SWANS: An Interdisk Wear-Leveling Strategy for RAID-0 Structured SSD Arrays

WEI WANG and TAO XIE, San Diego State University
ABHINAV SHARMA, Qualcomm Inc.

NAND flash memory–based solid state disks (SSDs) have been widely used in enterprise servers. However, flash memory has limited write endurance, as a block becomes unreliable after a finite number of program/erase cycles. Existing wear-leveling techniques are essentially intradisk data distribution schemes, as they can only even wear out across the flash medium within a single SSD. When multiple SSDs are organized in an array manner in server applications, an interdisk wear-leveling technique, which can ensure a uniform wear-out distribution across SSDs, is much needed. In this article, we propose a novel SSD-array level wear-leveling strategy called SWANS (*S*moothing *W*ear *A*cross *N* *S*SDs) for an SSD array structured in a RAID-0 format, which is frequently used in server applications. SWANS dynamically monitors and balances write distributions across SSDs in an intelligent way. Further, to evaluate its effectiveness, we build an SSD array simulator on top of a validated single SSD simulator. Next, SWANS is implemented in its array controller. Comprehensive experiments with real-world traces show that SWANS decreases the standard deviation of writes across SSDs on average by 16.7x. The gap in the total bytes written between the most written SSD and the least written SSD in an 8-SSD array shrinks at least 1.3x.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design Studies; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: NAND flash memory, solid state disk, wear-leveling, SSD array

ACM Reference Format:
Wei Wang, Tao Xie, and Abhinav Sharma. 2016. SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays. ACM Trans. Storage 12, 3, Article 10 (April 2016), 21 pages.
DOI: http://dx.doi.org/10.1145/2756555

1. INTRODUCTION

A flash memory–based solid state disk (SSD) is a data storage device that uses NAND flash memory to store persistent data [Narayanan et al. 2009]. Major components of an SSD are the flash controller, internal buffer, and an array of identical flash memory packages [Birrell et al. 2007] (Figure 1). The flash controller manages the entire SSD including error correction, interface with flash memory, and servicing host requests [Agrawal et al. 2008]. Each flash memory package consists of multiple dies [Birrell et al. 2007]. Each die contains typically four planes, each having thousands of blocks and one data register as an I/O buffer. Each block typically has 256 pages. The size of

This work was supported in part by the U.S. National Science Foundation under grant CNS(CAREER)-0845105.
Authors' addresses: W. Wang, Computational Science Research Center, San Diego State University; email: wangw8210@gmail.com; T. Xie, Computer Science Department, San Diego State University; email: txie@mail.sdsu.edu; A. Sharma, Qualcomm Inc., San Diego, CA 92121; email: asharma.sd001@gmail.com.

DOI: http://dx.doi.org/10.1145/2756555

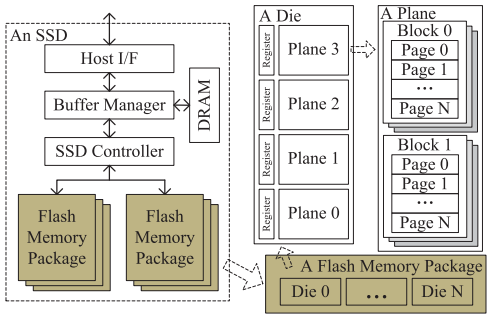


Fig. 1. SSD and flash memory organizations.

one page varies from 2KB to 16KB [Hu et al. 2011]. While read and write operations are page-oriented, erase can only be conducted at block granularity [Bez et al. 2003]. Without specification, the term "block" in the rest of this article stands for a flash block, which is 512KB when there are 128 pages with each being 4KB. Compared with traditional rotating-based HDDs, SSDs possess a number of desirable properties: much faster random access, significantly higher energy-efficiency, and enhanced reliability due to lack of mechanical moving parts [Chen et al. 2009]. Thanks to SSDs' attractive features and decreasing price, applying SSDs in server domains has received tremendous attention from both industry and academia [Narayanan et al. 2009; Balakrishnan et al. 2010; Gray and Fitzgerald 2008]. A prototype supercomputer named DASH that employs SSD arrays has been built recently [He et al. 2010].

Integrating SSDs into server-domain data-intensive applications, however, faces several challenges. One of the toughest problems is that the lifetime of flash memory is limited by the number of programming/erase operations, beyond which flash memory is no longer reliable [Cactus Technologies 2008]. Contemporary single-level cell (SLC) flash devices typically can guarantee only 100K program/erase cycles [Cactus Technologies 2008]. Multi-level cell (MLC) flash memory, even worse, normally can sustain only 5K to 10K program/erase cycles [Hu et al. 2010]. Since many server-class I/O-intensive workloads have heavy localities [Cherkasova and Gupta 2004], some blocks of flash memory could prematurely fail due to a high concentration of write cycles. Consequently, the whole flash memory becomes unreliable after it runs out of its spare blocks. Therefore, wear leveling, a technique for prolonging the service life of flash memory by managing data so that writes are distributed evenly across the blocks, was developed [Chang and Du 2009; Jiang et al. 2010; Zertal and Harrison 2011; Jung et al. 2007].

Existing wear-leveling techniques are essentially intradisk data distribution schemes as they can distribute erases and writes evenly only across the flash medium within a single SSD. Server applications, however, in addition to generating large volumes of data, often demand a high-performance and highly reliable storage system, which makes an SSD array become indispensable [Balakrishnan et al. 2010]. After analyzing several traces collected from RAID-0 employed servers such as TPC-C [Leutenegger and Dias 1993], Build [SNIA IOTTA Repository 2011], and Exchange [SNIA IOTTA Repository 2011], we discovered uneven distributions of write requests across disks. For example, in the Exchange workload, one particular SSD in a 4-SSD array receives 167.64GB data (including new write and update), which is, on average, 2.25 times larger than that of other three SSDs. Therefore, that SSD wears out 2.25 times faster than other drives in the array (see Section 5.4 for more details). Consequently, the SSD reaches the end of its lifetime earlier than other drivers; thus, it has

to be replaced earlier than its expected service life. Specifically, as no redundancy exists in a RAID-0 array, any drive failure leads to an array-level data corruption. Therefore, keeping all drivers at a similar worn-out rate and replacing them together before the end of the expected service life is a practical choice for SSD-based arrays.

One might think that the data striping technique (i.e., spreading data among the disks in a round-robin fashion) used in RAID-0 and RAID-5 can automatically prevent a skewed write distribution across disks by randomizing the disk accessed by each request. Thus, the SSD wear-out uneven problem in an SSD array does not exist. Unfortunately, earlier studies in the RAID systems reveal that workloads with small and random I/O access patterns do not benefit much from using static striping-based allocation because small requests usually cannot take advantage of parallel operations and the requests may be skewed to a small number of disks [Gray et al. 1990]. Further, in a RAID organization with parity data, such as RAID-5, the uneven write distribution problem could be even worse because each data update may cause a parity change. As a result, the disk that stores parity associated with popular data will receive a large number of writes [Mao et al. 2012]. We take our investigations on real-world workloads [SNIA IOTTA Repository 2011; Leutenegger and Dias 1993] and other researchers' study results [Gray et al. 1990; Mao et al. 2012] as clear evidence that the problem of unbalanced write distributions in RAID-structured disk arrays do exist in realistic settings.

When a noticeable variance in wear-out degree of individual SSDs organized in an array appears, current wear-leveling schemes become inadequate, as they are incapable of distributing write cycles out of the boundary of an individual SSD. In this sense, they are only a "local solution" to the wear-out-uneven problem in an SSD array. When a greatly skewed write distribution across SSDs in an array arises, SSDs-received heavy writes may reach their write cycle limitation much earlier than others. As a result, after these overly written SSDs become unreliable, the entire SSD array turns out to be undependable before the end of its designed lifetime. Therefore, we believe that a "global solution" (i.e., an interdisk wear-leveling technique at the array level), which evenly distributes writes/erasures across all SSDs, is much needed.

