Design and Implementation of a Log-Structured File System for Flash-Based Solid State Drives

Changwoo Min, Sang-Won Lee, and Young Ik Eom

Abstract—Even in modern SSDs, the disparity between random and sequential write bandwidth is more than 10-fold. Moreover, random writes can shorten the limited lifespan of SSDs because they incur more NAND block erases per write. To overcome the problems of random writes, we propose a new file system, SFS, for SSDs. SFS is similar to the traditional log-structured file system (LFS) in that it transforms all random writes at the file system level to sequential ones at the SSD level, as a way to exploit the maximum write bandwidth of the SSD. But, unlike the traditional LFS, which performs hot/cold data separation *on segment cleaning*, SFS takes a new *on writing* data grouping strategy. When data blocks are to be written, SFS puts those with similar update likelihood into the same segment for sharper bimodal distribution of segment utilization, and thus aims at minimizing the inevitable segment cleaning overhead that occurs in any log-structured file system. 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. Our experiments on SSDs show that SFS outperforms LFS by up to 2.5 times in terms of throughput. In comparison to modern file systems, SFS drastically reduces the block erase count inside SSDs by up to 23.3 times. Although SFS was targeted for SSDs, its data grouping *on writing* would also work well in HDDs. To confirm this, we repeated the same set of experiments over HDDs, and found that SFS is quite promising in HDDs: although the slow random reads in HDDs make SFS slightly less effective, SFS still outperforms LFS by 1.7 times.

Index Terms—Log-structured file systems, segment cleaning, random write, solid state drives, hard disk drives

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 [1]. Even worse, the ever-increasing bit density for higher capacity in NAND flash chips has resulted in a sharp drop in the number of program/erase cycles from 10 K to 5 K for the last two years [2]. Meanwhile, previous work [3], [4] 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 even after random writes are stopped, because random writes cause more data pages in NAND flash blocks to be copied elsewhere and erased. Therefore, the lifespan of SSDs can be drastically reduced by random writes.

Not surprisingly, researchers have devoted much effort to resolve these problems. Most of the work has been focused on the *flash translation layer* (FTL)—an SSD firmware emulating an HDD by hiding the complexity of NAND flash memory. Some studies [5], [6] improved random write performance by providing more efficient logical to physical address mapping. Meanwhile, other studies [7], [6] proposed 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 [8]–[10] in which every write to the same file block is always redirected to a new LBA. Several new database storage schemes are proposed to improve random write performance taking into account the performance characteristics of SSDs [11], [12]. However, general applications cannot benefit from such database specific optimizations.

In this paper, we propose a novel file system, SFS, that can improve random write performance and extend the lifetime of SSDs. Basically, SFS is very close to LFS [13]. LFS is known to be very promising in SSD because it writes all modifications to storage sequentially in a log-like structure. Moreover, unlike HDD, the fast uniform random read performance in SSD can substantially mitigate the overhead of random reads in LFS. However, the segment cleaning overhead in LFS can still severely degrade the performance [14], [15] and shorten the lifespan of SSD. This is because, at every segment cleaning, a large number of pages need to be copied to secure a large empty chunk for sequential writes. Although the excessive cleaning overhead in LFS could be somewhat remedied by careful victim selection, previous studies [16]–[19] have shown that the cleaning overhead even using the improved selection scheme is still high under large segment size. This situation is problematic especially with SSD, where maximum write performance can be sustained only by large segment

• *C. Min, S.W. Lee, and Y.I. Eom are with the College of Information & Communication Engineering, Sungkyunkwan University, Suwon, Gyeonggi-do 440-746, Korea. E-mail: {multics69, swlee, yieom}@skku.edu.*

Manuscript received 30 Apr. 2012; revised 14 Feb. 2013; accepted 16 Apr. 2013. Date of publication 23 Apr. 2013; date of current version 07 Aug. 2014. Recommended for acceptance by P. McDaniel.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.97

size (32 MB in our experiments). To overcome this limitation of LFS with SSD, we investigate how to take advantage of the performance characteristics of SSD and the skewness in I/O patterns. To summarize, SFS can sustain the maximum write bandwidth of SSD with large segment size while minimizing the segment cleaning overhead.

This paper makes the following specific contributions:

- We introduce the design principles for SSD-based file systems. They should exploit the performance characteristics of SSD and directly utilize file block level statistics. In fact, the architectural differences between SSD and HDD result 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, 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 the proper selection of grouping criteria. To determine the grouping criteria, we propose an *iterative segment quantization* algorithm. We also propose *cost-hotness policy* for victim segment selection. Our eager data grouping will colocate frequently updated blocks in the same segment; thus most blocks in that segment 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 moving 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 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.
- Finally, though SFS was designed mainly for SSDs, its key techniques are agnostic to storage devices. To verify this, we evaluate SFS on HDD. Our experimental results show that SFS can also effectively improve the random write performance of HDDs.

The rest of this paper is organized as follows. Section 2 overviews the characteristics of SSD and I/O workloads. Section 3 elaborates the design of SFS, and Section 4 shows the evaluation. Related work is described in Section 5. 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 (i.e., block erase before write), and (3) limited program/erase cycles—roughly 100 K for a single-level cell (SLC) and roughly 5 K to 10 K for a multi-level cell (MLC).

A typical SSD is composed of host interface logic (SATA, USB, and PCI Express), an array of NAND flash memories, and an SSD controller. FTL, run by SSD controller, emulates 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 [20]: (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 to extend the lifespan of SSD.

Much research has been carried out on FTL to improve the performance and extend the lifetime of SSD [21], [5]–[7]. 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. Due to the coarse-grained mapping, the mapping table is small enough to be kept in memory entirely. Unfortunately, this results in higher garbage collection overhead. In contrast, since a *page-level FTL* scheme manages a fine-grained page-level mapping table, it results in lower garbage collection overhead. However, such fine-grained mapping requires a large mapping table on RAM. To overcome such technical difficulties, *hybrid FTL* schemes [21], [5], [7] 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 [6] extends the page-level mapping by selectively caching page-level mapping table entries on RAM.

2.2 Imbalance between Random and Sequential Write Performance in SSDs

Write performance in SSDs is highly workload dependent and is eventually limited by the garbage collection performance of FTL. Previous work [3], [4], [12], [22], [23] 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. This is because 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* [5]. In page-level FTL, as it tends to fragment blocks evenly, garbage collection has large copying overhead [24].

