SFS: Random Write Considered Harmful in Solid State Drives

Changwoo Min[a], Kangnyeon Kim[b], Hyunjin Cho[c], Sang-Won Lee[d], Young Ik Eom[e]
[abde]*Sungkyunkwan University, Korea*
[ac]*Samsung Electronics, Korea*
{multics69[a], kangnuni[b],wonlee[d],yieom[e]}@ece.skku.ac.kr, hj1120.cho[c]@samsung.com

Abstract

Over the last decade we have witnessed the relentless technological improvement in flash-based solid-state drives (SSDs) and they have many advantages over hard disk drives (HDDs) as a secondary storage such as performance and power consumption. However, the random write performance in SSDs still remains as a concern. Even in modern SSDs, the disparity between random and sequential write bandwidth is more than ten-fold. Moreover, random writes can shorten the limited lifespan of SSDs because they incur more NAND block erases per write.

In order to overcome these problems due to random writes, in this paper, we propose a new file system for SSDs, SFS. First, SFS exploits the maximum write bandwidth of SSD by taking a log-structured approach. SFS transforms all random writes at file system level to sequential ones at SSD level. Second, SFS takes a new data grouping strategy *on writing*, instead of the existing data separation strategy *on segment cleaning*. It puts the data blocks with similar update likelihood into the same segment. This minimizes the inevitable segment cleaning overhead in any log-structured file system by allowing the segments to form a sharp bimodal distribution of segment utilization.

We have implemented a prototype SFS by modifying Linux-based NILFS2 and compared it with three state-of-the-art file systems using several realistic workloads. SFS outperforms the traditional LFS by up to 2.5 times in terms of throughput. Additionally, in comparison to modern file systems such as ext4 and btrfs, it drastically reduces the block erase count inside the SSD by up to 7.5 times.

1 Introduction

NAND flash memory based SSDs have been revolutionizing the storage system. An SSD is a purely electronic device with no mechanical parts, and thus can provide lower access latencies, lower power consumption, lack of noise, shock resistance, and potentially uniform random access speed. However, there remain two serious problems limiting wider deployment of SSDs: limited lifespan and relatively poor random write performance. The limited lifespan of SSDs remains a critical concern in reliability-sensitive environments, such as data centers [5]. Even worse, the ever-increased bit density for higher capacity in NAND flash memory chips has resulted in a sharp drop in the number of program/erase cycles from 10K to 5K for the last two years [4]. Meanwhile, previous work [12, 9] shows that random writes can cause internal fragmentation of SSDs and thus lead to performance degradation by an order of magnitude. In contrast to HDDs, the performance degradation in SSDs caused by the fragmentation lasts for a while after random writes are stopped. The reason for this is that random writes cause the data pages in NAND flash blocks to be copied elsewhere and erased. Therefore, the lifespan of an SSD can be drastically reduced by random writes.

Not surprisingly, researchers have devoted much effort to resolving these problems. Most of work has been focused on a *flash translation layer* (FTL) – an SSD firmware emulating an HDD by hiding the complexity of NAND flash memory. Some studies [24, 14] improved random write performance by providing more efficient logical to physical address mapping. Meanwhile, other studies [22, 14] propose a separation of hot/cold data to improve random write performance. However, such under-the-hood optimizations are purely based on logical block addresses (LBA) requested by a file system so that they would become much less effective for the no-overwrite file systems [16, 48, 10] in which every write to the same file block is always redirected to a new LBA. There are other attempts to improve random write performance especially for database systems [23, 39]. Each attempt proposes a new database storage scheme, taking into account the performance characteristics of SSDs. However, despite the fact that these flash-conscious techniques are quite effective in specific applications, they cannot provide the benefit of such optimization to general applications.

In this paper, we propose a novel file system, SFS, that can improve random write performance and extend the lifetime of SSDs. Our work is motivated by LFS [32], which writes all modifications to disk sequentially in a log-like structure. In LFS, the segment cleaning overhead can severely degrade performance [35, 36] and shorten the lifespan of an SSD. This is because quite a high number of pages need to be copied to secure a large empty chunk for a sequential write at every segment cleaning. In designing SFS, we investigate how to take advantage of performance characteristics of SSD and I/O workload skewness to reduce the segment cleaning overhead.

This paper makes the following specific contributions:

- We introduce the design principles for SSD-based file systems. The file system should exploit the performance characteristics of SSD and directly utilize file block level statistics. In fact, the architectural differences between SSD and HDD results in different performance characteristics for each system. One interesting example is that, in SSD, the additional overhead of random write disappears only when the unit size of random write requests becomes a multiple of a certain size. To this end, we take log-structured approach with a carefully selected segment size.
- To reduce the segment cleaning overhead in the log-structured approach, we propose an eager *on writing* data grouping scheme that classifies file blocks according to their update likelihood and writes those with similar update likelihoods into the same segment. The effectiveness of data grouping is determined by proper selection of the grouping criteria. For this, we propose an *iterative segment quantization* algorithm to determine the grouping criteria, while considering disk-wide hotness distribution. We also propose *cost-hotness policy* for victim segment selection. Our eager data grouping will colocate frequently updated blocks in the same segments; thus most blocks in those segments are expected to become rapidly invalid. Consequently, the segment cleaner can easily find a victim segment with few live blocks and thus can minimize the overhead of copying the live blocks.
- Using a number of realistic and synthetic workloads, we show that SFS significantly outperforms LFS and state-of-the-art file systems such as ext4 and btrfs. We also show that SFS can extend the lifespan of an SSD by drastically reducing the number of NAND flash block erases. In particular, while the random write performance of the existing file systems is highly dependent on the random write performance of SSD, SFS can achieve nearly maximum sequential write bandwidth of SSD for random writes at the file system level. Therefore, SFS can provide high performance even on mid-range or low-end SSDs as long as their sequential write performance is comparable to high-end SSDs.

The rest of this paper is organized as follows. Section 2 overviews the characteristics of SSD and I/O workloads. Section 3 describes the design of SFS in detail, and Section 4 shows the extensive evaluation of SFS. Related work is described in Section 5. Finally, in Section 6, we conclude the paper.

2 Background

2.1 Flash Memory and SSD Internals

NAND flash memory is the basic building block of SSDs. *Read* and *write* operations are performed at the granularity of a *page* (e.g. 2 KB or 4 KB), and the *erase* operation is performed at the granularity of a *block* (composed of 64 – 128 pages). NAND flash memory differs from HDDs in several aspects: (1) asymmetric speed of read and write operations, (2) no in-place overwrite – the whole block must be erased before overwriting any page in that block, and (3) limited program/erase cycles – a single-level cell (SLC) has roughly 100K erase cycles and a typical multi-level cell (MLC) has roughly 10K erase cycles.

A typical SSD is composed of host interface logic (SATA, USB, and PCI Express), an array of NAND flash memories, and an SSD controller. A *flash translation layer* (FTL) run by an SSD controller emulates an HDD by exposing a linear array of *logical block addresses* (LBAs) to the host. To hide the unique characteristics of flash memory, it carries out three main functions: (1) managing a *mapping table* from LBAs to physical block addresses (PBAs), (2) performing *garbage collection* to recycle invalidated physical pages, and (3) *wear-leveling* to wear out flash blocks evenly in order to extend the SSD's lifespan. Agrawal et al. [2] comprehensively describe the broad design space and tradeoffs of SSD.

Much research has been carried out on FTL to improve performance and extend the lifetime [18, 24, 22, 14]. In a *block-level FTL* scheme, a logical block number is translated to a physical block number and the logical page offset within a block is fixed. Since the mapping in this instance is coarse-grained, the mapping table is small enough to be kept in memory entirely. Unfortunately, this results in a higher garbage collection overhead. In contrast, since a *page-level FTL* scheme manages a fine-grained page-level mapping table, it results in a lower garbage collection overhead. However, such fine-grained mapping requires a large mapping table on RAM. To overcome such technical difficulties, *hybrid FTL* schemes [18, 24, 22] extend the block-level FTL. These schemes logically partition flash blocks into *data blocks* and *log blocks*. The majority of data blocks are mapped using block level mapping to reduce the required RAM size and log blocks are mapped using page-level mapping to reduce the garbage collection overhead. Similarly, DFTL [14] extends the page-level mapping by

	SSD-H	SSD-M	SSD-L
Manufacturer	Intel	Samsung	Transcend
Model	X25-E	S470	JetFlash 700
Capacity	32 GB	64 GB	32 GB
Interface	SATA	SATA	USB 3.0
Flash Memory	SLC	MLC	MLC
Max Sequential Reads (MB/s)	216.9	212.6	69.1
Random 4 KB Reads (MB/s)	13.8	10.6	5.3
Max Sequential Writes (MB/s)	170	87	38
Random 4 KB Writes (MB/s)	5.3	0.6	0.002
Price ($/GB)	14	2.3	1.4

Table 1: Specification data of the flash devices. List price is as of September 2011.

selectively caching page-level mapping table entries on RAM.

