ΔFTL: Improving SSD Lifetime via Exploiting Content Locality

Guanying Wu and Xubin He

Department of Electrical and Computer Engineering
Virginia Commonwealth University, Richmond, VA 23284, USA
{wug, xhe2}@vcu.edu

Abstract

NAND flash-based SSDs suffer from limited lifetime due to the fact that NAND flash can only be programmed or erased for limited times. Among various approaches to address this problem, we propose to reduce the number of writes to the flash via exploiting the content locality between the write data and its corresponding old version in the flash. This content locality means, the new version, i.e., the content of a new write request, shares some extent of similarity with its old version. The information redundancy existing in the difference (delta) between the new and old data leads to a small compression ratio. The key idea of our approach, named ΔFTL (Delta Flash Translation Layer), is to store this compressed delta in the SSD, instead of the original new data, in order to reduce the number of writes committed to the flash. This write reduction further extends the lifetime of SSDs due to less frequent garbage collection process, which is a significant write amplification factor in SSDs. Experimental results based on our ΔFTL prototype show that ΔFTL can significantly reduce the number of writes and garbage collection operations and thus improve SSD lifetime at a cost of trivial overhead on read latency performance.

Categories and Subject Descriptors C.4 [*PERFORMANCE OF SYSTEMS*]: Reliability, availability, and serviceability

General Terms Design, Reliability, Performance

Keywords SSD, Lifetime, NAND flash, Reliability, FTL

1. Introduction

Solid state drives (SSDs) exhibit good performance, particularly for random workloads, compared to traditional hard drives (HDDs). From a reliability standpoint, SSDs have no moving parts, no mechanical wear-out, and are silent and resistant to heat and shock. However, the limited lifetime of SSDs is a major drawback that hinders their deployment in reliability sensitive environments [3, 5]. As pointed out in [5], "endurance and retention (of SSDs) is not yet proven in the field" and integrating SSDs into commercial systems is "painfully slow". The reliability problem of SSDs mainly comes from the following facts. Flash memory must be erased before it can be written and it may only be programmed/erased for a limited times (5K to 100K) [10]. In addition, the out-of-place writes result in invalid pages to be discarded by garbage collection (GC). Extra writes are introduced in GC operations to move valid pages to a clean block [2] which further aggravates the lifetime problem of SSDs.

Existing approaches for this problem mainly focus on two perspectives: 1) to prevent early defects of flash blocks by wear-leveling techniques [6, 38]; 2) to reduce the number of write operations on the flash. For the later, various techniques are proposed including in-drive buffer management schemes [16, 18, 22, 48] to exploit the temporal or spatial locality; FTLs (Flash Translation Layer) [12, 17, 23, 24] to optimize the mapping policies or garbage collection schemes to reduce the write-amplification factor; or data deduplication [7, 11] to eliminate writes of existing content in the drive.

In this paper, we aim to efficiently solve this lifetime issue from a different aspect. We propose a new FTL scheme, ΔFTL, to reduce the write count via exploiting the content locality. The content locality has been observed and exploited in memory systems [13], file systems [8], and block devices [32, 49, 50]. Content locality means data blocks, either blocks at distinct locations or created at different time, share *similar contents*. We exploit the content locality that exists between the new (the content of update write) and the old version of page data mapped to the same logical address. This content locality implies the new version resembles the old to some extend, so that the difference (*delta*) between them can be compressed compactly. Instead of storing new data in its original form in the flash, ΔFTL stores the compressed deltas to reduce the number of writes.

Research contributions:

- We propose a novel FTL scheme, ΔFTL to extend SSD lifetime via exploiting the content locality. We describe how ΔFTL functionality can be achieved from the data structures and algorithms that enhance the regular page-mapping FTL.
- We propose techniques to alleviate the potential performance overheads of ΔFTL.
- We model ΔFTL's performance on extending SSD's lifetime via analytical discussions and outline the workload characteristics favored by ΔFTL.
- We evaluate the performance of ΔFTL under real-world workloads via simulation experiments. Results show that ΔFTL significantly extends SSD's lifetime by reducing the number of garbage collection operations at a cost of trivial overhead on read latency performance. Specifically, ΔFTL results in 33% to 58% of the baseline garbage collection operations; and the read latency is only increased by approximately 5%.

The rest of the paper is organized as follows: Section 2 gives an introduction to NAND flash-based SSDs and a brief survey of techniques to extent SSD's lifetime as well as techniques to leverage the content locality. In Section 3, we discuss the design of ΔFTL in detail. Analytical modeling of ΔFTL's performance for SSD lifetime enhancement is expanded in Section 4. The performance evaluation under real-world workloads is given in Section 5. We conclude this paper and outline the future work in Section 6.

2. Background and Related Work

2.1 NAND Flash-based SSDs

The NAND flash by itself exhibits relatively poor performance [46, 47]. The high performance of an SSD comes from leveraging a hierarchy of parallelism. At the lowest level is the *page*, which is the basic unit of I/O read and write requests in SSDs. Erase operations operate at the *block* level, which are sequential groups of pages. A typical value for the size of a block is 64 or 128 pages. Further up the hierarchy is the plane, and on a single die there could be several planes. Planes operate semi-independently, offering potential speed-ups if data is striped across several planes. Additionally, certain copy operations can operate between planes without crossing the I/O pins. An upper level of abstraction, the chip interfaces, free the SSD controller from the analog processes of the basic operations, i.e., read, program, and erase, with a set of defined commands. NAND interface standards includes ONFI [34], BA-NAND [34], OneNAND [36], LBA-NAND [42], etc. SSDs hides the underlying details of the chip interfaces and exports the storage space as a standard block-level disk via a software layer called *Flash Translation Layer* (FTL). FTL is a key component of an SSD in that it not only is responsible for managing the "logical to physical" address mapping but also works as a flash memory allocator, wear-leveler, and garbage collection engine. The mapping policy is mostly related to our work in this paper. The mapping policies of FTLs can be classified into two types: page-level mapping [2, 12], where a logical page can be placed onto any physical page; or block-level mapping [17, 23, 24], where the logical page LBA is translated to a physical block address and the offset of that page in the block. Page-level mapping is believed to be popular in modern SSD design [7, 11]. In this paper, our design augments the regular page-mapping FTL to support the delta-encoding of the newly written data.

2.2 Extending SSD's Lifetime

To extend the lifetime of SSDs, many designs have been proposed in the literature such as FTLs, cache schemes, hybrid storage materials, etc.

FTLs: For block-level mapping, several FTL schemes have been proposed to use a number of physical blocks to log the updates. Examples include FAST [24], BAST [23], SAST [17], and LAST [25]. The garbage collection of these schemes involves three types of merge operations, *full*, *partial*, and *switch* merge. The block-level mapping FTL schemes leverage the spatial or temporal locality in write workloads to reduce the overhead introduced in the merge operations. For page level mapping, DFTL [12] is proposed to cache the frequently used mapping table in the in-drive SRAM so as to improve the address translation performance as well as reduce the mapping table updates in the flash; μ-FTL [27] adopts the μ-tree on the mapping table to reduce the memory footprint. Two-level FTL [45] is proposed to dynamically switch between page-level and block-level mapping. Content-aware FTLs (CAFTL) [7, 11] implement the deduplication technique as FTL in SSDs to eliminate contents that are "exactly" the same across the entire drive. CAFTL requires complicated FTL design and implementation, e.g., a large finger-print store to facilitate content lookup and multi-layer mapping tables to locate logical addresses associated to the same content. Due to the limited computation power of the micro-processor inside SSDs, the complexity of deduplication via CAFTL is a major concern. On the other hand, our ΔFTL focuses on leveraging the similarity existing among old and new versions of data at the same logical address (as opposed to the entire drive), which brings a lightweight FTL design and implementation.