To improve garbage collection performance, SSD combines several blocks striped over multiple NAND flash memories into a *clustered block* [25]. The purpose 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, the random write updates and invalidates a clustered block as a whole. Therefore, a *switch merge* [21] that has the lowest overhead occurs in the hybrid FTLs. Similarly, in the page-level FTLs, empty blocks with no live pages are selected as victims for garbage collection. This results in that the random write performance

TABLE 1
Specification Data of the Flash Devices. List Price Is as of September 2011

	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

converges with the 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 MLC 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 8 GB is measured. Fig. 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.

2.3 Skewness in I/O Workloads

Many researchers have pointed out that I/O workloads have non-uniform access frequency distribution [11], [26]–[31]. A disk-level trace of personal workstations exhibits a high locality of references in that 90% of the writes go to the 1% of blocks [26]. Roselli et al. [27] analyzed file system traces collected from four different groups of machines, and they found that files tend to be either read-mostly or write-mostly and the writes show substantial locality. And, the distribution of the write frequency in on-line transaction processing (OLTP) TPC-C workload is also highly skewed: 29% of writes go to 1.6% of pages [11]. Analysis of Bhadkamkar et al. [28] also confirms that the top 20% most frequently accessed blocks contribute to a substantially large (45-66%) percentage of total access. Fig. 2 depicts the cumulative write frequency distribution of three real workloads: an IO trace collected by ourselves while running TPC-C [32] using Oracle DBMS (TPC-C), a research group trace (RES), and a web server trace equipped with Postgres DBMS (WEB) collected by Roselli et al. [27]. Such skewness in I/O workloads 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 I/O workloads 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, victim selection policy, and crash recovery.

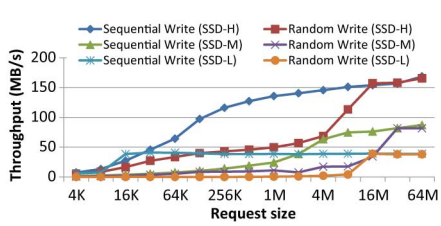

Fig. 1. Sequential vs. random write throughput.

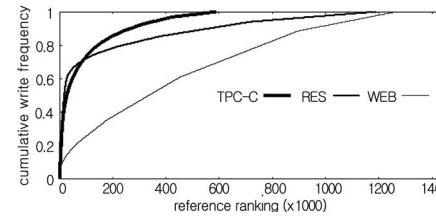

Fig. 2. Cumulative write frequency distribution.

3.1 SFS: Design of SSD-Based File Systems

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 segment utilization. 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 of 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 [33] and ext4 [34] always overwrite a dirty file block to the same LBA so that the same LBA ever corresponds to the file block unless the file frees it. 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 overwrite of the same LBA address. In fact, most FTL research [5]–[7], [35] 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 [9], and WAFL [8], where dirty file blocks are written to new LBAs. These systems are known to have better scalability, reliability, and manageability [36]. In those systems, however, because the unwritten

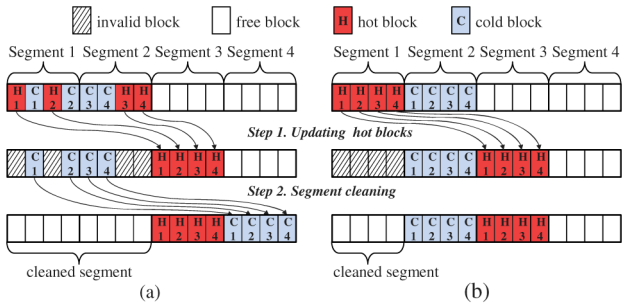

Fig. 3. Hot/cold data block placement and segment cleaning: (a) cold blocks are colocated with hot blocks in the same segment, and (b) hot and cold blocks are separated into different segments.

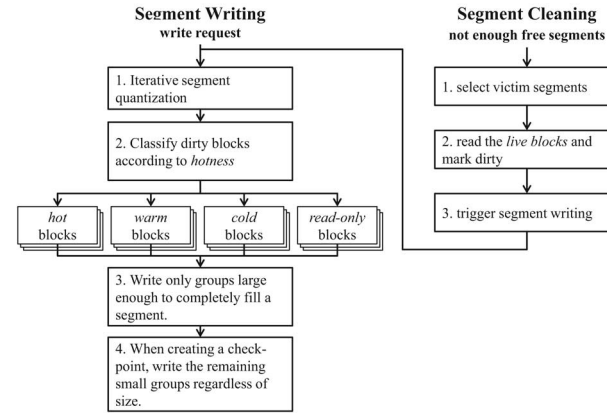

Fig. 4. Segment writing and cleaning process in SFS.

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. The 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 leads to improve the file system performance regardless of whether the unwritten assumption holds or not and regardless of how the underlying SSDs perform on random writes.

Write bandwidth maximization by sequentialized writes to SSD: In Section 2.2, we showed that the throughput of the random write becomes equal to that of the sequential write 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 the victim 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 [16]–[18]. Fig. 3 illustrates two examples of the segment cleaning procedure, and shows why careful data block placement is important. Let us assume that all hot blocks are updated (Step 1). In Fig. 3(a), a segment cleaner should move four cold blocks to secure empty segments (Step 2). On the other hand, in Fig. 3(b), since all hot blocks in Segment 1 are already invalidated, there is no need to move blocks at segment cleaning (Step 2). It shows that, if hot data and cold data are grouped into separate segments, the segment utilization distribution becomes bimodal: most of the segments are almost either full or empty of live blocks. Therefore, because the segment cleaner can almost always work with nearly empty segments, the cleaning overhead will be drastically reduced.

To form a bimodal distribution, LFS uses a cost-benefit policy [13] for segment cleaning that prefers cold segments to hot segments. However, previous studies [16]–[19] show that even the cost-benefit policy performs poorly under the large segment size (e.g., 8 MB), because the increased segment size makes it harder to find nearly empty segments. With SSD, the cost-benefit policy encounters a dilemma: small segment size enables LFS to form a bimodal distribution, but small random writes caused by the small segment severely degrades write performance of SSD. Instead of separating the data *lazily on segment cleaning* after writing them regardless of their hotness, SFS classifies data *proactively on writing* using file block level statistics, as well as on segment cleaning. In such eager data grouping, since segments are already composed of homogeneous data with similar update likelihood, the segment cleaning overhead will be significantly reduced. In particular, the I/O skewness commonly found in many real workloads will make this more attractive.