2.2 Imbalance between Random and Sequential Write Performance in SSDs

Most SSDs are built on an array of NAND flash memories, which are connected to the SSD controller via multiple channels. To exploit this inherent parallelism for better I/O bandwidth, SSDs perform read or write operations as a unit of a *clustered page* [19] that is composed of physical pages striped over multiple NAND flash memories. If the request size is not a multiple of the clustered page size, extra read or write operations are performed in the SSD and the performance is degraded. Thus, the clustered page size is critical in I/O performance.

Write performance in SSDs is highly workload dependent and is eventually limited by the garbage collection performance of FTL. Previous work [12, 9, 39, 37, 38] has reported that random write performance drops by more than an order of magnitude after extensive random updates and returns to the initial high performance only after extensive sequential writes. The reason for this is that random writes increase the garbage collection overhead in FTL. In a hybrid FTL, random writes increase the associativity between log blocks and data blocks, and incur the costly *full merge* [24]. In page-level FTL, as it tends to fragment blocks evenly, garbage collection has large copying overhead.

In order to improve garbage collection performance, SSD combines several blocks striped over multiple NAND flash memories into a *clustered block* [19]. The purpose of this is to erase multiple physical blocks in parallel. If all write requests are aligned in multiples of the clustered block size and their sizes are also multiples of the clustered block size, random write updates and invalidates a clustered block as a whole. Therefore, in hybrid FTL, a *switch merge* [24] with the lowest overhead occurs. Similarly, in page-level FTL, empty blocks with no live pages are selected as victims for garbage collection. The result of this is that random write performance converges with sequential write performance. To verify this, we measured sequential write and random write throughput on three different SSDs in Table 1, ranging from a high-end SLC SSD (SSD-H) to a low-end USB memory stick (SSD-L). To determine sustained write performance, dummy data equal to twice the device's capacity is first written for aging, and the throughput of subsequent writing for 8GB is measured. Figure 1 shows that random write performance catches up with sequential write performance when the request size is 16 MB or 32 MB. These unique performance characteristics motivate the second design principle of SFS: write bandwidth maximization by sequential writes to SSD.

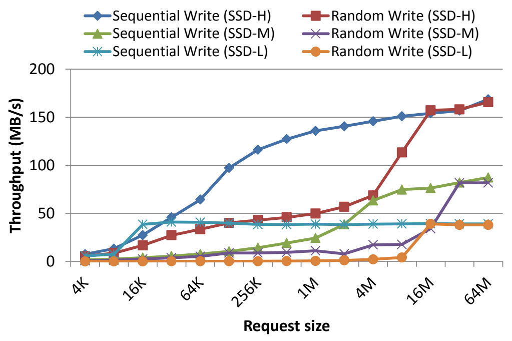


Figure 1: Sequential vs. random write throughput.

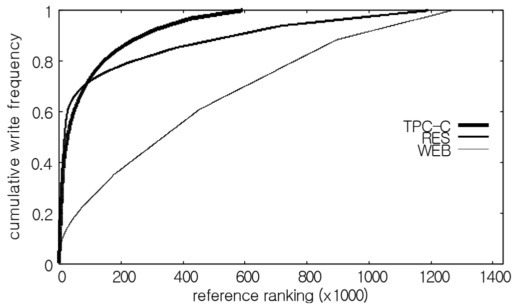


Figure 2: Cumulative write frequency distribution.

2.3 Skewness in I/O Workloads

Many researchers have pointed out that I/O workloads have non-uniform access frequency distribution [34, 31, 23, 6, 3, 33, 11]. A disk-level trace of personal workstations at Hewlett Packard laboratories exhibits a high locality of references in that 90% of the writes go to the 1% of blocks [34]. Roselli et al. [31] analyzed file system traces collected from four different groups of machines: an instructional laboratory, a set of computers used for research, a single web server, and a set of PCs running Windows NT. They found that files tend to be either read-mostly or write-mostly and the writes show substantial locality. Lee and Moon [23] showed that the update frequency of TPC-C workloads is highly skewed, in that 29% writes go to 1.6% of pages.

Bhadkamkar et al. [6] collected and investigated I/O traces of office and developer desktop workloads, a version control server, and a web server. Their analysis confirms that the top 20% most frequently accessed blocks contribute to a substantially large (45-66%) percentage of total access. Moreover, high and low frequency blocks are spread over the entire disk area in most cases. Figure 2 depicts the cumulative write frequency distribution of three real workloads: an IO trace collected by ourselves while running TPC-C [40] using Oracle DBMS (TPC-C), a research group trace (RES), and a web sever trace equipped with Postgres DBMS (WEB) collected by Roselli et al [31]. This observation motivates the third design principle of SFS: block grouping according to write frequency.

3 Design of SFS

SFS is motivated by a simple question: *How can we utilize the performance characteristics of SSD and the skewness of the I/O workload in designing an SSD-based file system?* In this section, we describe the rationale behind the design decisions in SFS, its system architecture, and several key techniques including hotness measure, segment quantization, segment writing, segment cleaning and victim selection policy, and crash recovery.

3.1 SFS: Design for SSD-based File Systems of the 2010s

Historically, existing file systems and modern SSDs have evolved separately without consideration of each other. With the exception of the recently introduced TRIM command, the two layers communicate with each other through simple read and write operations using only LBA information. For this reason, there are many impedance mismatches between the two layers, thus hindering the optimal performance when both layers are simply used together. In this section, we explain three design principles of SFS. First, we identify general performance problems when the existing file systems are running on modern SSDs and suggest that a file system should exploit the file block semantics directly. Second, we propose to take a log-structured approach based on the observation that the random write bandwidth is much slower than the sequential one. Third, we criticize that the existing *lazy* data grouping in LFS during segment cleaning fails to fully utilize the skewness in write patterns and argue that an *eager* data grouping is necessary to achieve sharper bimodality in segments. In followings we will describe each principle in detail.

File block level statistics – Beyond LBA: The existing file systems perform suboptimally when running on top of SSDs with current FTL technology. This suboptimal performance can be attributed to poor random write performance in SSDs. One of the basic functionalities of file systems is to allocate an LBA for a file block. With regard to this LBA allocation, there have been two general policies in file system community: *in-place-update* and *no-overwrite*. The in-place-update file systems such as FAT32 [27] and ext4 [25] always overwrite a dirty file block to the same LBA so that the same LBA ever corresponds to a file block unless the file frees the block. This *unwritten assumption* in file systems, together with the LBA level interface between file systems and storage devices, allows the underlying FTL mechanism in SSDs to exploit the overwrites against the same LBA address. In fact, most FTL research [24, 22, 13, 14] has focused on improving the random write performance based on the LBA level write patterns. Despite the relentless improvement in FTL technology, the random write bandwidth in modern SSDs, as presented in Section 2.2, still lags far behind the sequential one.

Meanwhile, several no-overwrite file systems have been implemented, such as btrfs [10], ZFS [48], and WAFL [16], where dirty file blocks are written to new LBAs. These systems are known to improve scalability, reliability, and manageability [29]. In those systems, however, because the unwritten assumption between file blocks and their corresponding LBAs is broken, the FTL receives new LBA write request for every update of a file block and thus cannot exploit any file level hotness semantics for random write optimization.

In summary, the LBA-based interface between the *no-overwrite* file systems and storage devices does not allow the file blocks' hotness semantic to flow down to the storage layer. In addition, the relatively poor random write performance in SSDs is the source of suboptimal performance in the *in-place-update* file systems. Consequently, we suggest that file systems should directly exploit the hotness statistics at the *file block level*. This allows for optimization of the file system performance regardless of whether the unwritten assumption holds and how the underlying SSDs perform on random writes.