Cache schemes: A few in-drive cache schemes like BPLRU [22], FAB [16], CLC [18], and BPAC [48] are proposed to improve the sequentiality of the write workload sent to the FTL, so as to reduce the merge operation overhead on the FTLs. CFLRU [35] which works as an OS level scheduling policy, chooses to prioritize the clean cache elements when doing replacements so that the write operations can be reduced or avoided. Taking advantage of fast sequential performance of HDDs, Griffin [39] and I-CASH [49] are proposed to extend the SSD lifetime by caching SSDs with HDDs.

Heterogeneous material: Utilizing advantages of PCRAM, such as the in-place update ability and faster access, Sun *et al.* [41] describe a hybrid architecture to log the updates on PCRAM for flash. FlexFS [26], on the other hand, combines MLC and SLC as trading off the capacity and erase cycle.

Wear-leveling Techniques: Dynamic wear-leveling techniques, such as [38], try to recycle blocks of small erase counts. To address the problem of blocks containing cold data, static wear-leveling techniques [6] try to evenly distribute the wear over the entire SSD.

2.3 Exploiting the Content Locality

The content locality implies that the data in the system share similarity with each other. Such similarity can be exploited to reduce the memory or storage usage by delta-encoding the difference between the selected data and its reference. Content locality has been leveraged in various level of the system. In virtual machine environments, VMs share a significant number of *identical* pages in the memory, which can be deduplicated to reduce the memory system pressure. Difference engine [13] improves the performance over deduplication by detecting the *nearly* identical pages and coalesce them via in-core compression [30] into much smaller memory footprint. Difference engine detects similar pages based on hashes of several chucks of each page: hash collisions are considered as a sign of similarity. Different from difference engine, GLIMPSE [31] and DERD system [8] work on the file system to leverage similarity across files; the similarity detection method adopted in these techniques is based on Rabin fingerprints over chunks at multiple offsets in a file. In the block device level, Peabody [32] and TRAP-Array [50] are proposed to reduce the space overhead of storage system backup, recovery, and rollback via exploiting the content locality between the previous (old) version of data and the current (new) version. Peabody mainly focuses on eliminating duplicated writes, i.e., the update write contains the same data as the corresponding old version (silent write) or sectors at different location (coalesced sectors). On the other hand, TRAP-Array reduces the storage usage of data backup by logging the compressed XORs (delta) of successive writes to each data block. The intensive content locality in the block I/O workloads produces a small compression ratio on such deltas and TRAP-Array is significantly space-efficient compared to traditional approaches. I-CASH [49] takes the advantage of content locality existing across the entire drive to reduce the number of writes in the SSDs. I-CASH stores only the reference blocks on the SSDs while logs the delta in the HDDs.

Our approach ΔFTL is mostly similar to the idea of TRAP-Array [50] , which exploits the content locality between new and old version. The major differences are: 1) ΔFTL aims at reducing the number of program/erase (**P/E**) operations committed to the flash memory so as to extend SSD's lifetime, instead of reducing storage space usage involved in data backup or recovery. Technically, the history data are backed up in TRAP-Array while they are considered "invalid" and discarded in ΔFTL; 2) ΔFTL is an embedded software in the SSD to manage the allocation and de-allocation of flash space, which requires relative complex data structures and algorithms that are "flash-aware". It also requires that the computation complexity should be kept minimum due to limited micro-processor capability.

3. Design of ΔFTL

In this section, we first outline the architecture of ΔFTL and then depict its major components in detail.

3.1 Overview

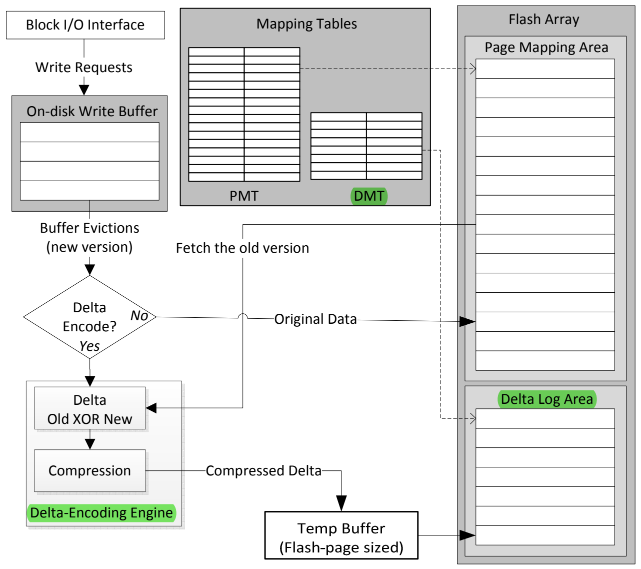

Figure 1. ΔFTL Overview

ΔFTL is designed as a flash management scheme that can store the write data in form of compressed deltas on the flash. Instead of devising from scratch, ΔFTL is rather an enhancement to the framework of the popular page-mapping FTL like DFTL [12]. Figure 1 gives an overview of ΔFTL and unveils its major differences from a regular page-mapping FTL:

- First, ΔFTL has a dedicated area, *Delta Log Area* (DLA), for logging the compressed deltas.
- Second, the compressed deltas must be associated with their corresponding old versions to retrieve the data. An extra mapping table, *Delta Mapping Table* (DMT), collaborates with *Page Mapping Table* (PMT) to achieve this functionality.

- Third, ΔFTL has a *Delta-encoding Engine* to derive and then compress the delta between the write buffer evictions and their old version on the flash. We have a set of dispatching rules (Section 3.2) determining whether a write request is stored in its original form or in its "delta-XOR-old" form. For the first case, the data is written to a flash page in page mapping area in its original form. For the later case, the delta-encoding engine derives and then compresses the delta between old and new. The compressed deltas are buffered in a flash-page-sized *Temp Buffer* until the buffer is full. Then, the content of the temp buffer is committed to a flash page in delta log area.

Details of the data structures and algorithms to implement ΔFTL are given in the following subsections.

3.2 Dispatching Policy: Delta Encode?

The content locality between the new and old data allows us to compress the delta, which has rich information redundancy, to a compact form. Writing the compressed deltas rather than the original data, would indeed reduce the number of flash writes. However, delta-encoding all data indiscriminately would cause overheads. First, if a page is stored in "delta-XOR-old" form, this page actually requires storage space for both delta and the old page, compared to only one flash page if in the original form. The extra space is provided by the over-provisioning area of the drive [2]. To make a trade-off between the over-provisioning resource and the number of writes, ΔFTL favors the data that are overwritten frequently. This policy can be interpreted intuitively with a simple example: in a workload, page data A is only overwritten once while B is overwritten 4 times. Assuming the compression ratio is 0.25, delta-encoding A would reduce the number of write by $3/4$ page (compared to the baseline which would take one page write) at a cost of $1/4$ page in the over-provision space. Delta-encoding B, on the other hand, reduces the number of write by $4 * (3/4) = 3$ pages at the same cost of space. Clearly, we would achieve better performance/cost ratio with such write "hot" data rather than the cold ones. The approach taken by ΔFTL to differentiate hot data from cold ones is discussed in Section 3.5.2. Second, fulfilling a read request targeting a page in "delta-XOR-old" form requires two flash page reads. This may have reverse impact on the read latency. To alleviate this overhead, ΔFTL avoids delta-encoding pages that are read intensive. If a page in "delta-XOR-old" form is found read intensive, ΔFTL will merge it to the original form to avoid the reading overhead. Again, the detailed approach is discussed in Section 3.5.2 in detail and evaluated in Section 5. Third, the delta-encoding process involves operations to fetch the old, derive delta, and compress delta. This extra time may potentially add overhead to the write performance (discussed in Section 3.3.2). ΔFTL must cease delta-encoding if it would degrade the write performance. To summarize, ΔFTL delta-encodes data that are *write-hot* but *read-cold* while ensuring the write performance is not degraded.