Hence, in this research, we design and implement an interdisk wear-leveling strategy called SWANS (*S*moothing *W*ear *A*cross *N* *S*SDs), which complements existing intradisk wear-leveling techniques to prolong SSD arrays' service life while achieving a decent performance. In this article, we concentrate on RAID-0 architecture, which has been widely used in enterprise applications such as database systems and email servers [Lee et al. 2009; SNIA IOTTA Repository 2011; Leutenegger and Dias 1993]. We leave the task of addressing array-level wear-leveling problems of other RAID structures such as RAID-5 as future work of this research. SWANS dynamically monitors the variance of write intensity across the array. It uses μ, the standard deviation of write intensity of all SSDs, to keep track of the variation of writes across the SSD array. Next, two thresholds defined as $th_{precautionary}$ and $th_{critical}$ are employed to balance write distributions among SSDs. The former is the maximum value of μ that can be tolerated by SWANS. If the value of μ lies in between the two thresholds, SWANS redirects all new write requests targeting the hottest SSD to the coldest SSD. When μ becomes greater than the latter, SWANS starts to migrate the hot data from the hottest SSD to the coldest SSD. Clearly, a smaller value of threshold results in a more even write distribution. On the other hand, more data migration processes degrades the overall performance of an array. The impact of the two thresholds is examined in Section 5.3. Unlike conventional disk load-balancing schemes that distribute loads across multiple disks to optimize resource use and maximize throughput, SWANS redistributes writes based only on the number of writes that an SSD has received. If an SSD receives a relatively small number of writes, SWANS may direct more write requests to

it even if other SSDs are idle. From this aspect, SWANS could be viewed as a load-unbalancing scheme. However, since SWANS evenly distributes writes across SSDs, it has a side-effect of overall write balancing. Since SWANS is independent of underlying SSD architecture, it can work with all existing wear-leveling techniques that reside in individual SSDs' flash controllers. In addition, SWANS can work with various types of flash memory such as self-healing NAND [Wu et al. 2011; Chen et al. 2013] to further prolong the lifetime of an SSD array. We also developed a simulation toolkit called Sim4SSD, which simulates an array of SSDs using C++. Sim4SSD borrows the skeleton of the hierarchical structure suggested by the FlashSim simulator [Kim et al. 2009] to simulate individual SSDs. Apart from that, Sim4SSD is a new simulator with its own wear-leveling, garbage collection, request scheduling, and delay-handling mechanisms. A comprehensive performance study presented in Section 5 demonstrates that SWANS can not only level wear across SSDs and, thus, enhance SSD array reliability, but also improve the performance in some cases. To the best of our knowledge, this research is the first investigation on wear-leveling problem at flash-based RAID-0 SSD arrays.

In Section 2, we discuss our motivation and related work. In Section 3, we describe the design and implementation of SWANS. The Sim4SDD simulator is presented in Section 4. In Section 5, we evaluate SWANS by using real-world traces. In Section 6, with present our conclusions and suggestions for future directions.

2. MOTIVATION AND RELATED WORK

Existing wear-leveling algorithms are implemented within the *flash translation layer* (FTL), which is a software layer running in the flash controller of an SSD [Hu et al. 2010]. The major function of the FTL is to map *logical block addresses* (LBAs) received to *physical block addresses* (PBAs) in the flash chip [Boboila and Desnoyers 2010]. Garbage collection (GC) is another function provided by an FTL. It reclaims used blocks, typically by first merging two or multiple blocks together and then erasing them. There are three types of merge operations: switch merge, partial merge, and full merge [Gupta et al. 2009]. There are a few wear-leveling mechanisms used in flash memory systems, each with varying levels of longevity enhancement [Chang and Du 2009; Jiang et al. 2010; Zertal and Harrison 2011; Jung et al. 2007; Murugan and Du 2011]. Existing wear-leveling algorithms can be generally categorized to two groups [Cactus Technologies 2008]: dynamic wear leveling and static wear leveling. Dynamic wear leveling gets its name because when a write request arrives, it dynamically selects a new free data block based on the number of erasure cycles that the block already has. It addresses the issue of repeated writes to the same blocks by redirecting new writes to different physical blocks [Zertal and Harrison 2011; Cactus Technologies 2008]. Static wear leveling redistributes all data blocks, including those that are not being written to. The most common static wear-leveling approach is to maintain a cleaning index for each block and use this information to move hot data into less-worn blocks or cold data into more-worn blocks [Jiang et al. 2010; Chang and Du 2009; Jung et al. 2007].

Existing wear-leveling schemes were built within the flash controller inside an SSD to prolong a single SSD's lifetime. However, when an SSD array is used for enterprise data-intensive applications, we observed significant variances of number of writes and merge operations received on individual SSDs. For instance, we found that, when the Microsoft Build1 trace [SNIA IOTTA Repository 2011] was running on an SSD array with 8 SSDs in our Sim4SSD simulator, the standard deviation of the number of writes that each SSD receives is as high as 12.4. In particular, the most-loaded SSD received 42% of total writes, whereas the least-loaded SSD received only 2% of total writes. In terms of number of merge operations including all three types, the standard deviation of the 8 SSDs is 19.8. Typically, SSD manufacturers characterize SSD lifetime in total

bytes written [Gasior 2013; Intel 2013]. Their predictions usually range from 20GB to 40GB per day for the length of three to five years lifespan. For example, an Intel 530 Series SSD can sustain a typical workload of 20GB of writes per day for 5 years [Intel 2013]. The written data size on the most loaded SSD is 4.14 times of the least-loaded SSD, which means that the lifetime of the most-loaded SSD is 4.14 times shorter than that of the least-loaded SSD under the Build1 workload (see Section 5).

Our observation is consistent with previous studies [Cherkasova and Gupta 2004; Gómez and Santonja 2002] on workload locality. Gómez and Santonja [2002] found that, in many applications, 80% accesses are always directed to 20% of the data, a phenomenon known as Pareto's Principle or "The 80/20 Rule." Further, after analyzing three sets of I/O traces provided by the Hewlett-Packard Labs, they discovered that some blocks are extremely hot and popular, while other blocks are rarely accessed [Gómez and Santonja 2002]. An investigation on enterprise media server workloads done by Cherkasova and Gupta [2004] also found that 14%∼30% of the files accessed on the server accounted for 92%∼94% of the bytes transferred.

Balakrishnan et al. [2010] proposed a new parity-based redundancy solution named Diff-RAID, which unevenly ages drives in an array so that the failure rates of different drives can be largely differentiated. Diff-RAID distributes parity data unevenly to prevent two or more SSDs to fail at similar times. However, we argue that the idea of Diff-RAID is unrealistic for the following reasons. First, the uneven distribution of writes leads to an imbalanced load distribution in an SSD array, which degrades the overall performance. As the authors admitted, the overall performance of an SSD array degrades when the workload imbalance among SSDs becomes obvious. This is because the drive that receives the largest number of writes quickly becomes the performance bottleneck of the whole array. The idea proposed by Diff-RAID of deliberately unbalancing the load across disks violates one of the key goals of RAID, which is largely improving performance by load balancing. Second, an uneven wear distribution also implies that SSDs in an array will be replaced at different times. Thus, newly added young SSDs and old existing SSDs coexist in an array, which increases management cost. This is because newly added young SSDs might have different models, manufacturers, and vintages, which makes it difficult for a system administrator to manage an array in which SSDs have various ages, models, manufacturers, and vintages [Pinheiro et al. 2007; Schroeder and Gibson 2007]. Typically, a disk array has tens of drives. Still, some systems can have more than 100 drives in an array. For example, the NetApp EF540 flash array storage system can have up to 120 SSDs [NetApp 2014]. It is not hard to see that managing the 120 SSDs with different lifetimes and performance incurs a very high routine management cost, which is unrealistic for an SSD array administrator. Based on our knowledge, the common practice is that homogeneous drives (no matter whether they are SSDs or HDDs) are always preferred during the lifetime of an array. Therefore, we think that a workload-balanced array (i.e., evenly distributes write requests) is more realistic. Third, the increasing unrecoverable bit error rate (UBER) of an SSD is not the only factor that causes an SSD to fail. Like other electronic devices, an SSD may suffer from a capacitor or controller IC failure [Cullen 2014; Ku 2011], which is unpredictable. Diff-RAID cannot guarantee that the SSD with the largest writes must be the first to die in an array. It is possible that, due to a hardware malfunction (e.g., a controller IC failure), an SSD fails before the workhorse SSD (i.e., the SSD with the heaviest writes). In this scenario, Diff-RAID does not work. Based on our knowledge, the common practice of disk array maintenance is that all drives in an array wear out at the same rate and will be replaced together before the end of their lifetimes (e.g., for an HDD array, a lifetime is normally 5 years). If one is really concerned that two drives will simultaneously fail, RAID-6 (which can tolerate a two-disk failure) should be adopted. Last, but not least, this research is targeting

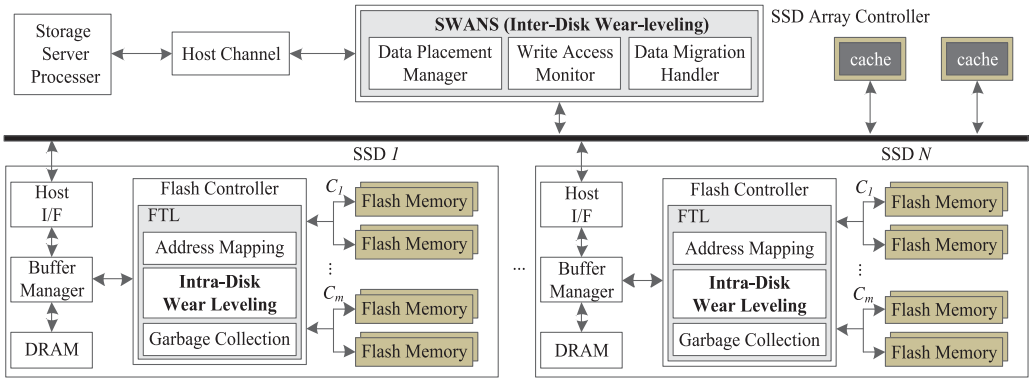


Fig. 2. The overall architecture of an SSD array and the interdisk wear-leveling technique.