Write bandwidth maximization by sequentialized writes to SSD: In Section 2.2, we show that the random write throughput becomes equal to the sequential write throughput only when the request size is a multiple of the clustered block size. To exploit such performance characteristics, SFS takes a log-structured approach that turns random writes at the file level into sequential writes at the LBA level. Moreover, in order to utilize nearly 100% of the raw SSD bandwidth, the segment size is set to a multiple of the clustered block size. The result is that the performance of SFS will be limited by the maximum sequential write performance regardless of random write performance.

Eager *on writing* data grouping for better bimodal segmentation: When there are not enough free segments, a segment cleaner copies the live blocks from vic-

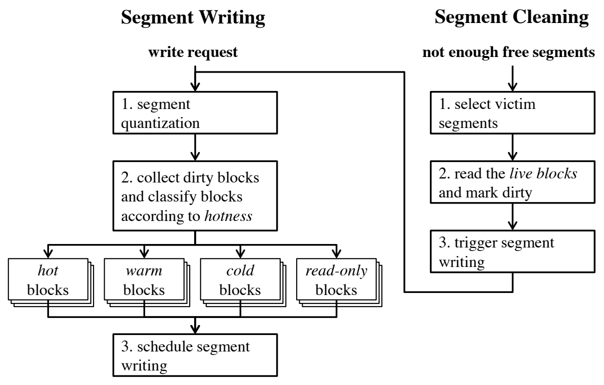

Segment Writing
write request
1. segment quantization
2. collect dirty blocks and classify blocks according to *hotness*
hot blocks
warm blocks
cold blocks
read-only blocks
3. schedule segment writing
Segment Cleaning
not enough free segments
1. select victim segments
2. read the *live blocks* and mark dirty
3. trigger segment writing

Figure 3: Overview of writing process and segment cleaning in SFS.

tim segments in order to secure free segments. Since segment cleaning includes reads and writes of live blocks, it is the main source of overhead in any log-structured file system. Segment cleaning cost becomes especially high when cold data are mixed with hot data in the same segment. Since cold data are updated less frequently, they are highly likely to remain live at the segment cleaning and thus be migrated to new segments. If hot data and cold data are grouped into different segments, most blocks in the hot segment will be quickly invalidated, while most blocks in the cold segment will remain live. As a result, the segment utilization distribution becomes bimodal: most of the segments are almost either full or empty of live blocks. The cleaning overhead is drastically reduced, because the segment cleaner can almost always work with nearly empty segments. To form a bimodal distribution, LFS uses a cost-benefit policy [32] that prefers cold segments over hot segments. However, LFS writes data regardless of hot/cold and then tries to separate data lazily *on segment cleaning*. If we can categorize hot/cold data when it is first written, there is much room for improvement.

In SFS, we classify data *on writing* based on file block level statistics as well as segment cleaning. In such early data grouping, since segments are already composed of homogeneous data with similar update likelihood, we can significantly reduce segment cleaning overhead. This is particularly effective because I/O skewness is common in real world workloads, as shown in Section 2.3.

3.2 SFS Architecture

SFS has four core operations: segment writing, segment cleaning, reading, and crash recovery. Segment writing and segment cleaning are particularly crucial for performance optimization in SFS, as depicted in Figure 3. Because the read operation in SFS is same as that of existing log-structured file systems, we will not cover its detail in this paper.

As a measure for representing the future update likelihood of data in SFS, we define *hotness* for file block, file, and segment, respectively. As the hotness is higher, the data is expected to be updated sooner. The first step of segment writing in SFS is to determine the hotness criteria for block grouping. This is, in turn, determined by segment quantization that quantizes a range of hotness values into a single hotness value for a group. For the sake of brevity, it is assumed throughout this paper that there are four segment groups: hot, warm, cold, and read-only groups. The second step of segment writing is to calculate the block hotness for each block and assign them to the nearest quantized group by comparing the block hotness and the group hotness. At this point, those blocks with similar hotness levels should belong to the same group (i.e. their future update likelihood is similar). As the final step of segment writing, SFS always fills a segment with blocks belonging to the same group. If the number of blocks in a group is not enough to fill a segment, the segment writing of the group is deferred until the segment is filled. This eager grouping of file blocks according to the hotness measure serves to colocate blocks with similar update likelihoods in the same segment. Therefore, segment writing in SFS is very effective at achieving sharper bimodality in segment utilization distribution.

Segment cleaning in SFS consists of three steps: select victim segments, read the live blocks in victim segments into the page cache and mark the live blocks as dirty, and trigger the writing process. The writing process treats the live blocks from victim segments the same as normal blocks; each live block is classified into a specific quantized group according to its hotness. After all the live blocks are read into the page cache, the victim segments are then marked as free so that they can be reused for writing. For better victim segment selection, *cost-hotness policy* is introduced, which takes into account both the number of live blocks in segment (i.e. cost) and the segment hotness.

In the following sections, we will explain each component of SFS in detail: how to measure hotness (§ 3.3), segment quantization (§ 3.4), segment writing (§ 3.5), segment cleaning (§ 3.6), and crash recovery (§ 3.7).

3.3 Measuring Hotness

In SFS, *hotness* is used as a measure of how likely the data is to be updated. Hotness is defined for file block, file, and segment, respectively. Although it is difficult to estimate data hotness without prior knowledge of future access pattern, SFS exploits both the skewness and the temporal locality in the I/O workload so as to estimate the update likelihood of data. From the skewness observed in many workloads, frequently updated data tends to be updated quickly. Moreover, because of the temporal locality in references, the recently updated data is likely to be changed quickly. Thus, using the skewness and the temporal locality, *hotness* is defined as $\frac{\text{write count}}{\text{age}}$. Each hotness of file block, file, and segment is specifically defined as follows.

First, *block hotness* H_b is defined by age and write count of a block as follows:

$$H_b = \begin{cases} \frac{W_b}{T - T_b^m} & \text{if } W_b > 0, \\ H_f & \text{otherwise.} \end{cases}$$

where T is the current time, T_b^m is the last modified time of the block, and W_b is the total number of writes on the block since the block was created. If a block is newly created ($W_b = 0$), H_b is defined as the hotness of the file that the block belongs to.

Next, *file hotness* H_f is used to estimate the hotness of a newly created block. It is defined by age and write count of a file as follows:

$$H_f = \frac{W_f}{T - T_f^m}$$

where T_f^m is the last modified time of the file, and W_f is the total number of block updates since the file was created.

Finally, *segment hotness* represents how likely a segment is to be updated. Since a segment is a collection of blocks, it is reasonable to derive its hotness from the hotness of live blocks contained within. That is, as the hotness of live blocks in a segment is higher, the segment hotness also becomes higher. Therefore, we define hotness of a segment H_s as the average hotness of the live blocks in the segment. However, it is expensive to calculate H_s because the liveness of all blocks in a segment must be tested. To determine H_s for all segments in a disk, the liveness of all blocks in the disk must be checked. To alleviate this cost, we approximately calculate the average hotness of live blocks in a segment as follows:

$$\begin{aligned} H_s &= \frac{1}{N}\sum_i H_{b_i} \\ &\approx \frac{\text{mean of write count of live blocks}}{\text{mean of age of live blocks}} \\ &= \frac{\sum_i W_{b_i}}{N \cdot T - \sum_i T_{b_i}^m} \end{aligned}$$

where N is the number of live blocks in a segment, H_{b_i}, $T_{b_i}^m$, and W_{b_i} are block hotness, last modified time, and write count of i-th live block, respectively. When a segment is created, SFS stores $\sum_i T_{b_i}^m$ and $\sum_i W_{b_i}$ in the segment usage meta-data file (SUFILE), and updates them by subtracting $T_{b_i}^m$ and W_{b_i} whenever a block is invalidated. Using this approximation, we can incrementally calculate H_s of a segment without checking the liveness of blocks in the segment. We will elaborate on how to manage meta-data for hotness in Section 4.1.

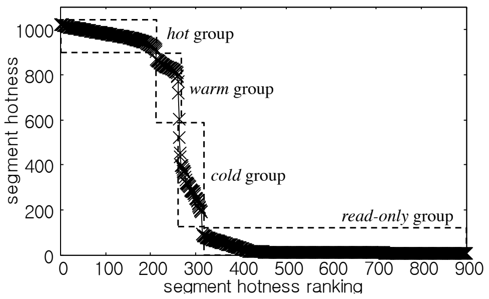


Figure 4: Example of segment quantization.