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 Fig. 4. Because the read operation is the 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 brevity, it is assumed throughout this paper that there are four segment groups: hot, warm, cold, and read-only groups. The second step is to calculate the block hotness for each dirty 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. The third step is to fill a segment with blocks belonging to the same group. If the number of blocks in a group is not enough to completely fill a segment, the segment writing of the group is deferred until the group grows to completely fill a segment. This eager grouping of file blocks according to the hotness 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. Meanwhile, upon a check-point for crash recovery, SFS unconditionally flushes all the groups to the disk, regardless of their size.

Segment cleaning in SFS consists of three steps: select victim segments, read the live blocks from the 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 the segment (i.e., cost) and the segment hotness.

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

3.3 Measuring Hotness

In SFS, *hotness* is used as a measure of how likely the data is to be updated. Although it is difficult to estimate data hotness without prior knowledge on future access pattern, SFS exploits both the skewness and the temporal locality in the I/O workload 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* can generally be 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 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 the update likelihood of a segment is determined by the live blocks contained in the segment, the segment hotness should be a representative value of the block hotness of the live blocks. Let us consider a graph whose x-axis and y-axis represent the age and the write count, respectively.

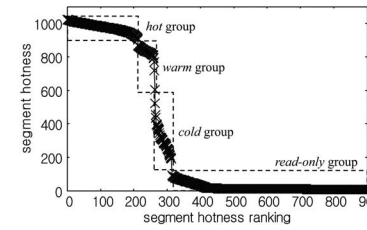

Fig. 5. Example of segment quantization.

On this graph, the block hotness can be defined as the slope for a line passing the origin and the particular (*age, write count*) point. Therefore, the segment hotness can also be defined as the slope of the best-fitting line for the (*age, write count*) points of the live blocks. The best-fitting line can be obtained by using the principle of least squares [37]. Since y-interception of the line should be zero by our definition of the hotness, the slope is simply $\frac{\bar{y}}{\bar{x}} = \frac{\text{mean of } y}{\text{mean of } x}$ according to the method of least squares [37]. Therefore, we define the hotness of a segment H_s as follows:

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

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. The computational cost of H_s is negligible, since we can incrementally calculate H_s without checking the liveness of blocks in the segment: 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. We will elaborate on meta-data management for hotness in Section 4.1.

3.4 Segment Quantization

To minimize the segment cleaning overhead, it is crucial for SFS to properly group blocks according to hotness. The effectiveness of block grouping is determined by the number of groups and the grouping criteria. They should reflect the hotness distribution. In fact, improper criteria may colocate blocks from different groups into the same segment, thus deteriorating the effectiveness of grouping. Moreover, given improper number of groups, it is hard to find the proper criteria. In this subsection, we will introduce our solution to find the proper criteria. The effect of the number of the groups on performance will be discussed in Section 4.2.

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 sub-ranges and equi-width partitioning makes each group have an equal number of segments. In Fig. 5, the segment hotness distribution is computed by measuring the hotness for all segments on the disk after running TPC-C workload under

70% file system 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 may be 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 k-means clustering [38], our iterative segment quantization partitions segments into k groups and tries to find the centers of natural groups through an iterative refinement approach. Moreover, unlike k-means clustering, because our algorithm repetitively clusters a gradually changing data set (i.e. segment hotness) at every segment writing, we can expedite the convergence by choosing the initial centers of the current run to those of the previous run. 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.

Even though its computational overhead is linear 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). For faster convergence, SFS stores H_{g_i} in meta-data and reloads them at mounting. Although a few heuristic-driven approaches have been used for optimizing block placement in storage systems [35], [39], [28], to our knowledge, SFS is the first file system which exploits machine learning algorithm in classifying blocks.

3.5 Segment Writing

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. As illustrated in Fig. 4, 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 whose hotness is closest to the block hotness.

To prevent blocks in different groups being colocated in the same segment, SFS writes only the groups large enough to completely fill a segment. 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

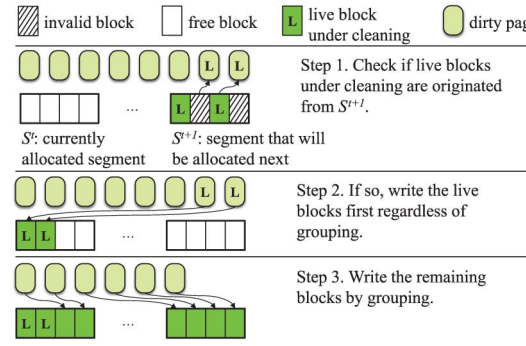

Fig. 6. Preventing data loss during segment cleaning.

SFS initiates a *check-point*, every dirty block including the deferred blocks should be immediately written to segment. In this case, we take a best-effort approach: writing out as many blocks as possible group-wise, and 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, in most cases except when updating a superblock, SFS always generates large sequential and well-aligned write requests that are optimal for SSD.

Meanwhile, the live blocks under segment cleaning should be carefully treated. Writing the blocks under segment cleaning can also be deferred if the corresponding group is not large enough. However, this deferred write can cause those not-yet-written blocks to be lost upon system crashes, if the segments to which the block originally belonged are already used for writing other blocks. To cope with such data loss, we propose two techniques. 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. Fig. 6 illustrates an example of this scheme. Let S^t denote the segment lastly allocated and S^{t+1} denote the segment to be allocated next. Before segment writing, SFS first checks whether there are not-yet-written blocks under cleaning that originate in S^{t+1} (Step 1). If such blocks exist, SFS writes those blocks to S^t regardless of their grouping so as to avoid overwriting the originating segment of each block (Step 2). Then, SFS writes the remaining blocks by grouping (Step 3). This guarantees that the segment-cleaned blocks are never lost against a system crash or a sudden power off, because each block always has its on-disk copy. The LRF allocation scheme increases the opportunity for a segment-cleaned block to be written by the block grouping.

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* [13] and *cost-benefit policy* [13], [40]. Greedy policy always selects segments with the smallest number of live blocks, hoping to reclaim as much space as possible with the least copying out overhead. Cost-benefit policy prefers cold

segments to hot segments when the number of live blocks is equal, because the cold data tends to remain unchanged for a long time before it becomes invalidated. Even though it is critical to estimate how long a segment remains unchanged, cost-benefit policy simply uses 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:

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

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). Since all segment usage information is mostly cached in memory, disk I/O is rarely required.

In SFS, the segment cleaner is invoked when the file system utilization exceeds a *water-mark*. The watermark for our experiments is set to 95% of the file system 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. [41], yet this could be seamlessly integrated with SFS.

