammana-Hosekote, D., Chien, A. A., and Gunawi, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 263–276.

[19] Hitachi. Hitachi Accelerated Flash 2.0. `https://www.hitachivantara.com/en-us/pdf/white-paper/hitachi-white-paper-accelerated-flash-storage.pdf`.

[20] IDC. The Digital Universe of Opportunities: Rich Data and the Increasing Value of The Internet of Things. `https://www.emc.com/leadership/digital-universe/2014iview/index.htm`, 2014.

[21] Ioannou, N., Kourtis, K., and Koltsidas, I. Elevating commodity storage with the SALSA host translation layer. In *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018), pp. 277–292.

[22] Jin, Y. T., Ahn, S., and Lee, S. Performance Analysis of NVMe SSD-Based All-flash Array Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), pp. 12–21.

[23] Jung, M., Choi, W., Shalf, J., and Kandemir, M. T. Triple-A: A Non-SSD Based Autonomic All-flash Array for High Performance Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 441–454.

[24] Jung, M., Choi, W., Srikantaiah, S., Yoo, J., and Kandemir, M. T. HIOS: A Host Interface I/O Scheduler for Solid State Disks. In *Proceedings of the Annual International Symposium on Computer Architecuture (ISCA)* (2014), pp. 289–300.

[25] Kaisler, S., Armour, F., Espinosa, J. A., and Money, W. Big Data: Issues and Challenges Moving Forward. In *Proceedings of the 46th Hawaii International Conference on System Sciences (ICSS)* (2013), pp. 995–1004.

[26] Kang, D. H., Min, C., and Eom, Y. I. An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices. In *Proceeding of the 22nd IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2014), pp. 239–248.

[27] Kang, J.-U., Hyun, J., Maeng, H., and Cho, S. The Multi-streamed Solid-State Drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[28] Kim, H., and Ahn, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 16:1–16:14.

[29] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H. Disk Schedulers for Solid State Drives. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT)* (2009), pp. 295–304.

[30] Kim, Y., Oral, S., Shipman, G. M., Lee, J., Dillow, D. A., and Wang, F. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2011), pp. 1–12.

[31] Kwon, H., Kim, E., Choi, J., Lee, D., and Noh, S. H. Janus-FTL: Finding the Optimal Point on the Spectrum between Page and Block Mapping Schemes. In *Proceedings of the International Conference on Embedded Software (EMSOFT)* (2010), pp. 169–178.

[32] Lee, C., Sim, D., Hwang, J., and Cho, S. F2FS: A New File System for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[33] Lee, J., Kim, Y., Shipman, G. M., Oral, S., and Kim, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32*, 2 (2013), 247–260.

[34] Lee, S., Liu, M., Jun, S., Xu, S., Kim, J., and Arvind. Application-Managed Flash. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 339–353.

[35] MAO, B., JIANG, H., WU, S., TIAN, L., FENG, D., CHEN, J., AND ZENG, L. HPDA: A Hybrid Parity-based Disk Array for Enhanced Performance and Reliability. *ACM Transactions on Storage 8*, 1 (Feb. 2012), 4:1–4:20.

[36] MARJANI, M., NASARUDDIN, F., GANI, A., KARIM, A., HASHEM, I. A. T., SIDDIQA, A., AND YAQOOB, I. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access 5* (2017), 5247–5261.

[37] MARRIPUDI, G., AND LLKER CEBELI. How Networking Affects Flash Storage Systems. Flash memory summit 2016.

[38] MENON, J. A Performance Comparison of RAID-5 and Log-structured Arrays. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)* (1995), pp. 167–178.

[39] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2012), pp. 139–154.

[40] NANAVATI, M., SCHWARZKOPF, M., WIRES, J., AND WARFIELD, A. Non-volatile storage. *ACM Queue 13*, 9 (2015), 20:33–20:56.

[41] OH, Y., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the Annual Haifa Experimental Systems Conference (SYSTOR)* (2010), pp. 2:1–2:9.

[42] OH, Y., LEE, E., HYUN, C., CHOI, J., LEE, D., AND NOH, S. H. Enabling Cost-Effective Flash Based Caching with an Array of Commodity SSDs. In *Proceedings of the Annual Middleware Conference (Middleware)* (2015), pp. 63–74.

[43] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[44] REINSEL, D., GANTZ, J., AND RYDNING, J. Data Age 2025: The Evolution of Data to Life-Critical. `https://www.seagate.com/our-story/data-age-2025/`, 2017.

[45] SAMSUNG. 960PRO SSD Specification. `https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/`.

[46] SAMSUNG. Over-provisioning: Maximize the Lifetime and Performance of Your SSD with Small Effect to Earn More. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_845DC_04_Over-provisioning.pdf`.

[47] SAMSUNG. Samsung NVMe SSD. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/SAMSUNG_Memory_NVMe_Brochure_web.pdf`.

[48] SAMSUNG. SSD 970 EVO NVMe M.2 1TB. `https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-1tb-mz-v7e1t0bw/`.

[49] SAMSUNG. PM1633a NVMe SSD. `https://goo.gl/PkRpKf`, 2016.

[50] SHIN, J.-Y., BALAKRISHNAN, M., MARIAN, T., AND WEATHERSPOON, H. Gecko: Contention-oblivious Disk Arrays for Cloud Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 285–298.

[51] SKOURTIS, D., ACHLIOPTAS, D., WATKINS, N., MALTZAHN, C., AND BRANDT, S. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2014), pp. 463–474.

[52] THE ECONOMIST. Data is giving rise to a new economy. `https://www.economist.com/news/briefing/21721634-how-it-shaping-up-data-giving-rise-new-economy`, 2017.

[53] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 145–158.

[54] WIKIPEDIA. mdadm. `https://en.wikipedia.org/wiki/Mdadm`.

[55] WIKIPEDIA. Write amplification. `https://en.wikipedia.org/wiki/Write_amplification`.

[56] WOODS, M. Optimizing Storage Performance and Cost with Intelligent Caching (NetApp's White Paper). https://logismarketpt.cdnwm.com/ip/elred-netapp-virtual-storage-tier-optimizing-storage-performance-and-cost-with-intelligent-caching-929870.pdf, 2010.

[57] WU, G., AND HE, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2012), pp. 10–10.

[58] WU, S., ZHU, W., LIU, G., JIANG, H., AND MAO, B. GC-Aware Request Steering with Improved Performance and Reliability for SSD-Based RAIDs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), pp. 296–305.

[59] WU, W., TRAISTER, S., HUANG, J., HUTCHISON, N., AND SPROUSE, S. Pre-emptive Garbage Collection of Memory Blocks, Jan. 7 2014. US Patent 8,626,986.

[60] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-Tail Flash: Near-Performance Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2017), pp. 15–28.