3.4 Segment Quantization

In order to minimize the overhead of copying the live blocks during segment cleaning, it is crucial for SFS to properly group blocks according to hotness and then to write them in grouped segments. The effectiveness of block grouping is determined by the grouping criteria. In fact, improper criteria may colocate blocks from different groups into the same segment, thus deteriorating the effectiveness of grouping. Ideally, grouping criteria should consider the distribution of all blocks' hotness in the file system, yet in reality this is too costly. Thus, we instead use segment hotness as an approximation of block hotness and devise an algorithm to calculate the criterion, *iterative segment quantization*.

In SFS, *segment quantization* is a process used to partition the hotness range of a file system into k sub-ranges and calculate a quantized value for each sub-range representing a group. There are many alternative ways to quantize hotness. For example, each group can be quantized using *equi-height partitioning* or *equi-width partitioning*. Equi-height partitioning equally divides the whole hotness range into multiple groups and equi-width partitioning makes each group have an equal number of segments. In Figure 4, the segment hotness distribution is computed by measuring the hotness for all segments on the disk after running TPC-C workload under 70% disk utilization. In such a distribution where most segments are not hot, however, both approaches fail to correctly reflect the hotness distribution and the resulting group quantization is suboptimal.

In order to correctly reflect the hotness distribution of segments and to properly quantize them, we propose an *iterative segment quantization* algorithm. Inspired by the data clustering approach in statistics domain [15], our iterative segment quantization partitions segments into k groups and tries to find the centers of natural groups through an iterative refinement approach. A detailed description of the algorithm is as follows:

1. If the number of written segments is less than or equal to k, assign a randomly selected segment hotness to initial value of H_{g_i}, which denotes hotness of the i-th group.
2. Otherwise update H_{g_i} as follows:
   (a) Assign each segment to the group G_i whose hotness is closest to the segment hotness.
   $$G_i = \{H_{s_j} : \|H_{s_j} - H_{g_i}\| \leq \|H_{s_j} - H_{g_{i^*}}\| \text{ for all } i^* = 1, \ldots, k\}$$
   (b) Calculate the new means to be the group hotness H_{g_i}.
   $$H_{g_i} = \frac{1}{|G_i|} \sum_{H_{s_j} \in G_i} H_{s_j}$$
3. Repeat Step 2 until H_{g_i} no longer changes or three times at most.

Despite the fact that its computational overhead increases in proportion to the number of segments, the large segment size means that the overhead of the proposed algorithm is reasonable (32 segments for 1 GB disk space given 32 MB segment size). To further reduce the overhead, SFS stores H_{g_i} in meta-data and reloads them at mounting for faster convergence.

3.5 Segment Writing

As illustrated in Figure 3, segment writing in SFS consists of two sequential steps: one to group dirty blocks in the page cache and the other to write the blocks groupwise in segments. Segment writing is invoked in four cases: (a) SFS periodically writes dirty blocks every five seconds, (b) flush daemon forces a reduction in the number of dirty pages in the page cache, (c) segment cleaning occurs, and (d) an *fsync* or *sync* occurs. The first step of segment writing is segment quantization: all H_{g_i} are updated as described in Section 3.4. Next, the block hotness H_b of every dirty block is calculated, and each block is assigned to the group H_{g_i} whose hotness is closest to the block hotness.

To avoid blocks in different groups being colocated in the same segment, SFS completely fills a segment with blocks from the same group. In other words, among all groups, only the groups large enough to completely fill a segment are written. Thus, when the group size, i.e. the number of blocks belonging to a group, is less than the segment size, SFS will defer writing the blocks to the segment until the group size reaches the segment size. However, when an *fsync* or *sync* occurs or SFS initiates a *check-point*, every dirty block including the deferred blocks should be immediately written to segment regardless of the group size. In this case, we take a best-effort approach: at first, writing out blocks groupwise as many as possible, then writing only the remaining blocks regardless of group. In all cases, writing a block accompanies updating relevant meta-data, T_b^m, W_b, T_f^m, W_f, $\sum_i T_{b_i}^m$, and $\sum_i W_{b_i}$, and invalidating the liveness of the overwritten block. Since the writing process continuously reorganizes file blocks according to hotness, it helps to form sharp bimodal distribution of segment utilization, and thus to reduce the segment cleaning overhead. Further, it almost always generates aligned large sequential write requests that are optimal for SSD.

Because the blocks under segment cleaning are handled similarly, their writing can also be deferred if the number of live blocks belonging to a group is not enough to completely fill a segment. As such, there is a danger that the not-yet-written blocks under segment cleaning might be lost if the originating segments of the blocks are already overwritten by new data but a system crash or a sudden power off is encountered. To cope with such data loss, two techniques are introduced. First, SFS manages a free segment list and allocates segments in the *least recently freed (LRF)* order. Second, SFS checks whether writing a normal block could cause a not-yet-written block under segment cleaning to be overwritten. Let S^t denote a newly allocated segment and S^{t+1} denote a segment that will be allocated in next segment allocation. If there are not-yet-written blocks under segment cleaning that originate in S^{t+1}, SFS writes such blocks to S^t regardless of grouping. This guarantees that not-yet-written blocks under segment cleaning are never overwritten before they are written elsewhere. The segment-cleaned blocks are thus never lost, even in a system crash or a sudden power off, because they always have an on-disk copy. The LRF allocation scheme increases the opportunity for a segment-cleaned block to be written by block grouping rather than this scheme. The details of minimizing the overhead in this process are omitted from this paper.

3.6 Segment Cleaning: Cost-hotness policy

In any log-structured file system, the victim selection policy is critical to minimizing the overhead of segment cleaning. There are two well-known segment cleaning policies: *greedy policy* [32] and *cost-benefit policy* [32, 17]. Greedy policy [32] always selects segments with the smallest number of live blocks, hoping to reclaim as much space as possible with the least copying out overhead. However, it does not consider the hotness of data blocks during segment cleaning. In practice, because the cold data tends to remain unchanged for a long time before it becomes invalidated, it would be very beneficial to separate cold data from hot data. To this end, cost-benefit policy [32, 17] prefers cold segments to hot segments when the number of live blocks is equal. Even though it is critical to estimate how long a segment remains unchanged, cost-benefit policy simply uses the last modified time of any block in the segment (i.e. the age of the youngest block) as a simple measure of the segment's update likelihood.

As a natural extension of cost-benefit policy, we introduce *cost-hotness policy*; since hotness in SFS directly represents the update likelihood of segment, we use segment hotness instead of segment age. Thus, SFS chooses a victim among the segments, which maximizes the following formula:

$$\text{cost-hotness} = \frac{\text{free space generated}}{\text{cost} * \text{segment hotness}} = \frac{(1 - U_s)}{2U_s H_s}$$

where U_s is segment utilization, i.e. the fraction of the live blocks in a segment. The cost of collecting a segment is $2U_s$ (one U_s to read valid blocks and the other U_s to write them back). Although cost-hotness policy needs to access the utilization and the hotness of all segments, it is very efficient because our implementation keeps them in segment usage meta-data file (SUFILE) and meta-data size per segment is quite small (48 bytes long). All segment usage information is very likely to be cached in memory and can be accessed without accessing the disk in most cases. We will describe the detail of meta-data management in Section 4.1.

In SFS, the segment cleaner is invoked when the disk utilization exceeds a *water-mark*. The water-mark for the our experiments is set to 95% of the disk capacity and the segment cleaning is allowed to process up to three segments at once (96 MB given the segment size of 32 MB). The prototype did not implement the idle time cleaning scheme suggested by Blackwell et al. [7], yet this could be seamlessly integrated with SFS.

3.7 Crash Recovery

Upon a system crash or a sudden power off, the in progress write operations may leave the file system inconsistent. This is because dirty data blocks or meta-data blocks in the page cache may not be safely written to the disk. In order to restore such inconsistencies from failures, SFS uses a *check-point* mechanism; on re-mounting after a crash, the file system is rolled back to the last check-point state, and then resumes in a normal manner. A check-point is the state in which all of the file system structures are consistent and complete. In SFS, a check-point is carried out in two phases; first, it writes out all the dirty data and meta-data to the disk, and then updates the superblock in a special fixed location on the disk. The superblock keeps the root address of the meta-data, the position in the last written segment and time-stamp. SFS can guarantee the atomic write of the superblock by alternating between writing it to two separate physical blocks on the disk. During re-mounting, SFS reads both copies of the superblock, compares their time stamps and uses the more recent one.

Frequent check-pointing can minimize data loss from crashes but can hinder normal system performance. Considering this trade-off, SFS performs a check-point in four cases: (a) every thirty seconds after creating a check-point, (b) when more than 20 segments (640 MB given a segment size of 32 MB) are written, (c) when performing *sync* or *fsync* operation, and (d) when the file system is unmounted.