RAID-0 architecture, which does not consider a drive failure during the lifetime of an SSD array. In this context, SWANS is more appropriate than Diff-RAID.

An obvious solution to the uneven wear distribution problem is to perform wear leveling at the disk array controller level. This is mainly because an interdisk wear-leveling scheme possesses a "global picture" of write-and-merge distribution among SSDs in an array. Therefore, it can redistribute write-intensive data blocks across all SSDs to balance wear-out. These important observations and analysis motivate us to propose SWANS, which will be detailed in the next section.

3. DESIGN AND IMPLEMENTATION

3.1. Architecture Overview

The overall architecture of an SSD array is depicted in Figure 2. We describe the architecture from the bottom up. There are N identical SSDs grouped together to form an SSD array. In our simulation study, we configured the N SSDs in a RAID-0 format to demonstrate our strategy. All SSDs are directly attached to the system bus through which requests/responses between individual SSDs and the SSD array controller are routed. Since high bandwidth is demanded by server-class SSDs, the multichannel architecture, as shown in Figure 2, supports up to m-way interleaving to hide flash programming latency and to increase bandwidth through parallel read/write operations, where m is the number of channels from C_1 to C_m. The FTL layer within the flash controller is responsible for several essential functions, such as address mapping, intradisk wear-leveling, and garbage collection. As we discussed in Section 2, hybrid FTLs can avoid the weakness of the two extreme mapping schemes: page-level and block-level mapping FTLs. They are widely adopted by modern SSDs [Gupta et al. 2009]. Thus, we implemented Fully-Associative Sector Translation (FAST) [Lee et al. 2007], a typical hybrid FTL algorithm, as the FTL layer for individual SSDs in our experiments. The basic idea of FAST is to make the degree of associativity between logical sectors and log blocks higher, thus achieving better write performance [Lee et al. 2007].

When a read or write request is issued from a host system, the I/O command will be decoded and processed by the flash controller (see Figure 2). Next, referenced logical addresses will be mapped to physical addresses. In the case that mapping information is changed by a write or merge operation, the mapping table needs to be updated. In order to enhance performance, an SSD normally uses a small amount of DRAM for program code, data, and buffer memory.

Table I. Basic Data Structures

Fields	Type	Size
Name: Zone Info (z_i); **Multiplicity:** No. of zones		
PhysicalSSDNumber	ulong	4B
PhysicalZoneNumber	ulong	4B
NumberOfWrites	ulong	4B
BlockWriteMap	bitmap	8B
Name: SSD Info (ssd_i); **Multiplicity:** SSD array size		
LeastPopularZone	ulong	4B
MostPopularZone	ulong	4B
NumberOfWrites	ulong	4B
Name: Migration Info (m_i); **Multiplicity:** No. of threads		
HotMigrationZone	ulong	4B
ColdMigrationZone	ulong	4B
HotSSD	short	2B
ColdSSD	short	2B

Within the SSD array controller, SWANS is implemented to perform the interdisk wear-leveling function through its three core modules: write access monitor, data placement manager, and data migration handler (see Figure 2). SWANS divides the entire logical address space of an SSD array into multiple equal-size units called zones. Typically, a zone consists of a fixed number of continuous flash blocks, with each block having a constant number of pages (e.g., 64). The write access monitor (WAM) maintains the write popularity of each SSD and zones. Essentially, SWANS realizes interdisk wear-leveling by managing data across the SSD array and the SSD array controller caches. Note that SWANS is independent of the underlying FTL schemes, including various intradisk wear-leveling techniques. The SSD array controller is connected to the storage server host through the host channel. Note that multiple SSD arrays can be connected with the storage server processor simultaneously.

3.2. Basic Data Structures

Table I summarizes three basic data structures employed by SWANS. The first one is called *Zone Info*, which maintains the information of a logical zone. Assume that the entire logical space of an SSD A with m SSDs is divided into n logical zones. It is symbolized as $A = \{z_1, \ldots, z_j, \ldots, z_n\}$, where z_i is a variable of the type of *Zone Info*, and it represents a particular logical zone. There are four fields in the *Zone Info* data structure. *PhysicalSSDNumber* stores the physical number of an SSD in array A, where the particular zone z_i is stored. The value of *PhysicalSSDNumber* is in the scope of 1 to m. Note that z_i is the logical representation of a zone; thus, it can be mapped to any physical zone *PhysicalZoneNumber* within the SSD denoted by *PhysicalSSDNumber*. The *PhysicalZoneNumber* field stores the location of a physical zone on which the logical zone z_i is mapped to. *NumberOfWrites* contains the popularity of a logical zone in terms of write. This information is utilized by SWANS to determine whether a zone is hot. Since read requests do not result in write/erasure cycles, SWANS maintains the metadata only for writes. Typically, reclaimed blocks (i.e., erased blocks) are mixed with used blocks in a migrated zone so that the *BlockWriteMap*, a bitmap, is used to record the status of a logical flash block within the zone. A set bit indicates that the logical flash block has been written at least once, therefore needs to participate in a migration process. Otherwise, the block will be skipped during data migration. The purpose of maintaining a *BlockWriteMap* map is to avoid migrating blocks that have no data. The total size of a *Zone Info* variable z_i is 20B, as each *ulong*-type field takes 4B and the bitmap type uses 8B. The number of logical zones in an array A decides the number of z_i $(1 \leq i \leq n)$ that SWANS needs to monitor. In our experiments, the

zone size is set to 16MB and the total capacity of an SSD array is fixed to 256GB (see Section 5.1). Therefore, there are 16,000 entries in the *Zone Info* table, which requires a less than 320KB array controller cache.

When the array capacity increases, a larger memory space is needed to store all the data structures in SWANS if the zone size remains unchanged. Obviously, a smaller zone size leads to a lower data migration overhead. However, it requires a larger memory space. In fact, there is a trade-off between the data-migration overhead and the memory-space overhead. As the storage system becomes more powerful, the memory space also enlarges noticeably, which relieves the memory stress. For example, in a NetApps off-the-shelf SSD array storage system, the total memory capacity can reach 24GB [NetApp 2014]. Hence, we believe that the memory consumption of SWANS is not a problem in modern SSD arrays.

The second basic data structure of SWANS is *SSD Info*, which sustains the metadata for each SSD in array *A*. While the *MostPopularZone* field stores the physical zone number of the most popular zone in write, the *LeastPopularZone* keeps the physical zone number of the least popular zone in write. The information stored in these two fields is important for SWANS to select appropriate zones in a data-migration process. Similar to *Zone Info*, *SSD Info* also keeps the number of writes for a particular SSD in the *NumberOfWrites* field. This information helps SWANS measure the "temperature" of an SSD when the hottest SSD and the coldest SSD in array *A* need to be discovered. The SSD with the largest number of writes is taken as the hottest SSD, whereas the SSD with the least number of writes is considered to be the coldest SSD. The size of ssd_i, a variable in *SSD Info* type, is 12B because each *ulong* type takes 4B (see Table I). The number of ssd_i is determined by the size of array *A*. Therefore, the total size of *SSD Info* data structures is $12m$ bytes if there are m SSDs in an array.

These two data structures are updated every time a write request comes. The CPU overhead of the updating process is low because of the following. (1) Data of each of the two data structures are organized as an array of records in a table. It is very simple to find a particular record by array indexing. (2) Updating a certain field of a record requires only one assignment or addition operation. Furthermore, the updating overhead is also a constant under different configurations (e.g., the number of SSDs) because locating a record and updating a certain field are not affected by the number of records.