3.7 Crash Recovery

Like the traditional LFS [13], SFS uses two mechanisms to recover from the file system inconsistency: *check-point*, which defines consistent states of the file system, and *roll-forward*, which is used to recover information written since the last check-point. Frequent check-pointing can minimize the roll-forward time after crashes but can hinder normal system performance. Considering this trade-off, SFS performs the check-point in four cases: (a) every thirty seconds after creating a check-point, (b) when more than 20 segments are written to disk, (c) when *sync* or *fsync* operation is performed, and (d) when unmounting the file system.

4 Evaluation

4.1 Experimental Systems

Implementation: We implemented SFS based on NILFS2 [42], which extends the log-structured file system [13] to support continuous snapshot [43] for ease of management. We retrofitted the in-memory and on-disk meta-data structures of NILFS to support block grouping and cost-hotness segment cleaning.

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 between the file offset to the virtual sector address. It translates the virtual sector address to the physical sector address by using the data address translation (DAT) meta-data file. It always assigns a new virtual sector address for every block update to support the continuous snapshot. Updating a leaf block in a b-tree 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. By turning off the continuous snapshot, we need to update only one virtual-to-physical entry in the DAT for a block update. 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. Of these, W_b and T_b^m are the largest, each being eight bytes long. Since the meta-data fields for continuous snapshot have been removed, the size of the DAT entry in SFS remains 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. 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 takes simple *time-stamp policy* [42] that selects the oldest dirty segment as a victim. For SFS, we implemented the cost-hotness policy and segment cleaning triggering policy.

In our implementation, we used Linux kernel 2.6.37 on a PC using a 2.67 GHz Intel Core i5 quad-core processor with 4 GB of physical memory. All experiments except for those with btrfs are performed on the kernel: we use the btrfs in the latest kernel version at this writing, 3.7.1, for fair comparison, because btrfs is one of the most actively developing file systems.

Target SSDs: 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. As Fig. 1 shows, 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: 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 [32] on Oracle 11g DBMS and the load server runs Benchmark Factory [44]. For desktop workload, we used RES [27], which is collected for 113 days on 13 desktop machines of a research group. In addition, two traces of random writes with different distributions are generated as synthetic workloads: Zipfian distribution and uniform random distribution. The uniform random write is the

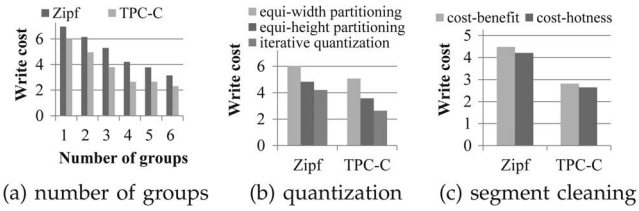

Fig. 7. Effectiveness of SFS techniques.

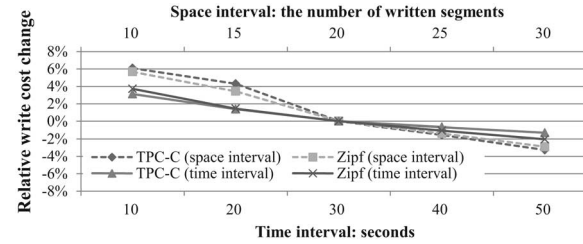

Fig. 8. Write cost vs. check-point intervals.

workload that shows the worst case behavior of SFS, since SFS tries to utilize the skewness in workloads during block grouping.

Since one of main goal of SFS is to achieve the maximum write bandwidth of the storage devices, write requests are replayed as fast as possible in a single thread. Although the write requests are replayed with a single thread, our evaluation result would achieve high IO parallelism in the storage layer, since all the write requests by the re-player are cached in the page cache and then flushed to the storage in bulk regularly by the file system. Throughput is measured at the application level and Native Command Queuing (NCQ) is enabled to maximize the parallelism in the SSD. To explore the system behavior on various file system 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 file system utilization, the amount of the meta-data update is also the same. Therefore, we can directly compare performance metrics for a workload regardless of the file system utilization.

Write Cost: To write new data, 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 in Table 1. Since the unit of write is always a large sequential chunk, we choose the maximum sequential write bandwidth for throughputs of W_c^{sc} and W_c^{new}. Meanwhile, since the live blocks are assumed to be randomly located in a victim segment, the 4 KB random read bandwidth is selected for the throughput of R_c^{sc}. We measured the amount of I/O while replaying the workload trace and thus calculated the write cost for the 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 examine how each technique contributes to the performance, we measured the write costs of Zipf and TPC-C workload under 85% file system 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 Fig. 7(a), we can observe that the effect of block grouping is considerable. When the blocks are not grouped (i.e. the number of groups is one), the write cost is fairly high: 6.96 for Zipf and 5.98 for TPC-C. Even for two or three groups, 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. For 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.

Next, we compared the write cost of the different quantization schemes across four groups. Fig. 7(b) 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 our scheme. The write costs of the equi-height partitioning scheme are 115% for Zipf and 135% for TPC-C over our scheme.

Then, we compared the write cost of cost-hotness policy and cost-benefit policy with the iterative segment quantization for four groups. As shown in Fig. 7(c), cost-hotness policy can reduce the write cost by approximately 7% over both TPC-C and Zipf workload.

Finally, to evaluate the effect of check-point interval on the performance, we measured the write cost by varying both time and space interval. Fig. 8 presents the relative write cost change to our baseline configuration, 30 seconds and 20 segments. As observed in Fig. 8, as the time and space intervals increase, the write costs moderately decrease across all the intervals (6.1% increase at most and 3.3% decrease at least). One interesting point to note in Fig. 8 is that the write costs are more sensitive to the space interval, and this is because the check-points are mostly triggered by the space interval in our experiments. Throughout the rest of this paper, we will use the default checkpoint interval of 30 seconds and 20 segments.