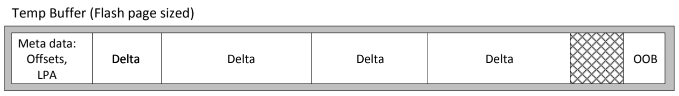

Figure 2. ΔFTL Temp Buffer

3.3 Write Buffer and Delta-encoding

The in-drive write buffer resides in the volatile memory (SRAM or DRAM) managed by an SSD's internal controller and shares a significant portion of it [16, 18, 22]. The write buffer absorbs repeated writes and improves the spatial locality of the output workload from it. We concentrate our effort on FTL design, which services write buffer's outputs, and adopt simple buffer management schemes like FIFO or SLRU [19] that are usual in disk drives. When buffer eviction occurs, the evicted write pages are dispatched according to our dispatching policy discussed above to either ΔFTL's *Delta-encoding Engine* or directly to the page mapping area. Delta-encoding engine takes the new version of the page data (i.e., the evicted page) and the corresponding old version in page mapping area, as its inputs. It derives the delta by XOR the new and old version and then compress the delta. The compressed delta are buffered in *Temp Buffer*. *Temp Buffer* is of the same size as a flash page. Its content will be committed to delta log area once it is full or there is no space for the next compressed delta. Splitting a compressed delta on two flash pages would involve in unnecessary complications for our design. Storing multiple deltas in one flash page requires meta-data, like LPA (logical page address) and the offset of each delta (as shown in Figure 2) in the page, to associate them with their old versions and locate the exact positions. The meta-data is stored at the MSB part of a page instead of attached after the deltas, for the purpose of fast retrieval. This is because the flash read operation always buses out the content of a page from its beginning [34]. The content of temp buffer described here is essentially what we have in flash pages of delta log area. Delta-encoding engine demands the computation power of SSD's internal micro-processor and would introduce overhead for write requests. We discuss the delta-encoding latency in Section 3.3.1 and the approach adopted by ΔFTL to control the overhead in Section 3.3.2.

3.3.1 Delta-encoding Latency

Delta-encoding involves two steps: to derive delta (XOR the new and old versions) and to compress it. Among many data compression algorithms, the lightweight ones are favorable for ΔFTL due to the limited computation power of the SSD's internal micro-processor. We investigate the latency of a few candidates, including Bzip2 [37], LZO [33],

Table 1. Delta-encoding Latency

Frequency(MHz)	304	619	934
Compression(μs)	89.5	44.0	29.1
Decompression(μs)	22.2	10.9	7.2

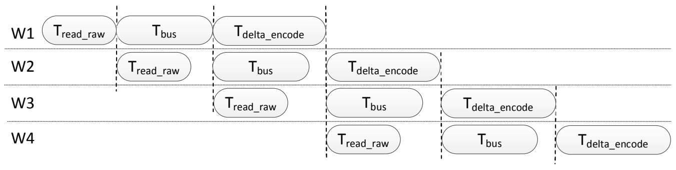

Figure 3. ΔFTL Delta-encoding Timeline

LZF [28], Snappy [9], and Xdelta [30], by emulating the execution of them on the ARM platform: the source codes are cross-compiled and run on the *SimpleScalar-ARM* simulator [29]. The simulator is an extension to SimpleScalar supporting ARM7 [4] architecture and we configured a processor similar to ARM®Cortex R4 [1], which inherits ARM7 architecture. For each algorithm, the number of CPU cycles is reported and the latency is then estimated by dividing the cycle number by the CPU frequency. We select LZF (LZF1X-1) from the candidates because it makes a good trade-off between speed and compression performance, plus a compact executable size. The average number of CPU cycles for LZF to compress and decompress a 4KB page is about 27212 and 6737, respectively. According to Cortex R4's write paper, it can run at a frequency from 304MHz to 934MHz. The latency values in μs are listed in Table 1. An intermediate frequency value (619MHz) is included along with the other two to represent three classes of micro-processors in SSDs.

3.3.2 Discussion: Write Performance Overhead

ΔFTL's delta-encoding is a two-step procedure. First, delta-encoding engine fetches the old version from the page mapping area. Second, the delta between the old and new data are derived and compressed. The first step consists of raw flash access and bus transmission, which exclusively occupy the flash chip and the bus to the micro-processor, respectively. The second step occupies exclusively the micro-processor to perform the computations. Naturally, these three elements, the flash chip, the bus, and micro-processor, forms a simple pipeline, where the delta-encoding procedures of a serial of write requests could be overlapped. An example of four writes is demonstrated in Figure 3, where T_{delta_encode} is the longest phase. This is true for a micro-processor of 304MHz or 619MHz assuming T_{read_raw} and T_{bus} take 25μs and 40μs (Table 3), respectively. A list of symbols used in this section is summarized in Table 2. For an analytical view of the write overhead, we assume there is a total number of n write requests pending for a chip. Among these requests, the percentage that is considered *compressible* according to

Table 2. List of Symbols

Symbols	**Description**
n	Number of pending write pages
P_c	Probability of compressible writes
R_c	Average compression ratio
T_{write}	Time for page write
T_{read_raw}	Time for raw flash read access
T_{bus}	Time for transferring a page via bus
T_{erase}	Time to erase a block
T_{delta_encode}	Time for delta-encoding a page
B_s	Block size (pages/block)
N	Total Number of page writes in the workload
T	Data blocks containing invalid pages (baseline)
t	Data blocks containing invalid pages (ΔFTL's PMA)
PE_{gc}	The number of P/E operations done in GC
F_{gc}	GC frequency
OH_{gc}	Average GC overhead
G_{gc}	Average GC gain (number of invalid pages reclaimed)
S_{cons}	Consumption speed of available clean blocks

Table 3. Flash Access Latency

Parameter	**Value**
Flash Read/Write/Erase	$25\mu s/200\mu s/1.5ms$
Bus Transfer Time	$40\mu s$

our dispatching policy is P_c and the average compression ratio is R_c. The delta-encoding procedure for these n requests takes a total time of: $MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode}) * n * P_c$ The number of page writes committed to the flash is the sum of original data writes and compressed delta writes: $(1 - P_c) * n + P_c * n * R_c$. For the baseline, which always outputs the data in their original form, the page write total is n. We define that the write overhead exists if ΔFTL's write routine takes more time than the baseline. Thus, there is *no* overhead if the following expression is true:

$$MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode}) * n * P_c + \\ ((1 - P_c) * n + P_c * n * R_c) * T_{write} < n * T_{write} \quad (1)$$

Expression 1 can be simplified to:

$$1 - Rc > \frac{MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode})}{T_{write}} \quad (2)$$

Substituting the numerical values in Table 1 and Table 3, the right side of Expression 2 is 0.45, 0.22, and 0.20, for micro-processor running at 304, 619, and 934MHz, respectively. Therefore, the viable range of R_c should be smaller than 0.55, 0.78, and 0.80. Clearly, high performance micro-processor would impose a less restricted constraint on R_c. If R_c is out of the viable range due to weak content locality in the workload, in order to eliminate the write overhead, ΔFTL must switch to the baseline mode where the delta-encoding procedure is bypassed.