The last critical data structure is *Migration Info*, which stores the information related to an active data-migration process. Its *HotMigrationZone* field stores the physical zone number of the hottest zone in terms of writes, whereas the *ColdMigrationZone* field holds the physical zone number of the coldest zone in writes. The two zones indicated by the values of these two fields are the zones that are participating in a data-migration process. The *Migration Info* data structure also contains the physical SSD number of the most popular SSD (i.e., hottest SSD) in its *HotSSD* field and the physical SSD number of the least popular SSD (i.e., coldest SSD) in its *ColdSSD* field. These two SSDs are participants in an ongoing data migration. The size of m_i, a variable of *Migration Info* type, is 12B, as each short type possesses 2 bytes. The number of variables m_i is equal to the number of data-migration threads in a migration process. Currently, Sim4SSD launches only one single-thread data migration process whenever data migration becomes necessary. Hence, there is only one variable m_i to maintain, which assumes 12B of memory.

These basic data structures have m copies if there are m SSDs. Each SSD keeps a copy of the data structures. The data structures occupy an entry block and are stored on logical block number (LBN) 1024 within a drive. The LBN in which to store the data structures can be configured in the SWANS scheme. SWANS periodically updates the newest data to all SSDs concurrently. The interval time between two successive

updates is also configurable. In our design, it is set to 5min. Having multiple copies of data structures can avoid data loss if a drive failure occurs.

3.3. The Three Core Components

Now, we present the implementation of the SWANS strategy, which is composed of three core software modules: WAM, data placement manager (DPM), and data migration handler (DMH). The WAM module dynamically maintains the Zone Info Table (ZIT) and SSD Info Table (SIT). The ZIT table stores the write popularity (hereafter, popularity) of every zone and the SIT table records the popularity of each SSD in an array. After initializing the two tables, WAM listens to every write request from the host channel and updates them accordingly. This module (see Algorithm 1) is called on the arrival of every write request. To manage the *zone info* data structures, in addition to the ZIT, SWANS employs 2 double-linked lists. One is an *empty zone list*, which stores all *zone info* data structures corresponding to all empty zones. A used zone can turn into an empty zone when its data has been migrated. When a new empty zone appears, its corresponding *zone info* data structure is added at the tail of the *empty zone list*. When an empty zone receives write requests, it becomes a used zone and its *zone info* data structure is deleted from the *empty zone list*. The second linked list is the *used zone list*, which contains all *zone info* data structures corresponding to all used zones. The *used zone list* is sorted by the number of writes that a zone has received. The *zone info* structure at the head of the list points to the zone with the largest number of writes, whereas the data structure corresponding to the least written zone is at the tail of the list.

When a write request is sent to a zone, SWANS updates the used zone list by comparing the zone's write count with its neighbor elements and than moving the zone info structure to the right position. The list is updated every time when a new write request arrives. Therefore, a small number of comparisons are performed due to the fact that the number of writes only increases by one. For a larger-size SSD array, searching the most popular zone does not incur an obviously high overhead because the *used zone list* is a sorted list.

To monitor the status of write distribution across all SSDs in an array A, the WAM module also periodically measures the value of μ, the standard deviation of write of all SSDs. The time interval between two successive measurements is defined as $t_{test-cycle}$. Its default value is called $epoch_{default}$, which is set up to 40s in our experiments. Based on the value of μ, WAM then decides which data organization method to use to make write distribution back to even. There are two thresholds for μ, $th_{precautionary}$ and $th_{critical}$. While $th_{precautionary}$ is the value of μ under which no action is needed to make write distribution even, $th_{critical}$ is the value of μ beyond which SWANS considers the SSD array to be in critical condition, and data migration is launched to restore the even distribution of writes (see Algorithm 1). In case the value of μ lies in between the two thresholds, SWANS invokes the DPM module to redirect each write request targeting an empty zone (i.e., a zone for which none of its blocks has been accessed) on the hottest SSD to an empty zone in the coldest SSD (see Algorithm 1). If no such empty-zone-oriented request is coming or there is no empty zone on the coldest SSD, DPM does nothing. Since data migration takes a long time to complete, the next write distribution testing time after a data-migration process should be enlarged to $epoch_{migration}$, which is larger than $epoch_{default}$. Similarly, the testing time interval is increased to $epoch_{placement}$ after a data placement process.

The algorithm of the DPM is depicted in Algorithm 2. After a data placement process becomes active, the DPM checks each write request to see whether its destination zone $Zone_{current}$ is empty and whether it is targeting the hottest SSD. If so, the DPM first determines the coldest SSD in the array and an empty zone $Zone_{empty}$ on it. Next, the

ALGORITHM 1: Write Access Monitor

```
Initialize the ZIT table and the SIT table
for each arrived request R_current do
    if isWriteRequest(R_current) then
        Retrieve Zone_current from R_current
        Update the popularity of Zone_current in ZIT
        Update the popularity of its SSD in SIT
        Update the used zone list and empty zone list
    end
    if t_current > t_test−cycle then // It's time for testing μ
        Calculate μ, the Std. Dev. of Writes for array A
        case μ < the precautionary // do nothing
            t_test−cycle = t_test−cycle + epoch_default
        endsw
        case th_precautionary ⩽ μ < th_critical
            t_test−cycle = t_test−cycle + epoch_placement
            Call the DPM module // launch data placement
        endsw
        case μ ⩾ th_critical
            t_test−cycle = t_test−cycle + epoch_migration
            Call the DMH module // launch data migration
        endsw
    end
end
```

ALGORITHM 2: Data Placement Manager

```
for each arrived request R_current do
    if isWriteRequest(R_current) then
        Retrieve Zone_current from R_current
        if (isEmpty(Zone_current))&(isHottestSSD(Zone_current)) then
            Find SSD_coldest, the coldest SSD in array
            Find Zone_empty, an empty zone on SSD_coldest
            SwapPhysicalZone(Zone_current, Zone_empty)
            SwapPhysicalSSD(Zone_current, Zone_empty)
        end
    end
end
```

DPM swaps the physical zone numbers between $Zone_{current}$ and $Zone_{empty}$. Last, the physical SSD numbers of the two zones are also exchanged. This way, all future write requests falling on an empty zone of the hottest SSD will be forwarded to an empty zone on the coldest SSD. As a result, the partial write load of the hottest SSD is taken by the coldest SSD. Hence, the overall distribution of writes on the array becomes more even. Since the data associated with the redirected write requests is placed on a different SSD, we name this process *data placement*. Note that data placement incurs little performance overhead as it simply redirects writes without any data movements.

The last module of SWANS is the DMH, which handles data migration between two zones when the value of μ exceeds the upper bound $th_{critical}$. Its algorithm is shown in Algorithm 3. Data migration is a process to migrate all the contents of the most popular zone of the hottest SSD to an empty zone of the coldest SSD. The DMH module has two submodules: zone grabber (*ZoneGrabber*) and zone restorer (*ZoneRestorer*). In addition, DMH utilizes a zone-size buffer for temporarily storing data fetched from the

ALGORITHM 3: Data Migration Handler

Initialize the M array with $MigrationInfo$ variables
for $i = 1; i \leqslant number_{thread}; i++$ **do**
    Determine current hottest zone hot_{zone} and its SSD hot_{SSD}
    Determine an empty zone $cold_{zone}$ and its SSD $cold_{SSD}$
    $T_i.[HotMigrationZone] = hot_{zone}$
    $T_i.[ColdMigrationZone] = cold_{zone}$
    $T_i.[HotSSD] = hot_{SSD}; T_i.[ColdSSD] = cold_{SSD};$
    Copy T_i into the element $M[i]$
end
Spawn $number_{thread}$ data migration threads
for *each thread* $thread_j$ **do**
    Allocate a buffer $buffer_{hot}$
    $ZoneGrabber(M[j].HotSSD, M[j].HotMigrationZone, buffer_{hot})$
    Wait for the zone grabber thread to complete
    $SwapZoneStats(M[j].HotMigrationZone, M[j].ColdMigrationZone)$
    $UpdateDiskStats(M[j].HotSSD, M[j].ColdSSD)$
    $SwapPhysicalZone(M[j].HotMigrationZone, M[j].ColdMigrationZone)$
    $SwapPhysicalSSD(M[j].HotMigrationZone, M[j].ColdMigrationZone)$
    $ZoneRestorer(buffer_{hot}, M[j].HotMigrationZone)$
    Wait for the zone restorer thread to complete
    Delete the buffer $buffer_{hot}$
end

hottest zone. Although we activate only one thread in a data migration process in our experiments, the DMH module is designed to be multithreaded. In the case that multiple threads are activated within a data migration process, each thread will identify one zone, then move the data from the hottest zone to the empty zone. Consequently, the total number of buffers used during a data migration process is equal to the number of concurrent migration threads ($number_{thread}$). A data structure that facilitates a data migration process is called M, which includes an array of variables in the type of *Migration Info* (see Table I). The size of M is equal to $number_{thread}$.