4 Evaluation

4.1 Experimental Systems

Implementation: SFS is implemented based on NILFS2 [28] by retrofitting the in-memory and on-disk meta-data structures to support block grouping and cost-hotness segment cleaning. NILFS2 in the mainline Linux kernel is based on log-structured file system [32] and incorporates advanced features such as b-tree based block management for scalability and continuous snapshot [20] for ease of management.

Implementing SFS requires a significant engineering effort, despite the fact that it is based on the already existing NILFS2. NILFS2 uses b-tree for scalable block mapping and virtual-to-physical block translation in data address translation (DAT) meta-data file to support continuous snapshot. One technical issue of b-tree based block mapping is the excessive meta-data update overhead. If a leaf block in a b-tree is updated, its effect is always propagated up to the root node and all the corresponding virtual-to-physical entries in the DAT are also updated. Consequently, random writes entail a significant amount of meta-data updates — writing 3.2 GB with 4 KB I/O unit generates 3.5 GB of meta-data. To reduce this meta-data update overhead and support the check-point creation policy discussed in Section 3.7, we decided to cut off the continuous snapshot feature. Instead, SFS-specific fields are added to several meta-data structures: superblock, inode file (IFILE), segment usage file (SUFILE), and DAT file. Group hotness H_{g_i} is stored in the superblock and loaded at mounting for the iterative segment quantization. Per file write count W_f and the last modified time T_f^m are stored in the IFILE. The SUFILE contains information for hotness calculation and segment cleaning: U_s, H_s, $\sum_i T_{b_i}^m$ and $\sum_i W_{b_i}$. Per-block write count W_b and the last modified time T_b^m are stored in the DAT entry along with virtual-to-physical mapping. Of these, W_b and T_b^m are the largest, each being eight bytes long. Since the meta-data fields for continuous snapshot in the DAT entry have been removed, the size of the DAT entry in SFS is the same as that of NILFS2 (32 bytes). As a result of these changes, we reduce the runtime overhead of meta-data to 5%–10% for the workloads used in our experiments. In SFS, since a meta-data file is treated the same as a normal file with a special inode number, a meta-data file can also be cached in the page cache for efficient access.

Segment cleaning in NILFS2 is not elaborated to the state-of-the-art in academia. It takes simple *time-stamp policy* [28] that selects the oldest dirty segment as a victim. For SFS, we implemented the cost-hotness policy and segment cleaning triggering policy described in Section 3.6.

In our implementation, we used Linux kernel 2.6.37, and all experiments are performed on a PC using a 2.67 GHz Intel Core i5 quad-core processor with 4 GB of physical memory.

Target SSDs: Currently, the spectrum of SSDs available in the market is very wide in terms of price and performance; flash memory chips, RAM buffers, and hardware controllers all vary greatly. For this paper, we select three state-of-the-art SSDs as shown in Table 1. The high-end SSD is based on SLC flash memory and the rest are based on MLC. Hereafter, these three SSDs are referred to as *SSD-H*, *SSD-M*, and *SSD-L* ranging from high-end to low-end.

Figure 1 shows sequential vs. random write throughput of the three devices. The request sizes of random write whose bandwidth converges to that of sequential write are 16 MB, 32 MB, and 16 MB for SSD-H, SSD-M, and SSD-L, respectively. To fully exploit device performance, the segment size is set to 32 MB for all three devices.

Workloads: To study the impact of SFS on various workloads, we use a mixture of synthetic and real-world workloads. Two real-world file system traces are used in our experiments: OLTP database workload, and desktop workload. For OLTP database workload, the file system level trace is collected while running TPC-C [40]. The database server runs Oracle 11g DBMS and the load server runs Benchmark Factory [30] using TPC-C benchmark scenario. For desktop workload, we used RES from the University of California at Berkeley [31]. *RES* is a research workload collected for 113 days on a system consisting of 13 desktop machines of a research group. In addition, two traces of random writes with different distributions are generated as synthetic workloads: one with Zipfian distribution and the other with uniform random distribution. The uniform random write is the workload that shows the worst case behavior of SFS, since SFS tries to utilize the skewness in workloads during block grouping.

Since our main area of interest is in maximum write performance, write requests in the workloads are replayed as fast as possible in a single thread and throughput is measured at the application level. Native Command Queuing (NCQ) is enabled to maximize the parallelism in the SSD. In order to explore the system behavior on various disk utilizations, we sequentially filled the SSD with enough dummy blocks, which are never updated after creation, until the desired utilization is reached. Since the amount of the data block update is the same for a workload regardless of the disk utilization, the amount of the meta-data update is also the same. Therefore, in our experiment results, we can directly compare performance metrics for a workload regardless of the disk utilization.

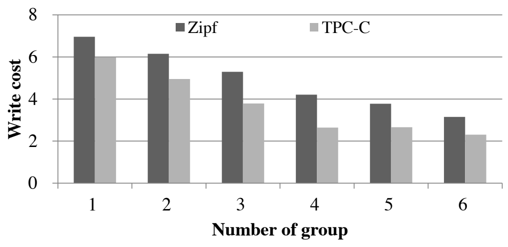


Figure 5: Write cost vs. number of group. Disk utilization is 85%.

Write Cost: To write new data in SFS, a new segment is generated by the segment cleaner. This cleaning process will incur additional read and write operations for the live blocks being segment-cleaned. Therefore, the write cost of data should include the implicit I/O cost of segment cleaning as well as the pure write cost of new data. In this paper, we define the write cost W_c to compare the write cost induced by the segment cleaning. It is defined by three component costs – the write cost of new data W_c^{new}, the read and the write cost of the data being segment-cleaned, R_c^{sc} and W_c^{sc} – as follows:

$$W_c = \frac{W_c^{new} + R_c^{sc} + W_c^{sc}}{W_c^{new}}$$

Each component cost is defined by division of the amount of I/O by throughput. Since the unit of write in SFS is always a large sequential chunk, we choose the maximum sequential write bandwidth in Table 1 for throughputs of W_c^{sc} and W_c^{new}. Meanwhile, since the live blocks being segment-cleaned are assumed to be randomly located in a victim segment, the 4 KB random read bandwidth in Table 1 is selected for the read throughput of R_c^{sc}. Throughout this paper, we measured the amount of I/O while replaying the workload trace and thus calculated the write cost for a workload.

4.2 Effectiveness of SFS Techniques

As discussed in Section 3, the key techniques of SFS are (a) on writing block grouping, (b) iterative segment quantization, and (c) cost-hotness segment cleaning. To

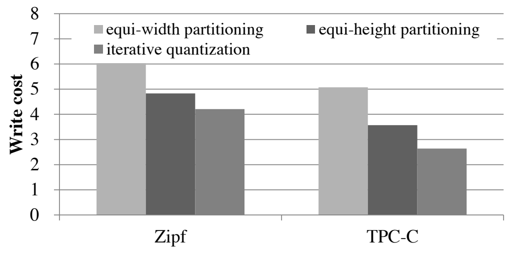


Figure 6: Write costs of quantization schemes. Disk utilization is 85%.

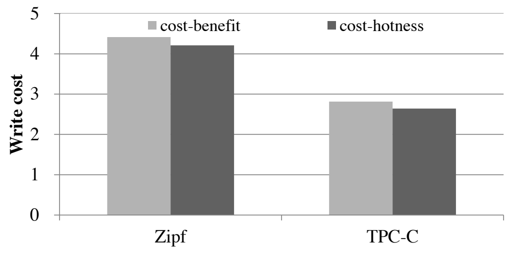


Figure 7: Write cost vs. segment cleaning scheme. Disk utilization is 85%.

examine how each technique contributes to the overall performance, we measured the write costs of Zipf and TPC-C workload under 85% disk utilization on SSD-M.

First, to verify how the block grouping is effective, we measured the write costs by varying the number of groups from one to six. As shown in Figure 5, we can observe that the effect of block grouping is considerable. When the blocks are not grouped (i.e. the number of groups is 1), the write cost is fairly high: 6.96 for Zipf and 5.98 for TPC-C. Even when the number of groups increases to two or three, no significant reduction in write cost is observed. However, when the number of groups reaches four the write costs of Zipf and TPC-C workloads significantly drop to 4.21 and 2.64, respectively. In the case of five or more groups, the write cost reduction is marginal. The additional groups do not help much when there are already enough groups covering hotness distribution, but may in fact increase the write cost. Since more blocks can be deferred due to insufficient blocks in a group, this could result in more blocks being written without grouping when creating a checkpoint.