3.4 Flash Allocation

ΔFTL's flash allocation scheme is an enhancement to the regular page mapping FTL scheme with a number of flash blocks dedicated to store the compressed deltas. These blocks are referred to as *Delta Log Area* (DLA). Similar to page mapping area (PMA), we allocate a clean block for DLA so long as the previous active block is full [2]. The garbage collection policy will be discussed in Section 3.6. DLA cooperates with PMA to render the latest version of one data page if it is stored as *delta-XOR-old* form. Obviously, read requests for such data page would suffer from the overhead of fetching two flash pages. To alleviate this problem, we keep the track of the read access popularity of each delta. If one delta is found read-popular, it is merged with the corresponding old version and the result (data in its original form) is stored in PMA. Furthermore, as discussed in Section 3.2, write-cold data should not be delta-encoded in order to save the over-provisioning space. Considering the temporal locality of a page may last for only a period in the workload, if a page previously considered write-hot is no longer demonstrating its temporal locality, this page should be transformed to its original form from its delta-XOR-old form. ΔFTL periodically scans the write-cold pages and merges them to PMA from DLA if needed.

3.5 Mapping Table

The flash management scheme discussed above requires ΔFTL to associate each valid delta in DLA with its old version in PMA. ΔFTL adopts two mapping tables for this purpose: *Page Mapping Table* (PMT) and *Delta Mapping Table* (DMT). Page mapping table is the primary table indexed by logical page address (LPA) of 32bits. For each LPA, PMT maps it to a physical page address (PPA) in page mapping area, either the corresponding data page is stored as its original form or in delta-XOR-old form. For the later case, the PPA points to the old version. PMT differentiates this two cases by prefixing a flag bit to the 31bits PPA (which can address 8TB storage space assuming a 4KB page size). As demonstrated in Figure 4: if the flag bit is "1", which means this page is stored in delta-XOR-old form, we use the PPA (of the old version) to consult the delta mapping table and find out on which physical page the corresponding delta resides. Otherwise, the PPA in this page mapping table entry points to the original form of the page. DMT does not maintain the offset information of each delta in the flash page; we locate the exact position with the metadata prefixed in the page (Figure 2).

3.5.1 Store Mapping Tables On the Flash

ΔFTL stores both mapping tables on the flash and keeps an *journal* of update records for each table. The updates are first buffered in the in-drive RAM and when they grow up to a full page, these records are flushed to the journal on the flash. In case of power failure, a built-in capacitor or battery in the SSD (e.g., a SuperCap [43]) may provide the power to flush the un-synchronized records to the flash. The journals are merged with the tables periodically.

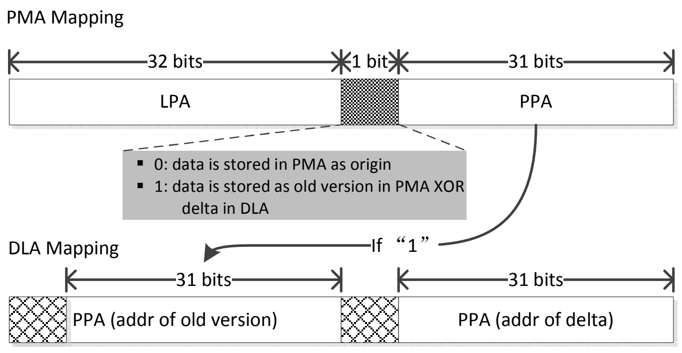
PMA Mapping
32 bits
1 bit
31 bits
LPA
PPA
0: data is stored in PMA as origin
1: data is stored as old version in PMA XOR delta in DLA
DLA Mapping
If "1"
31 bits
31 bits
PPA (addr of old version)
PPA (addr of delta)

Figure 4. ΔFTL Mapping Entry

3.5.2 Cache Mapping Table In the RAM

ΔFTL adopts the same idea of caching popular table entries in the RAM as DFTL [12], as shown in Figure 5(a). The cache is managed using *segment LRU* scheme (SLRU) [19]. Different from two separate tables on the flash, the mapping entries for data either in the original form or delta-XOR-old form are included in one SLRU list. For look-up efficiency, we have all entries indexed by the LPA. Particularly, entries for data in delta-XOR-old form associate the LPA with PPA of old version and PPA of delta, as demonstrated in Figure 5(b). When we have an address look-up miss in the mapping table cache and the target page is in delta-XOR-old form, both on-flash tables are consulted and we merge the information together to an entry as shown in Figure 5(b). As discussed in Section 3.4, the capability of differentiating write-hot and read-hot data is critical to ΔFTL. We have to avoid delta-encoding the write-cold or read-hot data and merge the delta and old version of one page if it is found read-hot or found no longer write-hot. To keep the track of read/write access frequency, we associate each mapping entry in the cache with an *access count*. If the mapping entry of a page is found having a read-access (or write-access) count larger or equal to a predefined threshold, we consider this page read-hot (or write-hot) and vice versa. In our prototyping implementation (discussed in Section 5), we set this threshold as 2 and it captures the temporal locality for both read and writes successfully in our experiments. This information is forwarded to the dispatching policy module to guide the destination of a write request. In addition, merge operations take place if needed.

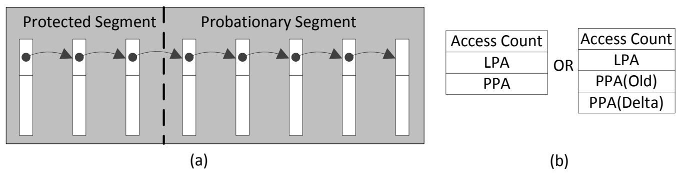

Figure 5. ΔFTL Buffered Mapping Entry

3.6 Garbage Collection

Overwrite operations causes invalidation of old data, which the garbage collection engine is required to discard when clean flash blocks are short. GC engine copies the valid data on the victim block to a clean one and erase the victim thereafter. ΔFTL selects victim blocks based on a simple "greedy" policy, i.e., blocks having the most number of invalid data result in the least number of valid data copy operations and the most clean space reclaimed [21]. ΔFTL's GC victim selection policy does not differentiate blocks from page mapping area or delta log area. In delta log area, the deltas becomes *invalid* in the following scenarios:

- If there is a new write considered not compressible (the latest version will be dispatched to PMA), according to the dispatching policy, the corresponding delta of this request and the old version in PMA become invalid.
- If the new write is compressible and thus a new delta for the same LPA is to be logged in DLA, the old delta becomes invalid.
- If this delta is merged with the old version in PMA, either due to read-hot or write-cold, it is invalidated.
- If there is a TRIM[1] command indicating that a page is no longer in use, the corresponding delta and the old version in PMA are invalidated.

For any case, ΔFTL maintains the information about the invalidation of the deltas for GC engine to select the victims. In order to facilitate the merging operations, when a block is selected as GC victim, the GC engine will consult the mapping table for information about the access frequency of the valid pages in the block. The GC engine will conduct necessary merging operations while it is moving the valid pages to the new position. For example, for a victim block in PMA, GC engine finds out a valid page is associated with a delta which is read-hot, then this page will be merged with the delta and mark the delta as invalidated.