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 all four workloads in Figs. 9 and 10. 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

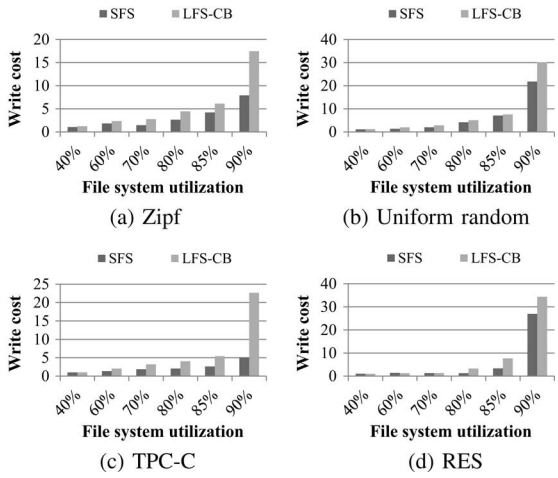

Fig. 9. Write cost vs. file system utilization with SFS and LFS-CB on SSD-M.

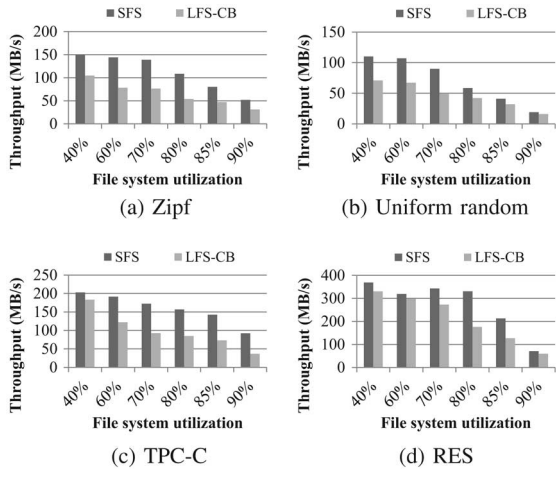

Fig. 10. Throughput vs. file system utilization with SFS and LFS-CB on SSD-M.

shown here. The performance of SFS on different devices is shown in Section 4.3.3.

First, it is clear from Fig. 9 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 the utilization increases. Since there is not enough time for the segment cleaner to reorganize blocks under high utilization, our *on writing* data grouping shows greater effectiveness. For the TPC-C workload with high update skewness, SFS reduces the write cost by 77.4% under 90% utilization. For skewless uniform random workload, SFS reduces the write cost by 27.9% under 90% utilization. This shows that SFS can effectively reduce the write cost for various workloads.

To see if the lower write costs in SFS will result in higher performance, throughput is also compared. As Fig. 10 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 analyze why SFS significantly outperforms LFS-CB, we also compared the segment utilization distribution which is a

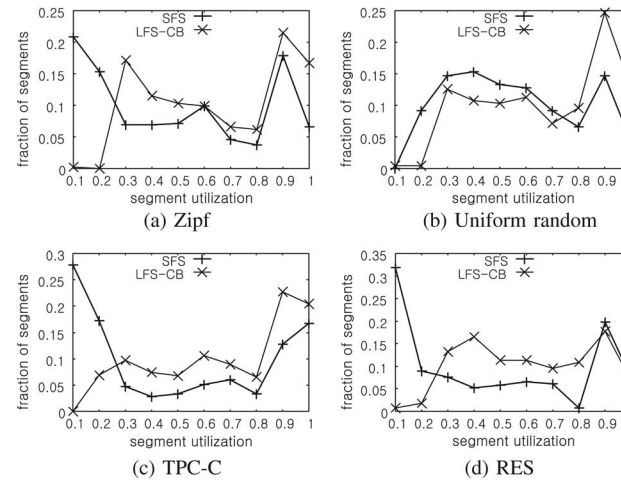

Fig. 11. Segment utilization vs. fraction of segments when segment size is 32 MB.

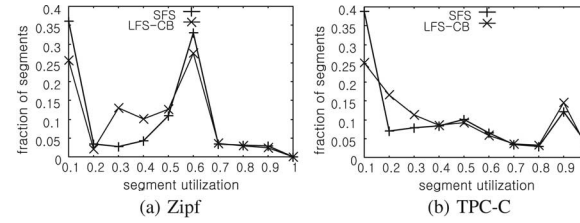

Fig. 12. Segment utilization vs. fraction of segments when segment size is 2 MB.

fraction of live blocks in a segment. After running a workload, the distribution is computed by measuring the utilizations of all non-dummy segments. Fig. 11 shows the segment utilization distribution when file system utilization is 70% and segment size is 32 MB. Since SFS continuously re-groups data blocks according to hotness on writing and segment cleaning, it is likely that a sharp bimodal distribution is formed. We can see the obvious bimodal segment distribution in SFS for all workloads except for the skewless uniform random workload. Even in the case, 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 Fig. 11(a), SFS will select a victim segment with 10% utilization, while LFS-CB will select a victim segment with 30% utilization. In this case, since SFS moves only one-third blocks during segment cleaning, the segment cleaning overhead of SFS is significantly lower than that of LFS-CB. This experiment shows that SFS forms a sharp bimodal distribution of segment utilization by data block grouping, and reduces the write cost.

As we discussed in Section 3.1, the cost-benefit policy used in LFS-CB is less effective under large segment size. To verify how the small segment affect the segment utilization distribution and write cost, we run TPC-C and Zipf workloads when the segment size is 2 MB, which is the same size used by Rosenblum and Ousterhout [13]. Fig. 12 shows that LFS-CB forms far sharper bimodal distribution than in 32 MB segment size. However, as shown in Fig. 1, in SSD-M, the sustained random write performance in 2 MB is about 11 times lower than that in 32 MB, and thus the write costs of SFS and LFS-CB are surged to 7.7 and 4.0 times for Zipf and 5.9 and 3.5 times

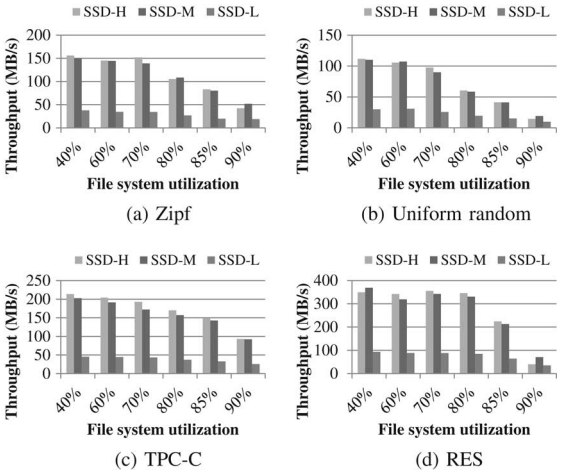

Fig. 13. Throughput vs. file system utilization with SFS on different devices.

for TPC-C, respectively. From this, we know that, although LFS-CB is quite effective in forming bimodal distribution with small segment size (e.g., 2 MB), it has its limitations in achieving high skewness under large segment size (e.g., 32 MB), which is a prerequisite in SSD for sustaining maximum write bandwidth. In contrast, SFS can achieve sharper bimodal segment distribution even with the segment size large enough to sustain maximum write bandwidth.