DMH uses the *used zone list* to find the "most popular zone" and the "least popular zone," which are always at the head and the tail of the list, respectively. When the DMH gets the most popular zone for a data-migration thread, the corresponding *zone info* structure is removed from the list. The second *zone info* structure becomes the head and points to the zone that receives the largest number of writes at this moment. In this way, the SWANS can efficiently identify the most popular zone and the next-most popular zone.

Obviously, a larger zone size incurs a heavier overhead. Assume that the zone size is set to 16MB. Also, assume that the throughput of an SSD is 540MB/s and 490MB/s for sequential read and write, respectively [Intel 2014]. Moving a zone costs 62.3 *ms* (16/540 + 16/490) without considering software overhead. In our experiments, only a small number of data migrations are triggered during the simulation time of a trace (see Table IV). Therefore, the impact of data migration on performance is limited.

DMH manages the entire migration process, which might have multiple data migration threads. It first initializes M, then spawns $number_{thread}$ data-migration threads (see Algorithm 3). For each thread $thread_j$, DMH allocates one buffer $buffer_{hot}$ from the array controller cache, which is a battery-backed RAM to prevent data loss in the event of a power failure. It then spawns one zone-grabber thread ($ZoneGrabber$) to grab the data from the hot zone (see Algorithm 3). Meanwhile, all write requests to the current migration zone are directed to the buffer located in the SSD array controller

cache. The DMH waits for the zone-grabber thread to complete. Once the zone-grabber thread finishes, the meta-information of the two zones is swapped and the statistics of the two SSDs are also updated. After information maintenance, the DMH generates a zone restorer thread (*ZoneRestorer*) to restore the data in the hot buffer to its opposite SSD, that is, hot data to the cold SSD. A working queue is used to temporarily store the requests to the zone, which is in the data-migration process. The length of the queue is 64, beyond which the incoming requests are blocked. The DMH waits until the *ZoneRestorer* thread is complete. Finally, the DMH deletes the hot buffer. It stops running until all data-migration threads terminate.

In the current design, all the parameters used by the SWANS scheme are read from a configuration file. Users can adjust the parameters, such as zone size, according to their requirements. In the next version, we will consider using Linux kernel interfaces, such as *ioctl*, to establish connections between users and the SWANS scheme. It also provides an opportunity for application software designers to build a program that can efficiently work with SWANS. Further, we assume that an enterprise server must have comprehensive power solutions, such as battery-backed RAM, to prevent data loss in power failure scenarios. Therefore, data loss due to power failure is not considered in the SWANS scheme.

4. THE SIM4SSD SIMULATOR

To evaluate the performance of SWANS, we developed a simulation toolkit for SSD arrays called Sim4SSD. Sim4SSD is built by enhancing FlashSim [Kim et al. 2009], a validated SSD device simulator. FlashSim is an event-driven simulator, written in C++ to follow the objected-oriented programming paradigm for modularity [Kim et al. 2009]. To simulate queuing effects, FlashSim has to be integrated with DiskSim [Ganger et al. 1999], a well-regarded HDD simulation environment. In particular, FlashSim uses its SSD class to provide an interface to Disksim. The SSD class creates event objects to wrap the Disksim *ioreq_event* structures and returns the event time to DiskSim [Kim et al. 2009]. In fact, FlashSim is only intended to provide a flash SSD simulation framework that allows users to create and evaluate their algorithms for critical components such as address translator, FTL, garbage collection and wear leveling. In other words, all these components are missing in the FlashSim simulator. Sim4SSD significantly enhanceS FlashSim, as follows:

(1) While FlashSim is a simulator for a single SSD device, Sim4SSD is a simulation toolkit for an SSD array. Since SWANS is present in an SSD array controller rather than in an individual SSD's controller, we implemented a new hardware component class called *SSDSimArray* to enable the simulation of an array of identical SSDs. The *SSDSimArray* class determines the internal structure of an SSD array based on an array configuration file. Currently, it supports only the RAID-0 structure. More array organization formats such as RAID-5, which consider data redundancy, are left for future work. On arrival of a new event, the *SSDSimArray* class consults either SWANS or the default distribution to redirect an event to the right SSD.
(2) Sim4SSD is a standalone simulator, thus is totally independent of DiskSim. On the contrary, FlashSim still needs to work with DiskSim in order to simulate an SSD, as it lacks the capability of simulating queuing effects. Sim4SSD implements its own queuing delay simulation.
(3) Sim4SSD largely extended FlashSim by adding its own address translator, FTL scheme (i.e., FAST [Lee et al. 2007]), garbage collection algorithm and wear-leveling strategy, which are not present in FlashSim. Totally, we added about 5,100 lines of C++ code to implement these functions plus the disk array controller and the SWANS strategy.

5. PERFORMANCE EVALUATION

In this section, we present a comprehensive study in evaluating the performance of the SWANS strategy by using Sim4SSD. We first outline the experimental setup. Next, we analyze experimental outcomes from five real-world traces.

5.1. Experimental Setup

To evaluate the performance of SWANS, we compare a SWANS-powered RAID-0 SSD array with a default RAID-0 array without any array-level wear-leveling techniques. The latter is simply called the baseline configuration in this article. Unlike SWANS, baseline simply forwards each request to its destination SSD according to the trace without any interference. The total bytes written (TBW) on an SSD has been used as an index of its remaining lifetime because it can sustain only a limited amount of written data before it fails [Gasior 2013]. In this article, the TBW of each SSD in an array is measured to demonstrate the positive impacts of SWANS on prolonging SSDs' lifetime. The number of writes that an SSD receives is an indicator of its wear-out degree, as more writes normally result in more erasures [Boboila and Desnoyers 2010]. The number of merge operations that an SSD receives can also be used as a metric of its wear-out degree, as a merge operation leads to at least one block erasure [Gupta et al. 2009]. Therefore, we use the standard deviation of individual SSDs' numbers of writes (SDW) and the standard deviation of their numbers of merges (SDM) to quantify the degree of wear-out even distribution of an SSD array. Since there is a vast gap in terms of SDW between SWANS and the baseline configuration, we take a $log2$ scale measure for SDW in all figures. We use the mean response time, the average response time of all user requests, as the performance metric of an SSD array.

The experimental system is a Dell PowerEdge 1900 server with two Quad Core Intel Xeon E5310 1.60GHz processors and 8GB FB-DIMM memory. The operating system is Linux OpenSuse 10 with kernel 2.6.16.27. We conducted experiments in a variety of SSD array configurations, including different data distribution methods (SWANS or baseline) and the changing number of SSDs in an array. For all experiments, the total capacity of an array is fixed to 256GB because the maximum footprint of all five real-world traces is no larger than 160GB. The number of SSDs in an array varies from 2 to 8, which are organized in a RAID-0 structure. Hence, the capacity of an individual SSD changes from 128GB to 32GB. As the striping unit of TPC-C is 256KB [SNIA IOTTA Repository 2011], the same striping size is used in all experiments to make comparisons fair. The main characteristics of an SSD and system parameters are shown in Table II.

For SWANS, an optimal value of zone size should be chosen. A larger zone size will lead to a higher performance overhead as data migration takes more time to complete. However, a smaller zone size will increase the SSD array control cache imprint for SWANS needs to maintain a larger Zone Info table (see Table I). In all experiments, the zone size is varied from 8MB to 32MB so that a comprehensive understanding of impacts of zone size on SSD performance and endurance can be obtained. Other important parameters are two threshold values: $th_{precautionary}$ and $th_{critical}$. To avoid frequent data migration, we set a wide gap (i.e., 10) between the two thresholds in our experiments. The epoch time is the time interval to monitor the state of an array and it is set to 40s as default.