4. Discussion: SSD Lifetime Extension of ΔFTL

Analytical discussion about ΔFTL's performance on SSD lifetime extension is given in this section. In this paper, we use the number of program and erase operations executed to service the write requests as the metric to evaluate the lifetime of SSDs. This is a common practice in most existing related work targeting SSD lifetime improvement [7, 12, 39, 49]. This is because the estimation of SSDs' lifetime is very challenging due to many complicated factors that would affect the actual number of write requests an SSD could handle before failure, including implementation details the device manufacturers would not unveil. On the other hand, comparing the P/E counts resulted from our approach to the baseline is relatively a more practical metric for the purpose of performance evaluation. Write amplification is a well-known problem for SSDs: due to the out-of-place-update feature of NAND flash, the SSDs have to take multiple flash write operations (and even erase operations) in order to fulfill one write request. There are a few factors that would affect the write amplification, e.g., the write buffer, garbage collection, wear leveling, etc [15]. We focus on garbage collection for our discussion, providing that the other factors are the same for ΔFTL and the regular page mapping FTLs. We breakdown the total number of P/E operations into two parts: the **foreground** writes issued from the write buffer (for the baseline) or ΔFTL's dispatcher and delta-encoding engine; the **background** page writes and block erase operations involved in GC processes. Symbols introduced in this section are listed in Table 2.

[1] The TRIM [44] command informs a SSD which pages of data are no longer considered in use and can be marked as invalid. Such pages are reclaimed so as to reduce the no-in-place-write overhead caused by subsequent overwrites.

4.1 Foreground Page Writes

Assuming for one workload, there is a total number of N page writes issued from the write buffer. The baseline has N foreground page writes while ΔFTL has $(1 - P_c) * N + P_c * N * R_c$ (as discussed in Section 3.3.2). ΔFTL would resemble the baseline if P_c (percentage of compressible writes) approaches 0 or R_c (average compression ratio of compressible writes) approaches 1, which means the temporal locality or content locality is weak in the workload.

4.2 GC Caused P/E Operations

The P/E operations caused by GC processes is essentially determined by the frequency of GC and the average overhead of each GC, which can be expressed as:

$$PE_{gc} \propto F_{gc} * OH_{gc} \tag{3}$$

GC process is triggered when clean flash blocks are short in the drive. Thus, the GC frequency is proportional to *the consumption speed of clean space* and inversely proportional to *the average number of clean space reclaimed of each GC* (GC gain):

$$F_{gc} \propto \frac{S_{cons}}{G_{gc}} \tag{4}$$

Consumption Speed is actually determined by the number of foreground page writes (N for the baseline). *GC Gain* is determined by the average number of invalid pages on each GC victim block.

4.2.1 GC P/E of The Baseline

First, let's consider the baseline. Assuming for the given workload, all write requests are overwrites to existing data in the drive, then N page writes invalidate a total number of N

existing pages. If these N invalid pages spread over T data blocks, the average number of invalid pages (thus GC Gain) on GC victim blocks is N/T. Substituting into Expression 4, we have the following expression for the baseline:

$$F_{gc} \propto \frac{N}{N/T} = T \tag{5}$$

For each GC, we have to copy the valid pages (assuming there are B_s pages/block, we have $B_s - N/T$ valid pages on each victim block on average) and erase the victim block. Substituting into Expression 3, we have:

$$PE_{gc} \propto T * (Erase + Program * (B_s - N/T)) \tag{6}$$

4.2.2 GC P/E of ΔFTL

Now let's consider ΔFTL's performance. Among N page writes issued from the write buffer, $(1 - P_c) * N$ pages are committed in PMA causing the same number of flash pages in PMA to be invalidated. Assuming there are t blocks containing invalid pages caused by those writes in PMA, we apparently have $t \leq T$. The average number of invalid pages in PMA is then $(1 - P_c) * N/t$. On the other hand, $P_c * N * R_c$ pages containing compressed deltas are committed to DLA. Recall that there are three scenarios where the deltas in DLA get invalidated (Section 3.6). Omitting the last scenario which is rare compared to the first two, the number of deltas invalidated is determined by the *overwrite rate* (P_{ow}) of deltas committed to DLA: while we assume in the workload all writes are overwrites to existing data in the drive, this *overwrite rate* here defines the percentage of deltas that are overwritten by the subsequent writes in the workload. For example, no matter the subsequent writes are incompressible and committed to PMA or otherwise, the corresponding delta gets invalidated. The average invalid space (in the term of pages) of victim block in DLA is thus $P_{ow} * B_s$. Substituting these numbers to Expression 4: If the average GC gain in PMA outnumbers that in DLA, we have:

$$F_{gc} \propto \frac{(1 - P_c + P_c R_c)N}{(1 - P_c)N/t} = t(1 + \frac{P_c R_c}{1 - P_c}) \tag{7}$$

Otherwise, we have:

$$F_{gc} \propto \frac{(1 - P_c + P_c R_c)N}{P_{ow} B_s} \tag{8}$$

Substituting Expression 7 and 8 to Expression 3, we have for GC introduced P/E:

$$\begin{gathered} PE_{gc} \propto t(1 + \tfrac{P_c R_c}{1 - P_c}) * \\ (Erase + Program * (B_s - (1 - P_c)N/t)) \end{gathered} \tag{9}$$

or:

$$\begin{gathered} PE_{gc} \propto \tfrac{(1 - P_c + P_c R_c)N}{P_{ow} B_s} * \\ (T_{erase} + T_{write} * B_s(1 - P_{ow})) \end{gathered} \tag{10}$$

4.3 Summary

From above discussions, we observe that ΔFTL favors the disk I/O workloads that demonstrate: (i) High content locality that results in small R_c; (ii) High temporal locality for writes that results in large P_c and P_{ow}. Such workload characteristics are widely present in various OLTP applications such as TPC-C, TPC-W, etc [20, 40, 49, 50].

5. Performance Evaluation

We have implemented and evaluated our design of ΔFTL based on a series of comprehensive trace-driven simulation experiments. In this section, we present the experimental results comparing ΔFTL with the page mapping FTL as the baseline. In Section 4, the total number of P/E operations are broken down to foreground writes and GC introduced P/E's for intuitive analytical discussions. Essentially, the number of foreground writes and the efficiency of GC are reflected by the number of GC operations. Thus, in this section we use the number of GC operations as the major metric to evaluate ΔFTL's performance on extending SSD's lifetime. In addition, we evaluate the overheads ΔFTL may potentially introduce, including read and write performance. Particularly, read/write performance is measured in terms of response time.

5.1 Simulation Tool and SSD Configurations

ΔFTL is a device-level software in the SSD controller. We have implemented it (as well as the baseline page mapping FTL) in an SSD simulator based on the Microsoft Research SSD extension [2] for DiskSim 4.0. The simulated SSD is configured as follows: there are 16 flash chips, each of which owns a dedicated channel to the flash controller. Each chip has four planes that are organized in a RAID-0 fashion; the size of one plane is 1GB assuming the flash is used as 2-bit MLC (page size is 4KB). To maximize the concurrency, each individual plane has its own allocation pool [2]. The garbage collection processes are executed in the background so as to minimizing the interference upon the foreground requests. In addition, the percentage of flash space over-provisioning is set as 30%, which doubles the value suggested in [2]. Considering the limited working-set size of the workloads used in this paper, 30% over-provisioning is believed to be sufficient to avoid garbage collection processes to be executed too frequently. The garbage collection threshold is set as 10%, which means if the clean space goes below 10% of the exported space, the garbage collection processes are triggered. Due to negligible impact that the write buffer size has on ΔFTL's performance compared to the baseline, we only report the results with buffer size of 64MB. The SSD is connected to the host via a PCI-E bus of 2.0 GB/s. In addition, the physical operating parameters of the flash memory are summarized in Table 3.