4.3.3 Effects of SSD Performance

The internal hardware and software architectures vary significantly among SSDs [20], and thus each SSD exhibits different performance dynamics [3], [4], [25]. To verify 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 Fig. 13. As shown in Table 1, 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, Fig. 13 shows that SFS performs well 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 if sequential write performance is high enough.

4.4 Comparison with Other File Systems

In this section, we compared the performance of SFS with three other file systems, each with different block update policies: LFS-CB for *logging policy*, ext4 [34] for *in-place-update policy*, and btrfs [10] for *no-overwrite policy*. To enable btrfs' SSD optimization, btrfs was mounted in SSD mode. Though JFFS2 [45], YAFFS2 [46], and UBIFS [47] are popular log-structured file systems for flash storage in wide use, we did not compare SFS with them because they are designed to work only on raw flash devices. Four workloads were run on SSD-M with 85% file system utilization. To obtain the sustained performance, we measured 8 GB writing after 20 GB writing for aging.

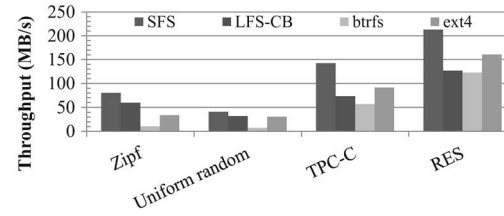

Fig. 14. Throughput under different file systems.

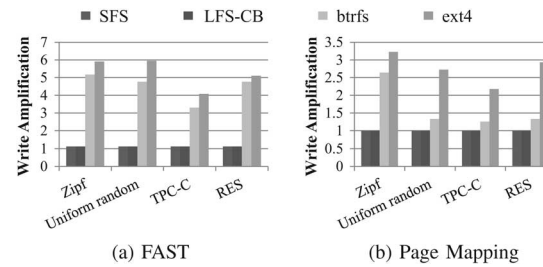

Fig. 15. Write amplification with different FTL schemes.

First, we compared throughput of the file systems in Fig. 14. SFS significantly outperforms other file systems for all four workloads. The average throughputs of SFS are higher than those of the others: 1.6 times for LFS-CB, 2.4 times for btrfs, and 1.5 times for ext4.

Next, we compared the write amplification in Fig. 15, which is the ratio of the number of physical data page writes inside SSD to the number of logical data page writes by a file system. We collected I/O traces issued by the file systems using `blktrace` [48] while running four workloads, and the traces were run on an FTL simulator, which we implemented with two FTL schemes: (a) FAST [5] as a representative hybrid FTL scheme and (b) page-level FTL [40]. 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. As Fig. 15 shows, in all cases, write amplifications of log-structured file systems, SFS and LFS-CB, are lowest: 1.1 in FAST and 1.0 in page-level FTL on average. The LBA-level sequential writing results in an optimal switch merge [5] in FAST and creates large chunks of contiguous invalid pages in page-level FTL. In contrast, the in-place-update file system, ext4, has the largest write amplifications: 5.3 in FAST and 2.8 in page-level FTL on average. Since the random writes at the file level result in random writes at the LBA level, this contributes to high write amplification. Meanwhile, because btrfs allocates a new block for every update, it is likely to lower the average write amplification: 4.5 in FAST and 1.6 in page-level FTL on average.

Finally, we compared block erase counts that determine the lifespan of SSD in Fig. 16. The number of block erases in SFS is smallest: LFS-CB incurs totally 20% more block erases in both FTL schemes. Erase counts of the overwrite file systems 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. Interestingly, btrfs incurs the largest number of block erases: in total, 8.9 times more block erases in FAST and 4.9 times more block erases in page-level FTL, and, in the worst case, 23.3 times more block erases than SFS. As shown in Fig. 17, btrfs generates 2.2 times more page writes

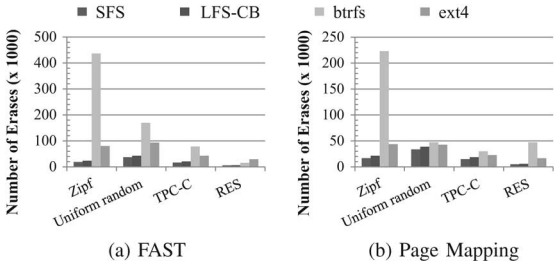

Fig. 16. Number of erases with different FTL schemes.

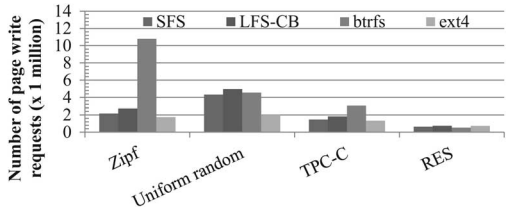

Fig. 17. Number of page writes on different file systems.

than SFS. This excessive page writes in btrfs can be explained by its intrinsic overhead to support copy-on-write and manage fragmentation induced by random writes at the file level [49]–[51].

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 policy and handling fragmentation in the file system. 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 those of all other file systems. Therefore, SFS can significantly increase the lifetime as well as the performance of SSDs.

4.5 Applicability of SFS to HDDs

Though SFS was originally designed to primarily target SSDs, its key techniques are agnostic to the types of storage devices. While random write is more serious in SSD since it hurts the lifespan as well as performance, it also hurts performance in HDD as a result of the increased seek-time. Therefore, SFS can be useful in mitigating the random write overhead in HDDs. In this subsection, we explored the applicability of SFS to HDDs. We evaluated SFS on a mid-range HDD spinning at 7,200 RPM with a maximum sequential write throughput of $109.6\ \mathrm{MB/s}$ and a 4 KB random read throughput of $0.4\ \mathrm{MB/s}$. To compare with the experimental results on SSDs, we used the same segment size and the same number of groups.

First, to evaluate how SFS and LFS-CB perform on HDD, we measured their throughput under various file system utilizations. As Fig. 18 shows, SFS outperforms LFS-CB in all cases. Under 90% utilization, SFS improves throughput of Zipf and uniform random workload by 70.9% and 37.5%, respectively.