Five real-world traces are used to evaluate SWANS: Build1 and Build2 [SNIA IOTTA Repository 2011], Exchange1 and Exchange2 [SNIA IOTTA Repository 2011], and TPC-C [Leutenegger and Dias 1993] (see Table III). Build1 and Build2 are traces generated from the activities on the Microsoft Build Server. In both traces, reads and writes are almost evenly distributed, and most of the requests are new writes. Exchange1 and

Table II. System Parameters

Description	Value
One page read (μs)	25
One page write (μs)	200
One block erase (ms)	1.5
Block size (KB)	512
Page size (KB)	4
Blocks per plane	4096
Planes per die	4
Dies per package	2, 4, 8
Package per SSD	2
Capacity of an SSD (GB)	32, 64, 128
Number of SSDs in array	8, 4, 2
Zone size (MB)	8, 16, 32
$Epoch_{default}$ (s)	40
Threshold $th_{critical}$	15
Threshold $th_{precautionary}$	5
Striping unit (KB)	256

Table III. Trace Statistics

Trace Name	Write Ratio	Ratio of Update to Write	Ave. Size (KB)
Build1	46%	39%	8
Build2	53%	31%	8
Exchange1	50%	34%	12
Exchange2	68%	65%	12
TPC-C	36%	35%	8

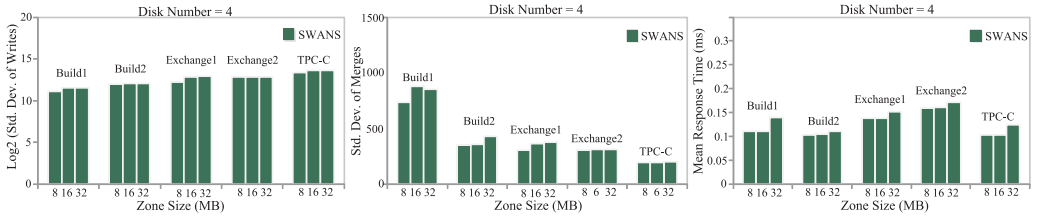


Fig. 3. Performance impacts of zone size.

Exchange2 are production traces collected at Microsoft using the event tracing for the Windows framework. These two traces are predominantly write intensive. TPC-C is an On-Line Transaction Processing (OLTP) benchmark that uses queries to update and lookup data warehouses. This trace consists of 6.15 million I/O references spread over 12.06 h. All five traces were originally collected from RAID-0 disk arrays.

We also use the Intel Open Storage Toolkit [Mesnier 2001] to generate synthetic benchmarks with different write-access patterns in terms of *percentage of write requests* (60%, 75%, 95%) and *percentage of random writes* (20% $\sim$ 95%). The duration for each benchmark is 3,000s and its size is 20GB. For example, the 60% requests of the "60% Writes: 20% Random Writes" benchmark are writes, among which 20% are random.

5.2. Impact of Zone Size

We use five real-world block-level traces with distinct access patterns to evaluate SWANS. To shorten simulation time, we take only the first 500K requests of each trace to conduct experiments. Figure 3 shows the impacts of zone size on the performance of SWANS under the five real-world traces. Obviously, 32MB zone size greatly degrades

Table IV. Data Migration and Redirection Times Under $th_{(5,15)}$

Trace Name	Data Migration	Data Redirection
Build1	5	946
Build2	3	587
Exchange1	3	757
Exchange2	5	629
TPC-C	5	576

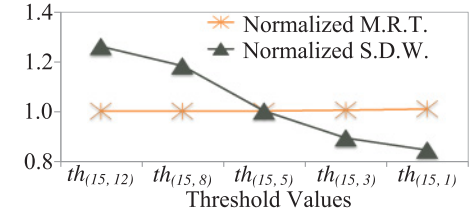
the performance of SWANS in terms of mean response time due to its high cost of data migration (see Figure 3). The mean response time of 16MB and 8MB zone-size configurations is almost the same under all five workloads. On average, the mean response time of the 8MB zone size configuration is smaller than that of the 32MB zone-size configuration by 14.6%. As the zone size increases, the three configurations deliver a similar wear-out performance in terms of SDW. According to Table I, the 8MB zone size, on the other hand, incurs a large *Zone Info* table, which utilizes too much space of the SSD array controller cache. The size of the *Zone Info* table in the 8MB zone-size configuration is 2 times as big as that in the 16MB size configuration, which is 640KB in our experiments. As the total capacity of the SSD array increases to the TB level, the table size in the 8MB zone-size configuration will become unacceptable. A better choice of zone size is highly associated with the SSD array configuration. All experiments use the 16MB zone size.

5.3. Impact of Threshold Values

SWANS employs two thresholds (i.e., $th_{precautionary}$ and $th_{critical}$) to control data-migration and redirection processes. Table IV shows the times of the two processes under five traces. The zone size is 16MB, and four SSDs are used in this experiment. The two thresholds, $th_{precautionary}$ and $th_{critical}$, are set as 5 and 15, respectively. We use a pair of numbers in parentheses to represent the two values. For example, $th_{(5,15)}$ stands for the value of $th_{precautionary}$ and $th_{critical}$, which are 5 and 15, respectively. It is clear that SWANS mainly uses data redirection to even write distributions among SSDs. The number of data migration processes is no larger than 5 under the five traces. However, several hundreds of data-redirection processes are triggered under each trace. The overhead of a data-redirection process is low because it simply redirects each write request targeting a hot SSD to a cold SSD. Therefore, the impact of data migration on the overall performance of an array is limited.

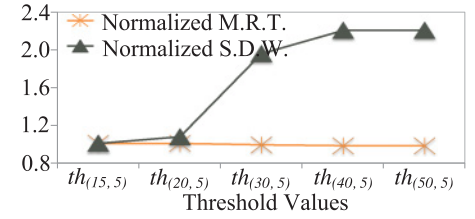
Further, the value of $th_{precautionary}$ and $th_{critical}$ are changed independently to comprehensively understand the impact of the thresholds. The performance in terms of mean response time (MRT) and reliability in terms of standard deviation of writes (SDW) are shown in Figure 4. All the values are normalized to that of under $th_{(15,5)}$. The two tables above the figures present the number of data redirection and data migration under each threshold value setting. Exchange2 trace is chosen as an illustrative example because it has the largest initial SDW value among the five traces. From Figure 4(a), we can see that the write distribution becomes more even (i.e., the value of SDW becomes small) when $th_{precautionary}$ is reduced. In addition, the MRT almost remains unchanged under different values of $th_{precautionary}$. When the value of $th_{precautionary}$ is decreased from 12 to 1, the MRT increases by only 1%. The table above Figure 4(a) shows that a smaller value of $th_{precautionary}$ leads to a larger number of data-redirection processes. The number of data-migration processes is almost unchanged. When $th_{critical}$ is increased (see Figure 4(b)), the write distribution becomes more uneven, whereas the MRT is decreased. From $th_{(15,5)}$ to $th_{(50,5)}$, the SDW value increases by 119.8% and the MRT is reduced by 2.5%. The above table shows that changing $th_{critical}$ affects both

Thresholds	$th_{(15,12)}$	$th_{(15,8)}$	$th_{(15,5)}$	$th_{(15,3)}$	$th_{(15,1)}$
Data Redirection	629	802	930	1039	1150
Data Migration	5	6	5	5	5



(a) Only change the $th_{precautionary}$ value.

Thresholds	$th_{(15,5)}$	$th_{(20,5)}$	$th_{(30,5)}$	$th_{(40,5)}$	$th_{(50,5)}$
Data Redirection	930	930	832	832	832
Data Migration	5	5	2	0	0



(b) Only change the $th_{critical}$ value.

Fig. 4. The impact of threshold values on performance and reliability.

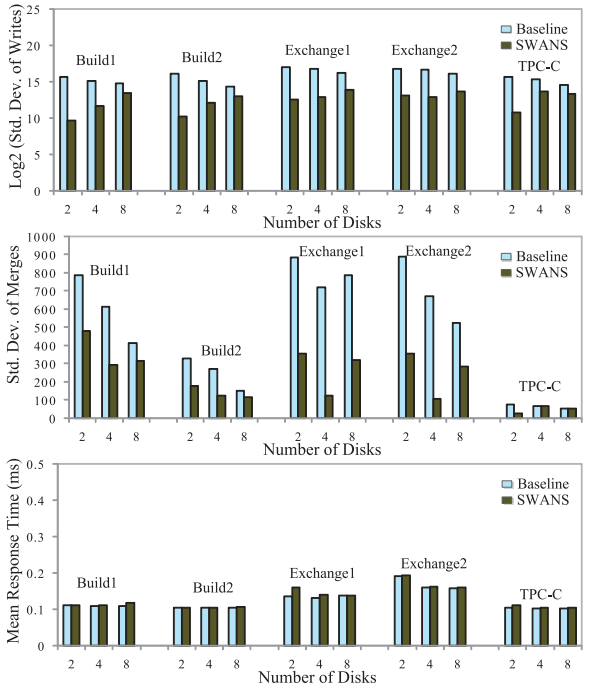


Fig. 5. Experimental results from real-world traces.

the number of data-redirection and data-migration processes. When $th_{critical}$ value is larger than 40, no data migration is triggered.