Next, we compared the write cost of the different segment quantization schemes across four groups. Figure 6 shows that our iterative segment quantization reduces the write costs significantly. The equi-width partitioning scheme has the highest write cost; 143% for Zipf and 192% for TPC-C over the iterative segment quantization. The write costs of the equi-height partitioning scheme are 115% for Zipf and 135% for TPC-C over the iterative segment quantization.

Finally, to verify how cost-hotness policy affects performance, we compared the write cost of cost-hotness policy and cost-benefit policy with the iterative segment quantization for four groups. As shown in Figure 7, cost-hotness policy can reduce the write cost by approximately 7% over for both TPC-C and Zipf workload.

4.3 Performance Evaluation

4.3.1 Write Cost and Throughput

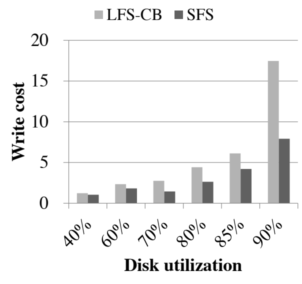
To show how SFS and LFS perform against various workloads with different write patterns, we measured their write costs and throughput for two synthetic workloads and two real workloads, and presented the performance results in Figure 8 and 9. For LFS, we implemented the cost-benefit cleaning policy in our code base (hereafter LFS-CB). Since throughput is measured at the application level, it includes the effects of the page cache and thus can exceed the maximum throughput of each device. Due to space constraints, only the experiments on SSD-M are shown here. The performance of SFS on different devices is shown in Section 4.3.3.

First, let us explain how much SFS can improve the write cost. It is clear from Figure 8 that SFS significantly reduces the write cost compared to LFS-CB. In particular, the relative write cost improvement of SFS over LFS-CB gets higher as disk utilization increases. Since there is not enough time for the segment cleaner to reorganize blocks under high disk utilization, our *on writing* data grouping shows greater effectiveness. For the TPC-C workload which has high update skewness, SFS reduces the write cost by 77.4% under 90% utilization. Although uniform random workload without skewness is a worst case workload, SFS reduces the write cost by 27.9% under 90% utilization. This shows that SFS can effectively reduce the write cost for a variety of workloads.

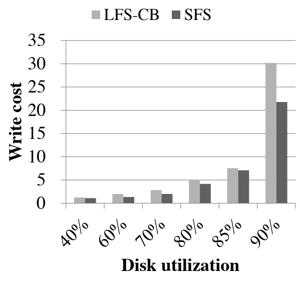
To see if the lower write costs in SFS will result in higher performance, throughput is also compared. As Figure 9 shows, SFS improves throughput of the TPC-C workload by 151.9% and that of uniform random workload by 18.5% under 90% utilization. It shows that the write cost reduction in SFS actually results in performance improvement.

4.3.2 Segment Utilization Distribution

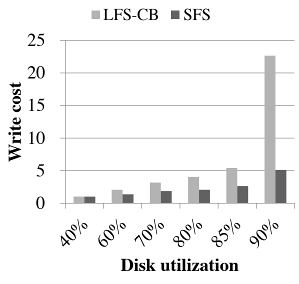
To further study why SFS significantly outperforms LFS-CB, we also compared the segment utilization distribution of SFS and LFS-CB. Segment utilization is calculated by dividing the number of live blocks in the segment by the number of total blocks per segment. After running a workload, the distribution is computed by measuring the utilizations of all non-dummy seg-



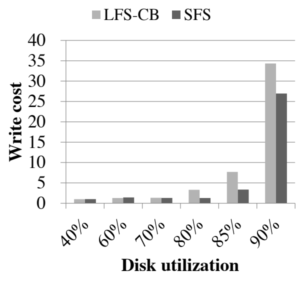
(a) Zipf



(b) Uniform random

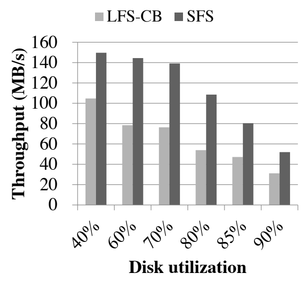


(c) TPC-C

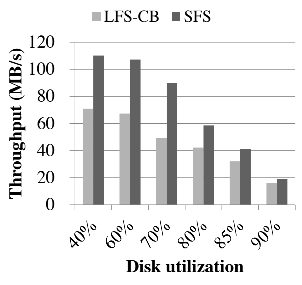


(d) RES

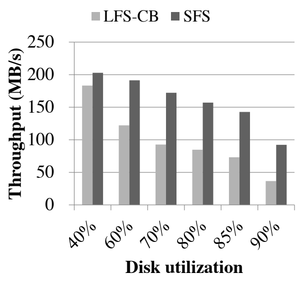
Figure 8: Write cost vs. disk utilization with SFS and LFS-CB on SSD-M.



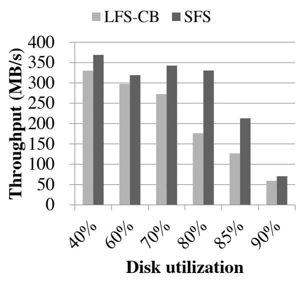
(a) Zipf



(b) Uniform random

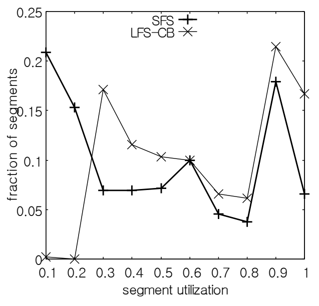


(c) TPC-C

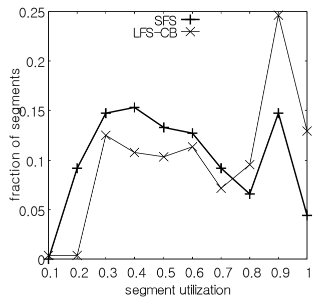


(d) RES

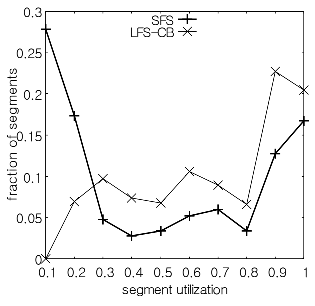
Figure 9: Throughput vs. disk utilization with SFS and LFS-CB on SSD-M.



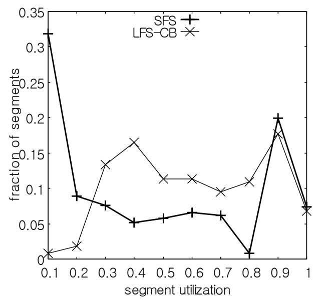
(a) Zipf



(b) Uniform random

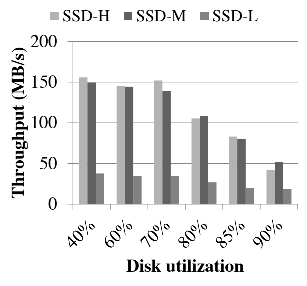


(c) TPC-C

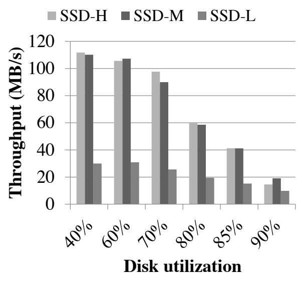


(d) RES

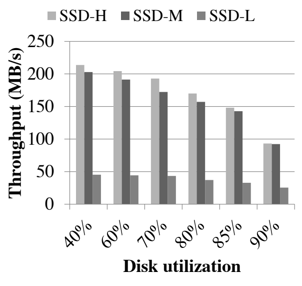
Figure 10: Segment utilization vs. fraction of segments. Disk utilization is 70%.