5.2 Workloads

We choose 6 popular disk I/O traces for the simulation experiments. *Financial 1* and *Financial 2* (F1, F2) [40] were obtained from OLTP applications running at two large financial institutions; the *Display Ads Platform and payload servers* (DAP-PS) and *MSN storage metadata* (MSN-CFS) traces were from the Production Windows Servers and described in [20] (MSN-CFS trace contains I/O requests on multiple disks and we only use one of them); the *Cello99* [14] trace pool is collected from the "Cello" server that runs HP-UX 10.20. Because the entire *Cello99* is huge, we randomly use one day traces (07/17/99) of two disks (Disk 3 and Disk 8). Table 4 summarizes the traces we use in our simulation.

Table 4. Disk Traces Information

	Reads(10^6)	Read %	Writes	Write %	Duration(h)
F1	1.23	23.2	4.07	76.8	12
F2	3.04	82.3	0.65	17.7	12
C3	0.75	35.3	1.37	64.7	24
C8	0.56	27.4	1.48	72.6	24
DAP	0.61	56.2	0.47	43.8	24
MSN	0.82	75.0	0.27	25.0	6

5.3 Emulating the Content Locality

As pointed out in [7, 8, 32, 50], the content locality of a workload is application specific and different applications may result in distinctive extent of content locality. In this paper, instead of focusing on only the workloads possessing intensive content locality, we aim at exploring the performance of ΔFTL under diverse situations. As discussed in Section 4, the content locality as well as temporal locality are leading factors that have significant impact on ΔFTL's performance. In our trace-driven simulation, we explore various temporal locality characteristics via 6 disk I/O traces; on the other hand, we emulate the content locality by assigning randomized compression ratio values to the write requests in the traces. The compression ratio values follows Gaussian distribution, of which the average equals R_c. Referring to the values of R_c reported in [50] (0.05 to 0.25) and in [32] (0.17 to 0.6), we evaluate three levels of content locality in our experiments, having $R_c = 0.50, 0.35$, and 0.20 to represent low, medium, and high content locality, respectively. In the rest of this section, we present the experimental results under 6 traces and three levels of content locality, comparing ΔFTL with the baseline.

5.4 Experimental Results

To verify the performance of ΔFTL, we measure the number of garbage collection operations and foreground writes, the write latency, and overhead on read latency.

5.4.1 Number of Garbage Collection Operations and Foreground Writes

First, we evaluate the number of garbage collection operations as the metric for ΔFTL's performance on extending SSD lifetime. Due to the large range of the numerical values of the experimental results, we normalize them to the corresponding results of the baseline as shown in Figure 6. Clearly, ΔFTL significantly reduces the GC count compared to the baseline: ΔFTL results in only 58%, 46%, and 33% of the baseline GC count on average, for $R_c = 0.50, 0.35, 0.20$ respectively. ΔFTL's maximum performance gain (22% of baseline) is achieved with C3 trace when $R_c = 0.20$; the minimum (82%) is with F1, $R_c = 0.50$. We may observe from the results that the performance gain is proportional to the content locality, which is represented by the average compression ratio R_c; in addition, ΔFTL performs relatively poorer with two traces F1 and F2, compared to the rests. In order to interpret our findings, we examine two factors that determine the GC count: the consumption speed of clean space (S_{cons}, Expression 4) and the speed of clean space reclaiming, i.e., the average GC gain (G_{gc}). **Consumption Speed:** As

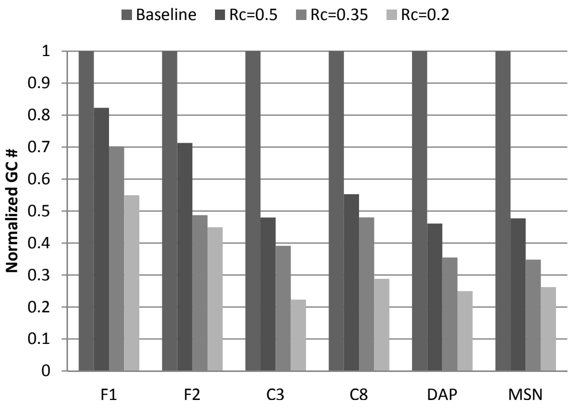

Figure 6. Normalized GC #: comparing baseline and ΔFTL; smaller # implies longer SSD lifetime.

discussed in Section 4, the consumption speed is determined by the number of foreground flash page writes. We plot the normalized number of foreground writes in Figure 7. As seen in the figure, the results are proportional to R_c as well; F1 and F2 produce more foreground writes than the others, which result in larger GC counts as shown in Figure 6. If there are N writes in the baseline, ΔFTL would have $(1 - P_c + P_c * R_c) * N$. The foreground write counts are reversely proportional to R_c (self-explained in Figure 7) as well as P_c. So, what does P_c look like? Recall in Section 3.2 that P_c is determined by the dispatching rules, which favor write-hot and read-cold data. The access frequency characteristics, i.e., the temporal locality, is workload-specific, which means the P_c values should be different among traces but not affected by R_c. This point is justified clearly in Figure 8, which plots the ratio of DLA

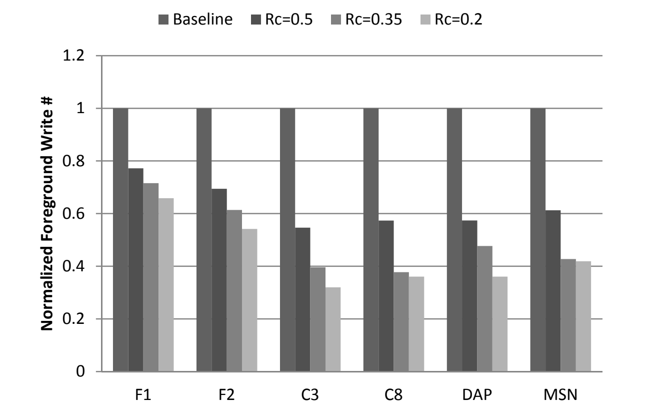

Figure 7. Normalized foreground write #: comparing baseline and ΔFTL; smaller # implies: a) larger P_c and b) lower consumption speed of clean flash space.