The two thresholds are empirical values. A smaller threshold value results in a more even write distribution. Meanwhile, the array may get a worse performance. Thus, it is better for users to set the two values based on their application requirements.

5.4. Real-World Trace Evaluation

Figure 5 shows the results from real-world trace simulations. We observe from Figure 5 that the baseline algorithm decreases both SDW and SDM when the number of SSDs increases. In terms of SDW, for Build1 and Build2, SWANSdecreases the SDW, on average, by 23.6 times and 24.6 times, respectively. The improvements in even distribution

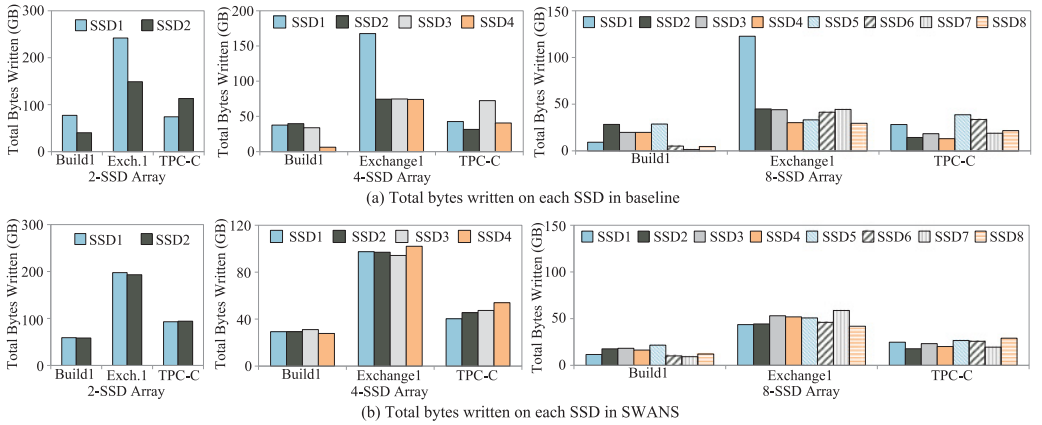


Fig. 6. Written data distribution from real-world traces.

of writes for the two workloads originate from the fact that their write-access patterns are close (see Table III). For Exchange1 and Exchange2, SWANS decreases the SDW, on average, by 10.7 times and 13.4 times, respectively. The gap between the improvements of SDW for the two traces is caused by their very different workload characteristics. While the percentage of write requests for Exchange1 is 50%, Exchange2 has 68% write requests. Further, the update to the write ratio of Exchange2 is almost twice that of Exchange1 (65% versus 34%, Table III). We found that, with more writes, especially more update writes, SWANS has more opportunities to redistribute them evenly across SSDs in an array. For TPC-C, SWANS decreases the SDW 12.1 times. This achievement can also be attributed to the workload characteristics of the trace. SWANS decreases SDW across all traces and SSD numbers, on average, by 16.7x.

As far as the SDM is concerned, compared with baseline, SWANS shows a considerable improvement as well. In particular, for Build1 and Build2, SWANSimproves the SDM, on average, by 1.8 times and 1.6 times, respectively. We discover that the number of merge operations of Build1 is more than that of Build2, for Build1 has a higher update/write ratio. For Exchange1 and Exchange2, SWANS decreases the SDM, on average, 3.6 times and 3.6 times, respectively. For TPC-C, SWANS reduces the SDM by 1.7.

As for MRT, SWANS performs worse than baseline for Build1 and Build2, on average, by 4.1% and 1.9%, respectively. The reason behind this is that the impacts of SWANS on MRT is correlated to the improvement of even distribution of merges as merge operations cause erasures, which are time-consuming. An uneven merge distribution leads to a higher MRT for the entire SSD array. Since both Build1 and Build2 show less improvement in SDM than that of other traces except Exchange1, the limited load-balancing benefit brought by SWANS cannot compensate for its overhead. For Exchange1 and Exchange2, SWANS, on average, increases MRT by 2.2% for Exchange1 and 1.4% for Exchange2. As for TPC-C trace, SWANS increases MRT by 2.9%, on average.

5.5. Total Data Written

Figure 6 illustrates the TBW received by each SSD in an array. It is clear that the TBW on each SSD varies wildly in baseline. For Build1 in a 2-SSD array, SSD1 receives 77.3GB data, which is 1.92 times larger than that in SSD2. Hence, SSD1 wears out 1.92 times faster than SSD2. For Exchange1, SSD1 receives a significantly larger amount of data compared to other disks. On average, SSD1 wears out 1.63 times, 2.25 times,

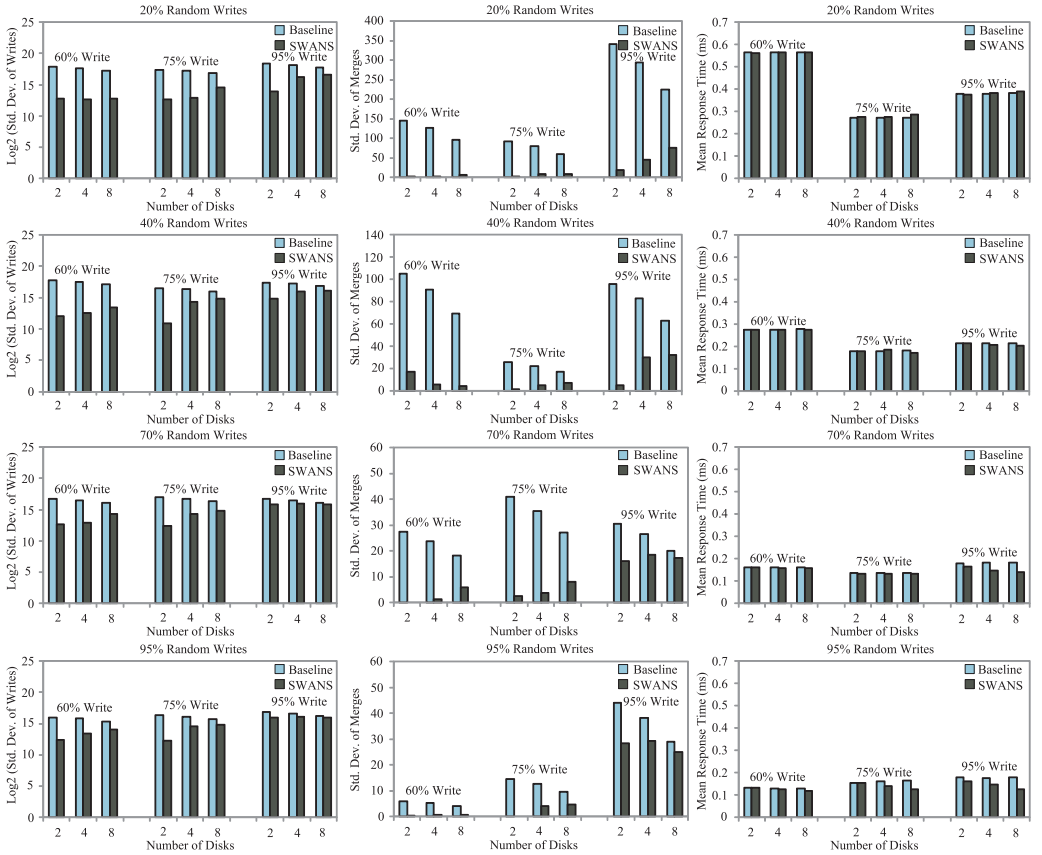


Fig. 7. Experimental results from synthetic benchmarks.

and 3.2 times faster than other disks in 2-SSD, 4-SSD, and 8-SSD array configurations, respectively. The uneven write distribution also occurs in TPC-C, in which the wear-out speed of each SSD varies noticeably. When SWANS is adopted, the differences of TBW received by each SSD in an array significantly reduce. In 2-SSD array and 4-SSD array configurations, compared with the least-loaded SSD, the most-loaded SSD receives only 1.2% (Exchange1) and 8.5% (TPC-C) more written data, on average, respectively. The gap in TBW between the most-written SSD and the least-written SSD in an 8-SSD array shrinks at least 1.3 times.