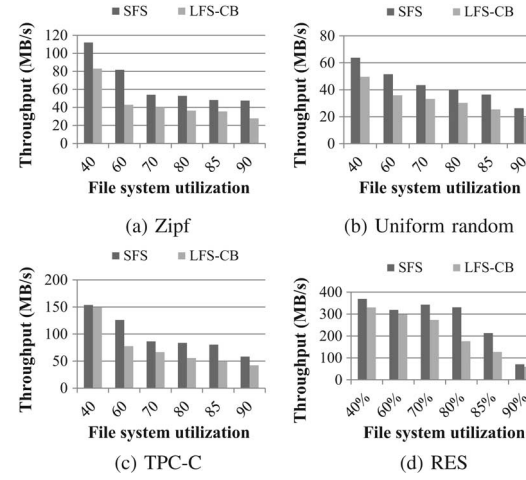

Fig. 18. Throughput vs. file system utilization with SFS and LFS-CB on HDD-M.

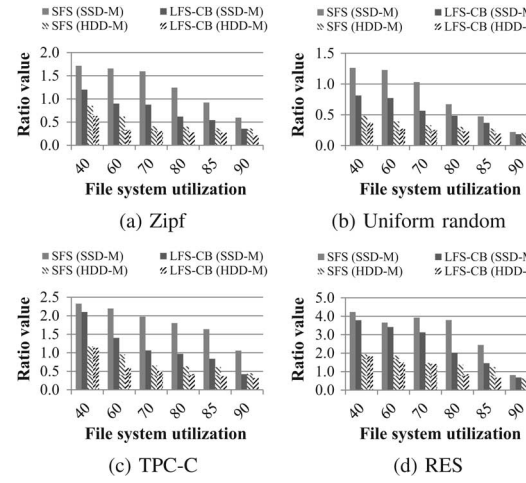

Fig. 19. Relative write throughput to each device's maximum sequential write throughput.

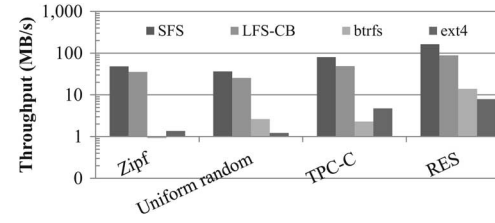

Fig. 20. Throughput under different file systems on HDD-M. (Y-axis is in log scale.)

Second, to compare how SFS and LFS-CB perform on HDD and SSD, we analyzed the ratio of write throughput in each file system to the maximum sequential write throughput of each device. Fig. 19 shows that the relative throughput of SSD-M is far higher than that of HDD-M. Since the random read speed on HDD is far slower than that of SSD, segment cleaning on HDD suffers more by the random reads than that on SSD.

Finally, in Fig. 20, we compare the throughput of SFS on HDD with LFS-CB, btrfs, and ext4. Under 85% utilization, the average throughputs of SFS are higher than those of other

file systems: 1.7 times for LFS-CB, 21.4 times for btrfs, and 21.6 times for ext4.

These results clearly show that SFS is also quite effective in HDDs by reducing the number of slow seeks, and that SFS is more effective in SSDs as a result of the far faster random read of SSDs that is incurred in segment cleaning.

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 research most related to our work.

FTL-level approaches: There are many FTL-level approaches to improve random write performance. Among hybrid FTL schemes, FAST [5] and LAST [7] are representative. FAST [5] enhances random write performance by improving the log area utilization with flexible mapping in log area. LAST [7] further improves FAST [5] by separating random log blocks into hot and cold regions to reduce the full merge cost. Among page-level FTL schemes, DAC [35] and DFTL [6] are representative. DAC [35] clusters data blocks with similar write frequencies into the same logical group to reduce the garbage collection cost. DFTL [6] 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 HDDs. In the *hole plugging* method [52], the valid blocks in victim segments are overwritten to the *holes*, i.e. invalid blocks, in other segments with a few invalid blocks. Although this reduces the copying cost in segment cleaning, it is beneficial only under a HDD that allows in-place updates. Matthews et al. [18] proposed an *adaptive method* that dynamically selects one of cost-benefit policy and hole-plugging. However, their cost model for the selection is based on the performance characteristics of HDD, seek and rotational delay. WOLF [39] separates hot pages and cold pages into two different segment buffers according to the update frequency, and writes the two segments to disk at once. This system works well only when hot and cold pages are roughly half and half, so that they can be separated into two segments. HyLog [19] uses logging hot pages for high write performance and overwriting cold pages for reducing the cleaning cost. However, its cost model for determining the update policy is based on the performance characteristics of HDD.

Flash-based log-structured file systems: Because the log-structured approach can naturally handle the no overwrite characteristics of flash, many log-structured file systems for flash memory such as JFFS2 [45], YAFFS2 [46], and UBIFS [47] have been widely used. In terms of segment cleaning, each uses a turn-based selection algorithm which is a variant of greedy policy incorporated with wear-leveling into the 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. While all these existing file systems for flash memory are designed to run on raw flash devices, SFS assumes block device interface.

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, 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. In addition, we evaluated SFS on HDD since our key techniques are agnostic to the types of storage devices. Our experimental results clearly show that SFS is also quite effective in HDDs because it turns the random writes into sequential ones, reducing the number of slow seeks.

There are many avenues for future work. First, Kim et al. [23] show that storage performance, especially SD card, indeed affects the performance of common applications on smartphones. Since the architecture of SD card is similar to that of low-end SSD, we expect that SFS is also beneficial to SD card. Next, most FTL in SSD takes variants of log-structured approach, and thus some SFS techniques, such as on writing data grouping and segment quantization, could be directly applicable to FTL to lower the write amplification.

Acknowledgment

Sang-Won Lee and Young Ik Eom are the corresponding authors of this paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (2012-0006423, 2010-0026511, and 2012R1A1A2A10044300).

References

[1] L. Barroso, "Warehouse-scale computing," in *Proc. Keynote Special Interest Group Manag. Data Conf. (SIGMOD'10)*, article no. 2, 2010.
[2] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 52–54, Jul./Aug. 2010.
[3] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput Syst.*, 2009, pp. 181–192.
[4] L. Bouganim, B. n. Jónsson, and P. Bonnet, "uFLIP: Understanding flash IO patterns," in *Proc. Conf. Innov. Data Syst. Res.*, 2009, pp. 1–12.
[5] 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. Embedded Comput. Syst.*, vol. 6, article no. 18, 2007.
[6] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2009, pp. 229–240.
[7] 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.*, vol. 42, pp. 36–42, 2008.
[8] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proc. USENIX Winter Tech. Conf.*, article no. 19, 1994.