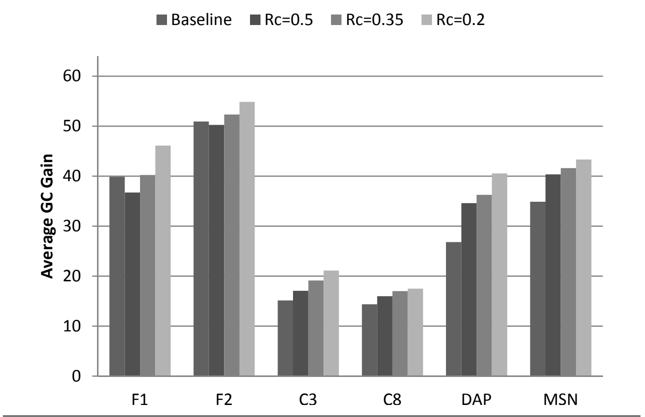

Figure 9. Average GC gain (number of invalid pages reclaimed): comparing baseline and ΔFTL; smaller # implies lower GC efficiency on reclaiming flash space.

writes (P_c) out of the total foreground writes. We may also verify that the foreground write counts (Figure 7) are reversely proportional to P_c: F1 and F2 have the least P_c values among all traces and they produce the most number of foreground writes; this trend can be also observed with other traces. **Garbage collection gain** is another factor that

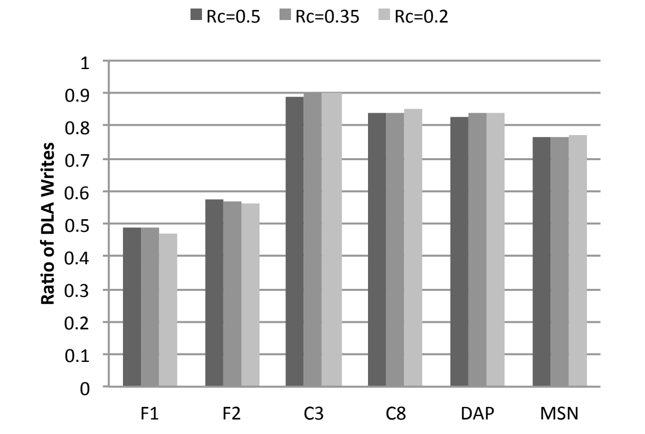

Figure 8. Ratio of DLA writes (P_c).

determines GC count. Figure 9 plots the average GC gain in terms of the number of invalid pages reclaimed. GC gain ranges from 14 (C8, baseline) to 54 (F2, $R_c = 0.20$). F1 and F2 outperform the other traces on the average GC gain. However, comparing to the baseline performance, ΔFTL actually does not improve much with F1 and F2: we normalize each trace's results with its individual baseline in Figure 10. ΔFTL even degrades average GC gain with F1 and F2 when $R_c = 0.50$. This also complies with the GC count results shown in Figure 6, where ΔFTL achieves poorer performance gain with F1 and F2 compared to the others. The reason why ΔFTL does not improve GC gain significantly over the baseline with F1 and F2 is: compared to the other traces, F1 and F2 result in larger invalid page counts in blocks of PMA, which makes ΔFTL's GC engine

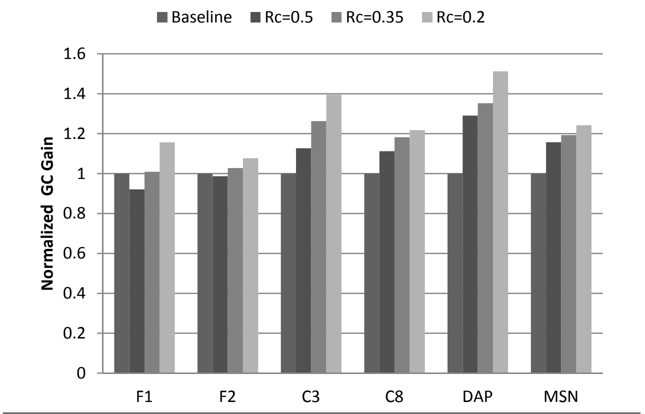

Figure 10. Normalized average GC gain (number of invalid pages reclaimed): comparing baseline and ΔFTL.

to choose more PMA blocks as GC victims than DLA blocks. Thus, the average GC gain performance of ΔFTL resembles the baseline. To the contrary, ΔFTL benefits from the relative higher temporal locality of write requests in the DLA than in the PMA, under the other 4 traces. This is the reason why ΔFTL outperforms the baseline with these traces. In order to verify this point, we collect the number of GC executed in DLA and plot the ratio over the total in Figure 11: the majority of the total GC operations lies in PMA for F1 and F2 and in DLA for the rest.

5.4.2 Write Performance

In ΔFTL, the delta-encoding procedure in servicing a write request may cause overhead on write latency if R_c is out of the viable range (Section 3.3.2). R_c values adopted in our simulation experiments ensures there is no write overhead. ΔFTL significantly reduces the foreground write counts, and the write latency performance also benefits from this. As shown in Figure 12, ΔFTL reduces the average write latency by 36%, 47%, and 51% when $R_c = 0.50, 0.35, 0.20$, respectively.

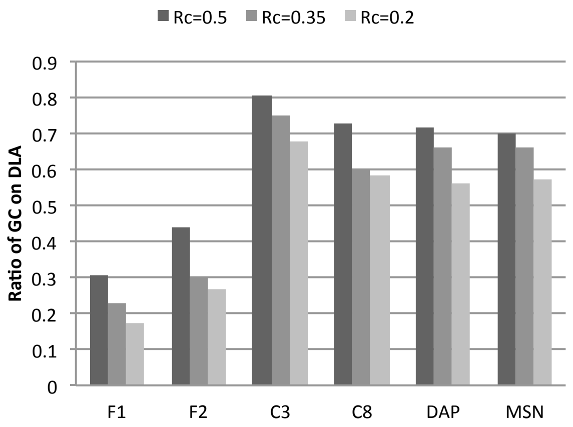

Figure 11. Ratio of GC executed in DLA.

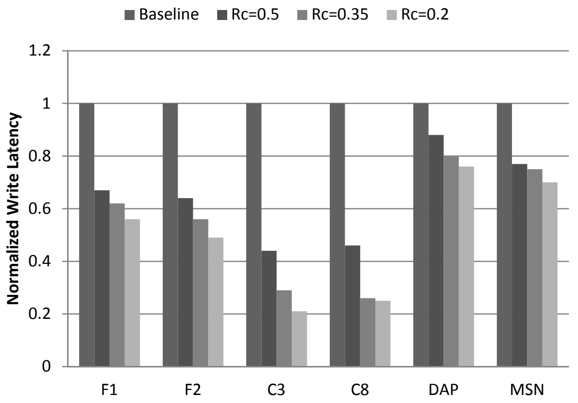

Figure 12. Normalized write latency performance: comparing baseline and ΔFTL.

5.4.3 Garbage Collection Overhead

The GC operation involves copying the valid data from the victim block to a clean block and erasing the victim block. The GC overhead, i.e., the time for a GC operation, may potentially hinder the foreground requests to be serviced. We evaluate the average GC overhead of ΔFTL and compare the results to the baseline in Figure 13. We observe that ΔFTL does not significantly increase the GC overhead under most cases.