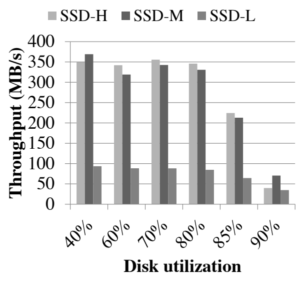
(a) Zipf



(b) Uniform random



(c) TPC-C



(d) RES

Figure 11: Throughput vs. disk utilization with SFS on different devices.

ments on the SSD. Since SFS continuously re-groups data blocks according to hotness, it is likely that a sharp bimodal distribution is formed. Figure 10 shows the segment utilization distribution when disk utilization is 70%. We can see the obvious bimodal segment distribution in SFS for all workloads except for the skewless uniform random workload. Even in the uniform random workload, the segment utilization of SFS is skewed to lower utilization. Under such bimodal distribution, the segment cleaner can select as victims those segments with few live blocks. For example, as shown in Figure 10a, SFS will select a victim segment with 10% utilization, while LFS-CB will select a victim segment with 30% utilization. In this case, the number of live blocks of a victim in SFS is just one-third of that in LFS-CB, thus the segment cleaner copies only one-third the amount of blocks. The reduced cleaning overhead results in a significant performance gap between SFS and LFS-CB. This experiment shows that SFS forms a sharp bimodal distribution of segment utilization by data block grouping, and reduces the write cost.

4.3.3 Effects of SSD Performance

In the previous sections, we showed that SFS can significantly reduce the write cost and drastically improve throughput on SSD-M. As shown in Section 2.2, SSDs have various performance characteristics. To see whether SFS can improve the performance on various SSDs, we compared throughput of the same workloads on SSD-H, SSD-M, and SSD-L in Figure 11. As shown in Table 1, SSD-H is ten-fold more expensive than SSD-L, the maximum sequential write performance of SSD-H is 4.5 times faster than SSD-L, and the 4 KB random write performance of SSD-H is more than 2,500 times faster than SSD-L. Despite the fact that these three SSDs show such large variances in performance and price, Figure 11 shows that SFS performs regardless of the random write performance. The main limiting factor is the maximum sequential write performance. This is because, except for updating superblock, SFS always generates large sequential writes to fully exploit the maximum bandwidth of SSD. The experiment shows that SFS can provide high performance even on mid-range or low-end SSD only if sequential write performance is high enough.

4.4 Comparison with Other File Systems

Up to now, we have analyzed how SFS performs under various environments with different workloads, disk utilization, and SSDs. In this section, we compared the performance of SFS using three other file systems, each with different block update policies: LFS-CB for *logging policy*, ext4 [25] for *in-place-update policy*, and btrfs [10] for *no-overwrite policy*. To enable btrfs' SSD

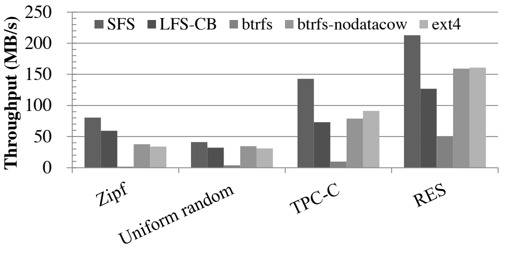
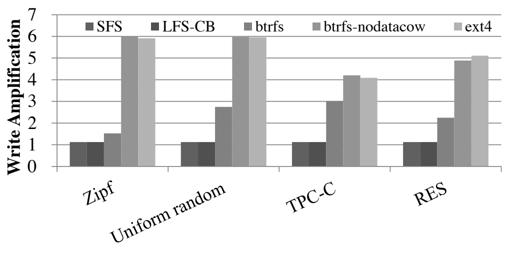


Figure 12: Throughput under different file systems.

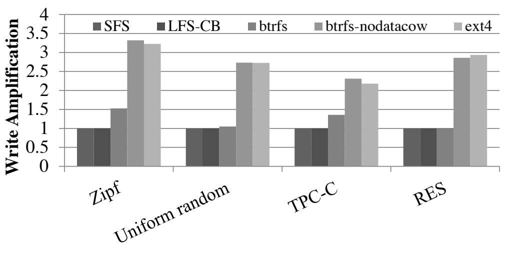
optimization, btrfs was mounted in SSD mode. The in-place-update mode of btrfs is also tested with the `nodatacow` option enabled to further analyze the behavior of btrfs (hereafter btrfs-nodatacow). Four workloads were run on SSD-M with 85% disk utilization. To obtain the sustained performance, we measured 8 GB writing after 20 GB writing for aging.

First, we compared throughput of the file systems in Figure 12. SFS significantly outperforms LFS-CB, ext4, btrfs, and btrfs-nodatacow for all four workloads. The average throughputs of SFS are higher than those of other file systems: 1.6 times for LFS-CB, 7.3 times for btrfs, 1.5 times for btrfs-nodatacow, and 1.5 times for ext4.

Next, we compared the write amplification that represents the garbage collection overhead inside SSD. We collected I/O traces issued by the file systems using `blktrace` [8] while running four workloads, and the traces were run on an FTL simulator, which we implemented, with two FTL schemes – (a) FAST [24] as a representative hybrid FTL scheme and (b) page-level FTL [17]. In both schemes, we configure a large block 32 GB NAND flash memory with 4 KB page, 512 KB block, and 10% over-provisioned capacity. Figure 13 shows write amplifications in FAST and page-level FTL for the four workloads processed by each file system. In all cases, write amplifications of log-structured file systems, SFS and LFS-CB, are very low: 1.1 in FAST and 1.0 in page-level FTL on average. This indicates that both FTL schemes generate 10% or less additional writings. Log-structured file systems collect and transform random writes at file level to sequential writes at LBA level. This results in optimal switch merge [24] in FAST and creates large chunks of contiguous invalid pages in page-level FTL. In contrast, in-place-update file systems, ext4 and btrfs-nodatacow, have the largest write amplification: 5.3 in FAST and 2.8 in page-level FTL on average. Since in-place-update file systems update a block in-place, random writes at file-level result in random writes at LBA-level. This contributes to high write amplification. Meanwhile, because btrfs never overwrites a block and allocates a new block for every update, it is likely to lower the average write amplification: 2.8 in FAST and

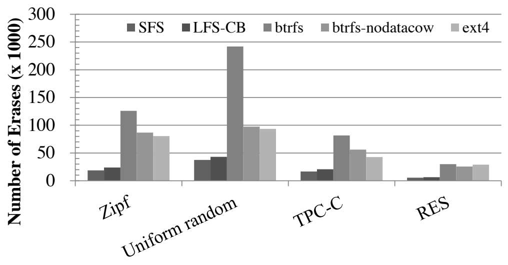


(a) FAST

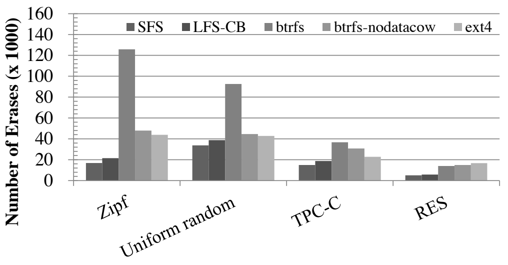


(b) Page Mapping

Figure 13: Write amplification with different FTL schemes.



(a) FAST



(b) Page Mapping

Figure 14: Number of erases with different FTL schemes.

1.2 in page-level FTL on average.

Finally, we compared the number of block erases that determine the lifespan of SSD in Figure 14. As can be expected from the write amplification analysis, the number of block erases in SFS and LFS-CB are significantly lower than in all others. Since the segment cleaning overhead of SFS is lower than that of LFS-CB, the number of block erases in SFS is smallest: LFS-CB incurs totally 20% more block erases in FAST and page-level FTL. Erase counts of overwrite file systems, ext4 and btrfs-nodatacow, are significantly higher than that of SFS. In total, ext4 incurs 3.1 times more block erases in FAST and 1.8 times more block erases in page-level FTL. Similarly, total erase counts of btrfs-nodatacow are 3.4 times higher in FAST and 2.0 times higher in page-level FTL. Interestingly, btrfs incurs the largest number of block erases: in total, 6.1 times more block erases in FAST and 3.8 times more block erases in page-level FTL, and in worst case 7.5 times more block erases than SFS. Although the no-overwrite scheme in btrfs incurs lower write amplification compared to ext4 and btrfs-nodatacow, btrfs shows large overhead to support copy-on-write and manage fragmentation [21, 46] induced by random writes at file-level.