5.6. Synthetic Benchmark Evaluation

We also test the *percentage of writes* with changing values of *percentage of random writes* and varying number of SSDs in an array in synthetic benchmark simulations. It is observed that SWANS achieves a much better performance in evenly distributing merges across an array in all scenarios. Several other important observations can also be made from Figure 7. First, the performance of SWANS is obviously improved when the 12 benchmarks become random-write dominant. On average, the MRT of SWANS is 1.4% worse than baseline in the case of 20% random writes. However, when the *percentage of random writes* increases to 95%, SWANS turns out to be 11.9% better than baseline. Second, when 95% requests are writes and the number of SSDs increases from 2 to 8, the SDW caused by SWANS also increases in most cases. The reason is

that, with an extremely heavy load of writes, the discrepancy in terms of number of writes among a larger number of SSDs also becomes more obvious. Third, since the size of synthetic traces generated by the Intel Open Storage Toolkit is relatively small (only 20GB) compared with the total capacity of the SSD array (256GB), the MRT of the baseline algorithm barely changes with an increasing number of SSDs. Fourth, with more random writes, the response times are increasing. This is because SWANS works on the flash controller level. A larger number of random writes on the array level indicate that requests have a lower locality. As a result, it is simpler for the array controller to perform load balancing. Furthermore, the low locality leads to a more even write distribution; thus, a few numbers of data migrations are needed for SWANS to adjust write distribution. Therefore, the MRT is reduced while the number of random writes is increased. An interesting observation is that SWANS outperforms the baseline algorithm in MRT when the size of the SSD array is four and eight in most cases. In these two array sizes, SWANS, on average, reduces MRT by approximately 11% and 21%, respectively. The rationale behind this is that SWANS largely balances the write load across the SSDs in an array by either write redirecting or data migrating. In addition, when SWANS migrates the most popular zone, all external writes targeting this zone are temporarily stored in a buffer located in the SSD array controller cache, which largely reduces the number of disk accesses.

In summary, SWANS achieves remarkable improvements in evenly distributing writes and merges in all situations. The implication is that the wear-out across all SSDs in an array is also evenly dispersed. This outcome prevents any individual SSD from being prematurely dead, thus prolongs the service life of the entire SSD array. As for performance, SWANS slightly increases MRT in cases in which a great improvement in SDM cannot be achieved. Still, due to the load-balancing effects brought by SWANS, it can even surprisingly improve performance when significant enhancements in even distribution of writes and merges are realized. In particular, it improves the performance under workloads of 95% writes: 75% and 95% are random.

6. CONCLUSIONS

In enterprise data-intensive applications for which SSD arrays are a must, we argue that an interdisk wear-leveling strategy, which can break the boundaries of individual SSDs, is mandatory. To verify the idea of SSD array level wear-leveling, we design and implement an interdisk wear-leveling strategy called SWANS (*S*moothing *W*ear *A*cross *N* *S*SDs). To the best of our knowledge, this research is the first attempt to addressing wear leveling in a RAID-0 SSD array for server-class data-intensive applications such as database systems and email servers [Lee et al. 2009; SNIA IOTTA Repository 2011; Leutenegger and Dias 1993]. To measure the effectiveness of SWANS, we further develop an event-driven and object-oriented SSD array simulation toolkit called Sim4SSD by significantly enhancing a validated flash device simulator called FlashSim [Kim et al. 2009]. Experimental results from real-world traces demonstrate that SWANS makes remarkable improvements in reducing the variations of the number of both writes and merges among SSDs in a RAID-0 array. We observe that SWANS even achieves a better performance in some scenarios because it is also a load-balancing scheme in the sense that it evenly distributes writes across SSDs.

Future work in this research can be extended in the following directions. We will further improve the performance of SWANS by making its data-migration process multithreaded. Currently, SWANS supports only the RAID-0 structure. We will extend it to other RAID formats, including RAID-5, which provides data redundancy. Developing an interdisk wear-leveling mechanism for RAID-5 is even more challenging because migrating data and its associated parity information incurs a more complicated and higher-cost data-migration scheme. Two possible techniques can be used to apply

interdisk wear-leveling for RAID-5: (1) delaying parity updating to increase the possibility that one parity update absorbs multiple data updates and (2) allocating parity data on any drives without following the conventional round-robin method. Further, integrating the shared-log approaches [Balakrishnan et al. 2012] into traditional RAID architecture is another interesting direction.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*. 57–70.

Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential RAID: Rethinking RAID for SSD reliability. In *Proceedings of the 5th European Conferencew on Computer Systems (EuroSys'10)*. 15–26. DOI: http://dx.doi.org/10.1145/1755913.1755916

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. In *NSDI*. 1–14.

Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. 2003. Introduction to flash memory. *Proceedings of the IEEE* 91, 4, 489–502.

Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. 2007. A design for high-performance flash disks. *SIGOPS Operating Systems Review* 41, 2, 88–93.

Simona Boboila and Peter Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *USENIX FAST*. 9–25.

Cactus Technologies. 2008. *Wear Leveling Static vs Dynamic*. Technical Report. Cactus Technologies, Hong Kong, P.R.China.

Li-Pin Chang and Chun-Da Du. 2009. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Transactions on Design Automation of Electronic Systems* 15, 1, 6.

Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*. 181–192.

Renhai Chen, Yi Wang, and Zili Shao. 2013. DHeating: Dispersed heating repair for self-healing NAND flash memory. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 7.

Ludmila Cherkasova and Minaxi Gupta. 2004. Analysis of enterprise media server workloads: Access patterns, locality, content evolution, and rates of change. *IEEE/ACM Transactions on Networking* 12, 5, 781–794.

Bruce Cullen. 2014. SSD Drive Failure: What Causes Solid State Drives to Stop Working and Fail? Retrieved April 1, 2016 from http://www.eprovided.com/data-recovery-blog/ssd-drive-failure-causes-solid-state-drive-stop-working-failures/.

Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. 1999. *The DiskSim Simulation Environment Version 3.0 Reference Manual*. Technical Report. Carnegie Mellon University, Pittsburgh, PA 15213, USA.

Geoff Gasior. 2013. Introducing the SSD Endurance Experiment. Retrieved April 1, 2016 from http://techreport.com/review/24841/introducing-the-ssd-endurance-experiment.

M. Gómez and Vicente Santonja. 2002. Characterizing temporal locality in I/O workload. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*.

Jim Gray and Bob Fitzgerald. 2008. Flash disk opportunity for server applications. *Queue* 6, 4, 18–23.

Jim Gray, Bob Horst, and Mark Walker. 1990. Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput. In *Proceedings of the 16th VLDB Conference*. 148–161.

Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. *DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings*. Vol. 44. ACM.

Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. 2010. Dash: A recipe for a flash-based data intensive supercomputer. In *SC*. 1–11.

Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*. ACM, 96–107.

Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *26th MSST*. 1–12.

Intel. 2013. Intel Solid-State Drive Specification. Retrieved April 1, 2016 from http://www.intel.com/content/www/us/en/solid-state-drives/ssd-530-sata-sp ecification.html.

Intel. 2014. Intel SSD Product Comparison. Retrieved April 1, 2016 from http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-ssd.html. (2014).

Anxiao Jiang, Robert Mateescu, Eitan Yaakobi, Jehoshua Bruck, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf. 2010. Storage coding for wear leveling in flash memories. *IEEE Transactions on Information Theory* 56, 10, 5290–5299.

Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2007. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES*. 160–164.

Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. Flashsim: A simulator for NAND flash-based solid-state drives. In *International Conference on Advances in System Simulation*. 125–131.

Andrew Ku. 2011. Investigation: Is Your SSD More Reliable Than A Hard Drive? Retrieved April 1, 2016 from http://www.tomshardware.com/reviews/ssd-reliability-failure-rate,2923.html. (2011).

Sang-Won Lee, Bongki Moon, and Chanik Park. 2009. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*. 863–870.

Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computer Systems* 6, 3, 18.

Scott T. Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *Proceedings of the 1993 SIGMOD International Conference on Management of Data*. 22–31.

Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. 2012. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Transactions on Storage* 8, 1.

M. Mesnier. 2001. Intel open storage toolkit. Retrieved April 1, 2016 from http://www.sourceforge.org/projects/intel-iscsi.

Muthukumar Murugan and David H. C. Du. 2011. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *27th MSST*. 1–12.

Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems*. 145–158.

NetApp. 2014. NetApp EF540 Technical Specifications. Retrieved April 1, 2016 from http://www.netapp.com/us/products/storage-systems/flash-ef540/ef540-tech-spe cs.aspx. (2014).

Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure trends in a large disk drive population. In *FAST*, Vol. 7. 17–23.

SNIA IOTTA Repository. 2011. Build and Ecahange server traces. (2011). http://iotta.snia.org/traces/list/BlockIO.

Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST*, Vol. 7. 1–16.

Qi Wu, Guiqiang Dong, and Tong Zhang. 2011. Exploiting heat-accelerated flash memory wear-out recovery to enable self-healing SSDs. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*.

Soraya Zertal and Peter G. Harrison. 2011. Investigating flash memory wear levelling and execution modes. *Simulation* 87, 12.

Received April 2014; revised November 2014; accepted April 2015