5.4.4 Overhead on Read Performance

ΔFTL reduces the write latency significantly and therefore alleviates the chip contention between the read and write requests, resulting less queuing delay for the reads. Under intensive workloads, the effective read latency (considering queuing delay on the device side) is reduced in Delta-FTL. However, ΔFTL inevitably introduces overhead on the raw read latency (despite queuing delay) when the target page is delta-encoded. Fulfilling such a read request requires two flash read operations. To overcome this potential overhead, ΔFTL delta-encodes only the write-hot and read-cold data and merges DLA pages to their original form if they are

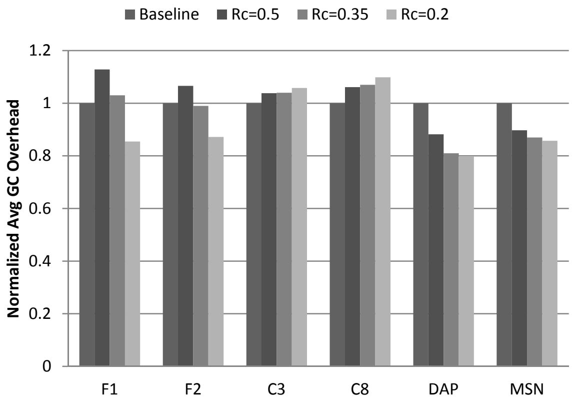

Figure 13. Normalized average GC overhead.

found read-hot. To evaluate the effectiveness of our approach, we collect the raw read latency values reported by the simulator and demonstrate the results in Figure 14. Compared to the baseline (normalized to 1), ΔFTL's impact on the read performance is trivial: the read latency is increased by 5.3%, 5.4%, and 5.6% on average[2] when $R_c = 0.50, 0.35, 0.20$, respectively. The maximum (F2, $R_c = 0.50$) is 10.7%. To summarize, our experimental

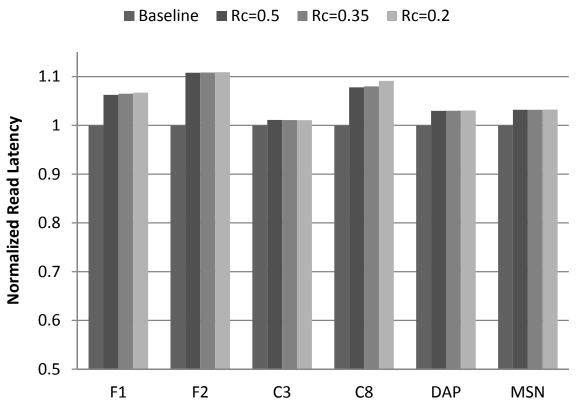

Figure 14. Normalized read latency performance: comparing baseline and ΔFTL.

results verify that ΔFTL significantly reduces the GC count and thus extends SSDs' lifetime at a cost of trivial overhead on read performance.

6. Conclusions and Future Work

The limited lifetime impedes NAND flash-based SSDs from wide deployment in reliability-sensitive environments. In this paper, we have proposed a solution, ΔFTL, to alleviate this problem. ΔFTL extends SSDs' lifetime by reducing the number of program/erase operations for servicing the disk I/O requests. By leveraging the content locality existing between the new data and its old version,

[2] x% read latency overhead implies that x% of the requested pages are delta-encoded, which would double the raw latency compared to non-delta-encoded pages.

ΔFTL stores the new data in the flash in the form of compressed delta. We have presented the design of ΔFTL in detail including the data structures, algorithms, and overhead control approaches in this paper. ΔFTL is prototyped and evaluated via simulation. Our trace-driven experiments demonstrate that ΔFTL significantly extends SSD's lifetime by reducing the number of garbage collection operations at a cost of trivial overhead on read latency performance. Specifically, ΔFTL results in 33% to 58% of the baseline garbage collection operations, while the read latency is only increased by approximately 5%.

Our future work will explore the integration of deduplication to ΔFTL to further improve the lifetime of SSDs. The major technical challenge is to alleviate the pressure of computation and space complexity introduced by the need of managing the mapping of logical addresses to the contents.

Acknowledgments

We thank our shepherd Steven Gribble and anonymous reviewers for their feedback. This research is sponsored in part by the U.S. National Science Foundation (NSF) under grants CCF-1102605 and CCF-1102624. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agency.

References

[1] ARM Cortex R4. `www.arm.com/files/pdf/Cortex-R4-white-paper.pdf`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX ATC*, Boston, Massachusetts, USA, 2008.

[3] D. Andersen and S. Swanson. Rethinking flash in the data center. *IEEE Micro*, 30(4):52–54, 2010.

[4] ARM®. Arm7. `http://www.arm.com/products/processors/classic/arm7/index.php`.

[5] L. A. Barroso. Warehouse-scale Computing. In *Keynote in the SIGMOD10 conference*, 2010.

[6] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *DAC*, San Diego, CA, USA, June 2007.

[7] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of FAST'2011*.

[8] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX ATC*, pages 1–23, 2003.

[9] Google. Snappy. `http://code.google.com/p/snappy/`.

[10] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33. ACM, 2009.

[11] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of FAST'2011*.

[12] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09*, 2009.

[13] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[14] Hewlett-Packard Laboratories. cello99 traces. `http://tesla.hpl.hp.com/opensource/`.

[15] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR09*, page 10. ACM, 2009.

[16] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[17] J. U. Kang, H. Jo, J. S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *International Conference on Embedded Software*, 2006.

[18] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Transactions on Computers*, 58(6):744–758, 2009.

[19] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3): 38–46, March 1994.

[20] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *IISWC*, 2008.

[21] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference*, pages 13–13. USENIX Association, 1995.

[22] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage Abstract. In *Proceedings of FAST*, 2008.

[23] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for Compact Flash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[24] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. FAST: An FTL Scheme with Fully Associative Sector Translations. In *UKC*, August 2005.

[25] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS*, 42(6), 2008.

[26] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *USENIX ATC*, June 2009.

[27] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim. uFTL: a memory-efficient flash translation layer supporting multiple mapping granularities. In *EMSOFT*, 2008.

[28] M. Lehmann. Lzf. `http://oldhome.schmorp.de/marc/liblzf.html`.

[29] S. LLC. Simplescalar/arm. `http://www.simplescalar.com/v4test.html`.

[30] J. MacDonald. xdelta. `http://xdelta.org`.

[31] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, 1994.

[32] C. Morrey III and D. Grunwald. Peabody: The time travelling disk. In *Proceedings of MSST 2003*, pages 241–253. IEEE.

[33] M. Oberhumer. Lzo. `http://www.oberhumer.com/opensource/lzo`.

[34] ONFI, 2010. `http://onfi.org/`.

[35] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of CASES'2006*, pages 234–241, 2006.

[36] Samsung, 2010. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products-OneNAND.html`.

[37] J. Seward. The bzip2 and libbzip2 official home page. 2002. `http://sources.redhat.com/bzip2`.

[38] SiliconSystems. Increasing flash solid state disk reliability. *Technical report*, 2005.

[39] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proceedings of FAST*. USENIX, Feb 2010.

[40] Storage Performance Council. SPC trace file format specification. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[41] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA-16*, pages 141–153. IEEE, Jan 2010.

[42] Toshiba. `http://www.toshiba.com/taec/news/press-releases/2006/memy-06-337.jsp`, 2010.

[43] wikipedia. Battery or super cap, 2010. `http://en.wikipedia.org/wiki/Solid-state-drive#Battery_or_SuperCap`.

[44] wikipedia. TRIM, 2012. `http://en.wikipedia.org/wiki/TRIM`.

[45] C.-H. Wu and T.-W. Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of ICCAD '06*, 2006.

[46] G. Wu and X. He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of FAST'2012*.

[47] G. Wu, X. He, N. Xie, and T. Zhang. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. In *MASCOTS*, pages 57–66, 2010.

[48] G. Wu, X. He, and B. Eckart. An Adaptive Write Buffer Management Scheme for Flash-Based SSDs. *ACM Transactions on Storage*, 8(1), 2012.

[49] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of HPCA 2011*, pages 278–289. IEEE.

[50] Q. Yang, W. Xiao, and J. Ren. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. *ACM SIGARCH Computer Architecture News*, 34(2):289–301, 2006.