[9] J. Bonwick. (2006). *ZFS: The Last Word in File Systems* [Online]. Available: http://tinyurl.com/36dhvvj
[10] C. Mason. (2007). *The btrfs Filesystem* [Online]. Available: http://tinyurl.com/bljvcau
[11] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2007, pp. 55–66.
[12] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and repairing write performance on flash devices," in *Proc. 5th Int. Workshop Data Manag. New Hardware*, 2009, pp. 9–14.
[13] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, pp. 26–52, 1992.
[14] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proc. USENIX Winter Conf.*, article no. 3, 1993.
[15] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File system logging versus clustering: A performance comparison," in *Proc. USENIX Tech. Conf. Proc.*, article no. 21, 1995.
[16] M. Rosenblum, "The design and implementation of a log-structured file system," PhD dissertation, Univ. California at Berkeley, Berkeley, CA, 1992.
[17] M. I. Seltzer, "File system performance and transaction support," PhD dissertation, Univ. California at Berkeley, Berkeley, CA, 1992.
[18] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," in *Proc. 16th ACM Symp. Oper. Syst. Princ.*, 1997, pp. 238–251.
[19] W. Wang, Y. Zhao, and R. Bunt, "HyLog: A high performance approach to managing disk layout," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 145–158.
[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.
[21] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[22] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High performance solid state storage under Linux," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–12.
[23] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 209–222.
[24] X.-Y. Hu and R. Haas, "The fundamental limit of flash random write performance: Understanding, analysis and performance modelling," IBM Research, Zurich, Res. Rep. RZ 3771, 2010.
[25] J. Kim, S. Seo, D. Jung, J. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 636–649, May 2012.
[26] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," in *Proc. USENIX Winter Tech. Conf.*, 1993, pp. 405–420.
[27] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 41–54.
[28] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for self-optimizing storage systems," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 183–196.
[29] S. Akyürek and K. Salem, "Adaptive block rearrangement," *ACM Trans. Comput. Syst.*, vol. 13, pp. 89–121, 1995.
[30] C. Ruemmler and J. Wilkes, "Disk Shuffling," Softw. Syst. Lab., Hewlett-Packard Lab., Tech. Rep. HPL-CSP-91-30, 1991.
[31] S. D. Carson, "A system for adaptive disk rearrangement," *Softw. Pract. Exp.*, vol. 20, pp. 225–242, 1990.
[32] Transaction Processing Performance Council. (2010). *TPC Benchmark C* [Online]. Available: http://www.tpc.org/tpcc/spec/tpcc_current.pdf
[33] S. Mitchel, *Inside the Windows 95 File System*. Sebastopol, CA, USA: O'Reilly and Associates, 1997.
[34] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, vol. 2, pp. 21–33.
[35] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw. Pract. Exp.*, vol. 29, pp. 267–290, 1999.
[36] R. Paul. (2009). *Panelists Ponder the Kernel at Linux Collaboration Summit* [Online]. Available: http://tinyurl.com/d7sht7
[37] W. Mendenhall, R. J. Beaver, and B. M. Beaver, *Introduction to Probability and Statistics*, 13th ed. Boston, MA, USA: Brooks/Cole, 2009.
[38] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *J. Roy. Stat. Soc. C*, vol. 28, no. 1, pp. 100–108, 1979.
[39] J. Wang and Y. Hu, "A novel reordering write buffer to improve write performance of log-structured file systems," *IEEE Trans. Comput.*, vol. 52, no. 12, pp. 1559–1572, Dec. 2003.
[40] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. USENIX Tech. Conf.*, 1995, pp. 13–13.
[41] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proc. USENIX Tech. Conf.*, 1995, pp. 23–23.
[42] R. Konishi. (2009). *The NILFS2 Filesystem: Review and Challenges* [Online]. Available: http://tinyurl.com/au4xqve
[43] R. Konishi, K. Sato, and Y. Amagai. (2007). *Filesystem Support for Continuous Snapshotting* [Online]. Available: http://tinyurl.com/bdufbnc, Ottawa Linux Symp. 2007 BOFS material.
[44] Quest Software. (2009). *Benchmark Factory for Databases* [Online]. Available: http://www.quest.com/benchmark-factory/
[45] D. Woodhouse, "JFFS: The journalling flash file system," in *Proc. Ottowa Linux Symp.*, 2001.
[46] C. Manning. (2010). *How YAFFS Works* [Online]. Available: http://tinyurl.com/amu5ta9
[47] A. Hunter. (2008). *A Brief Introduction to the Design of UBIFS* [Online]. Available: http://tinyurl.com/5c2anq
[48] J. Axboe, A. D. Brunelle, and N. Scott. (2006). *Blktrace(8)—Linux Main Page* [Online]. Available: http://linux.die.net/man/8/blktrace
[49] J. Kára, "Ext4, btrfs, and the others," in *Proc. Linux-Kongress Open-Solaris Developer Conf.*, 2009, pp. 99–111.
[50] M. Xie and L. Zefan, "Performance improvement of btrfs," in *Proc. LinuxCon Japan*, 2011.
[51] Linux Kernel Newbies. (2011). *Linux 3.0* [Online]. Available: http://kernelnewbies.org/Linux_3.0
[52] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, pp. 108–136, 1996.

Changwoo Min received the BS and MS degrees in computer science from Soongsil University, Seoul, South Korea, in 1996 and 1998, respectively. He is currently working toward the PhD degree at Sungkyunkwan University, Suwon, South Korea, and is a software engineer of Samsung Electronics, Korea. His research interests include storage system, virtualization, and parallel processing.

Sang-Won Lee received the PhD degree from the Computer Science Department, Seoul National University, Korea, in 1999. He is an associate professor with the College of Information & Communication Engineering, Sungkyunkwan University, Suwon, Korea. He was a research professor at Ewha Womans University and a technical staff at Oracle, Korea. His research interest includes flash-based database technology.

Young Ik Eom received the BS, MS, and PhD degrees from the Department of Computer Science and Statistics, Seoul National University, Korea, in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an associate professor at Dankook University, Yongin, Korea. He was also a visiting scholar with the Department of Information and Computer Science, the University of California, Irvine, from September 2000 to August 2001. Since 1993, he has been a professor at Sungkyunkwan University, Suwon, Korea. His research interests include storage system, virtualization, cloud system, and system securities.