In summary, the erase count of the in-place-update file system is high because of high write amplification. That of the no-overwrite file system is also high due to the number of write requests from the file system, even at relatively low write amplification. The majority of the overhead comes from supporting no-overwrite and handling fragmentation in the file system. Fragmentation of the no-overwrite file system under random write is a widely known problem [21, 46]: successive random writes eventually move all blocks into arbitrary positions, and this makes all I/O access random at the LBA level. Defragmentation, which is similar to segment cleaning in a log-structured file system, is implemented [21, 1] to reduce the performance problem of fragmentation. Similarly to segment cleaning, it also has additional overhead to move blocks. In case of log-structured file systems, if we carefully choose segment size to be aligned with the clustered block size, write amplification can be minimal. In this case, the segment cleaning overhead is the major overhead that increases the erase count. SFS is shown to drastically reduce the segment cleaning overhead. It can also be seen that the write amplification and erase count of SFS are significantly lower than for all other file systems. Therefore, SFS can significantly increase the lifetime as well as the performance of SSDs.

5 Related Work

Flash memory based storage systems and log-structured techniques have received a lot of interests in both academia and industry. Here we only present the papers most related to our work.

FTL-level approaches: There are many FTL-level approaches to improve random write performance.

Among hybrid FTL schemes, FAST [24] and LAST [22] are representative. FAST [24] enhances random write performance by improving the log area utilization with flexible mapping in log area. LAST [22] further improves FAST [24] by separating random log blocks into hot and cold regions to reduce the full merge cost. Among page-level FTL schemes, DAC [13] and DFTL [14] are representative. DAC [13] clusters data blocks of the similar write frequencies into the same logical group to reduce the garbage collection cost. DFTL [14] reduces the required RAM size for the page-level mapping table by using dynamic caching. FTL-level approaches exhibit a serious limitation in that they depend almost exclusively on LBA to decide sequentiality, hotness, clustering, and caching. Such approaches deteriorate when a file system adopts a *no-overwrite* block allocation policy.

Disk-based log-structured file systems: There is much research to optimize log-structured file systems on conventional hard disks. In the *hole plugging* method [44], the valid blocks in victim segments are overwritten to the *holes*, i.e. invalid blocks, in other segments with a few invalid blocks. This reduces the copying cost of valid blocks in segment cleaning. However, this method is beneficial only under a storage media that allows in-place updates. Matthews et al. [26] proposed the *adaptive method* that combines cost-benefit policy and hole-plugging. It first estimates the cost of cost-benefit policy and hole-plugging respectively, and then adaptively selects the policy with the lower cost. However, their cost model is based on the performance characteristics of HDD, seek and rotational delay. WOLF [42] separates hot pages and cold pages into two different segment buffers according to the update frequency of data pages, and writes two segments to disk at once. This system works well only when hot pages and cold pages are roughly half and half, so that they can be separated into two segments. HyLog [43] uses a hybrid approach: logging for hot pages to achieve high write performance and overwrite for cold pages to reduce the segment cleaning cost. In HyLog, it is critical to estimate the ratio of hot pages to determine the update policy. However, similar to the adaptive method, its cost model is based on the performance characteristics of HDD.

Flash-based log-structured file systems: In embedded systems with limited CPU and main memory, specially designed file systems that directly access raw flash devices are commonly used. To handle the unique characteristics of flash memory including no in-place-update, wear-leveling and bad block management, these systems take the log-structured approach. JFFS2 [45], YAFFS2 [47], and UBIFS [41] are widely used flash-based log-structured file systems. In terms of segment cleaning, each uses a turn-based selection algorithm [45, 47, 41] that incorporates wear-leveling into the segment cleaning process. This consists of two phases, namely X and Y turns. In the X turn, it selects a victim segment using greedy policy without considering wear-leveling. During the Y turn, it probabilistically selects a full valid segment as a victim block for wear-leveling.

6 Conclusion and Future Work

In this paper, we proposed a next generation file system for SSD, SFS. It takes a log-structured approach which transforms the random writes at the file system into the sequential writes at the SSD, thus achieving high performance and also prolonging the lifespan of the SSD. Also, in order to exploit the skewness in I/O workloads, SFS captures the hotness semantics at file block level and utilizes these in grouping data eagerly on writing. In particular, we devised an iterative segment quantization algorithm for correct data grouping and also proposed the cost-hotness policy for victim segment selection. Our experimental evaluation confirms that SFS considerably outperforms existing file systems such as LFS, ext4, and btrfs, and prolongs the lifespan of SSDs by drastically reducing block erase count inside the SSD.

Another interesting question is the applicability of SFS for HDD. Though SFS was originally designed for targeting primarily for SSDs, its key techniques are agnostic to storage devices. While random write is more serious in SSD since it hurts the lifespan as well as performance, it hurts performance also in HDD due to increased seek-time. We did preliminary experiments to see if SFS is beneficial in HDD and got promising experimental results. As future work, we intend to explore the applicability of SFS for HDD in greater depth.

Acknowledgements

We thank the anonymous reviewers and our shepherd Keith Smith for their feedback and comments, which have substantially improved the content and presentation of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2011-0020520). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0027613).

References

[1] Linux 3.0. `http://kernelnewbies.org/Linux_3.0`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceeding of*

USENIX 2008 Annual Technical Conference, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[3] S. Akyürek and K. Salem. Adaptive block rearrangement. *ACM Trans. Comput. Syst.*, 13:89–121, May 1995.

[4] D. G. Andersen and S. Swanson. Rethinking Flash in the Data Center. *IEEE Micro*, 30:52–54, July 2010.

[5] L. Barroso. Warehouse-scale computing. In Keynote in the SIGMOD'10 conference, 2010.

[6] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: block-reORGanization for self-optimizing storage systems. In *Proccedings of the 7th conference on File and storage technologies*, pages 183–196, Berkeley, CA, USA, 2009. USENIX Association.

[7] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 23–23, Berkeley, CA, USA, 1995. USENIX Association.

[8] blktrace. `http://linux.die.net/man/8/blktrace`.

[9] L. Bouganim, B. n Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '09, 2009.

[10] Btrfs. `http://btrfs.wiki.kernel.org`.

[11] S. D. Carson. A system for adaptive disk rearrangement. *Softw. Pract. Exper.*, 20:225–242, March 1990.

[12] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[13] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for plash memory. *Softw. Pract. Exper.*, 29:267–290, March 1999.

[14] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 229–240, New York, NY, USA, 2009. ACM.

[15] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):pp. 100–108, 1979.

[16] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[17] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[18] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48:366–375, May 2002.

[19] J. Kim, S. Seo, D. Jung, J. Kim, and J. Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *To Appear in IEEE Transactions on Computers*, 2011.

[20] R. Konishi, K. Sato, and Y. Amagai. Filesystem support for Continuous Snapshotting. `http://www.nilfs.org/papers/ols2007-snapshot-bof.pdf`, 2007. Ottawa Linux Symposium 2007 BOFS material.

[21] J. Kára. Ext4, btrfs, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, pages 99–111, 2009.

[22] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42:36–42, October 2008.

[23] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 55–66, New York, NY, USA, 2007. ACM.

[24] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6, July 2007.

[25] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of of the Linux Symposium*, June 2007.

[26] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adap-

tive methods. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[27] S. Mitchel. *Inside the Windows 95 File System*. O'Reilly and Associates, 1997.

[28] NILFS2. `http://www.nilfs.org/`.

[29] R. Paul. Panelists ponder the kernel at Linux Collaboration Summit. `http://tinyurl.com/d7sht7`, 2009.

[30] QuestSoftware. Benchmark Factory for Databases. `http://www.quest.com/benchmark-factory/`.

[31] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of USENIX Annual Technical Conference*, ATEC '00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992.

[33] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 1991.

[34] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of USENIX Winter 1993 Technical Conference*, page 405–420, 1993.

[35] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[36] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[37] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High performance solid state storage under Linux. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[38] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise Version 1.0. `http://www.snia.org/sites/default/files/SSS_PTS_Enterprise_v1.0.pdf`, 2011.

[39] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 9–14, New York, NY, USA, 2009. ACM.

[40] Transaction Processing Performance Council. TPC Benchmark C. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`.

[41] UBIFS. Unsorted Block Image File System. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

[42] J. Wang and Y. Hu. A Novel Reordering Write Buffer to Improve Write Performance of Log-Structured File Systems. *IEEE Trans. Comput.*, 52:1559–1572, December 2003.

[43] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[44] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14:108–136, February 1996.

[45] D. Woodhouse. JFFS : The Journalling Flash File System. In *Proceedings of the Ottowa Linux Symposium*, 2001.

[46] M. Xie and L. Zefan. Performance Improvement of Btrfs. In *LinuxCon Japan*, 2011.

[47] YAFFS. Yet Another Flash File System. `http://www.yaffs.net/`.

[48] ZFS. `http://opensolaris.org/os/community/zfs/`.