공학박사학위논문

낸드 플래시 저장장치를 위한 시스템 수준 정보 기반 수명 개선 기법

Lifetime Improvement Techniques for NAND Flash-Based Storage Devices Based on System-Level Information

서울대학교 대학원
전기·컴퓨터 공학부
김태진

낸드 플래시 저장장치를 위한
시스템 수준 정보 기반 수명 개선 기법

Lifetime Improvement Techniques for

NAND Flash-Based Storage Devices

Based on System-Level Information

지도교수 김 지 홍

이 논문을 공학박사 학위논문으로 제출함

서울대학교 대학원
전기·컴퓨터 공학부
김태진

김태진의 공학박사 학위논문을 인준함

위 원 장 _____ (인)

부위원장 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

Abstract

Replacing HDDs with NAND flash-based storage devices (SSDs) has been one of the major challenges in modern computing systems especially in regards to better performance and higher mobility. Although the continuous semiconductor process scaling and multi-leveling techniques lower the price of SSDs to the comparable level of HDDs, the decreasing lifetime of NAND flash memory, as a side effect of recent advanced device technologies, is emerging as one of the major barriers to the wide adoption of SSDs in high-performance computing systems.

In this dissertation, system-level lifetime improvement techniques for recent high-density NAND flash memory are proposed. Unlike existing techniques, the proposed techniques resolve the problems of decreasing lifetime by exploiting the write request characteristics, such as context of I/O or duplicate data contents.

We first propose a system-level approach to reduce WAF that exploits the I/O context of an application to increase the data lifetime prediction for the multi-streamed SSDs. Since the key motivation behind the proposed technqiue was that data lifetimes should be estimated at a higher abstraction level than LBAs, we employ a write program context as a stream management unit. Thus, it can effectively separate data with short lifetimes from data with long lifetimes to improve the efficiency of garbage collection.

Second, we present a write traffic reduction approach which reduces the amount of write traffic sent to a storage by eliminating redundant data,

therby improving the lifetime of storage devices. In particular, we improve the likelihood of eliminating redundant data by introducing sub-page chunk based on the understanding of duplicate contents such as partial updates or zero padding of file system. It also resolves technical difficulties caused by its finer granularity, i.e., increased memory requirement and read response time.

In order to evaluate the effectiveness of the proposed techniques, we performed a series of evaluations using both a trace-driven simulator and emulator with I/O traces which were collected from various real-world systems. To understand the feasibility of the proposed techniques, we also implemented them in Linux kernel on top of our in-house flash storage prototype and then evaluated their effects on the lifetime while running real-world applications. Our experimental results show that system-level optimization techniques are more effective over existing optimization techniques.

Keywords: NAND Flash-Based Storage Devices, Storage Lifetime, Embedded Software, Operating System

Student Number: 2012-30201

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

NAND flash-based solid-state drives (SSDs) are widely used in personal computing systems as well as mobile embedded systems. However, in enterprise environments, SSDs are employed in only limited applications because SSDs are not yet cost competitive with HDDs. Fortunately, the prices for SSDs have fallen to the comparable level of HDDs by continuous semiconductor process scaling (e.g., 10 nm-node process) combined with multi-leveling technologies (e.g., MLC and TLC). However, the limited endurance of NAND flash memory, which have declined further as a side effect of the recent advanced device technologies, is emerging as another major barrier to the wide adoption of SSDs. (NAND endurance is the ability of a memory cell to endure program/erase (P/E) cycling, and is quantified as the maximum number $MAX_{P/E}$ of P/E cycles that the cell can tolerate while maintaining its reliability requirements.) For example, although the NAND capacity per die doubles every two years, the actual lifetime of SSDs does not increase as much as projected in the past seven years because the $MAX_{P/E}$ has declined by 70% during that period [1]. Since the reduction in the $MAX_{P/E}$ seriously limits the overall lifetime of flash-based SSDs, the issues concerning the lifetime of SSDs should be properly resolved for

SSDs to be commonly used in enterprise environments.

Since the Lifetime L_C of an SSD with the total capacity C is proportional to the maximum number $MAX_{P/E}$ of P/E cycles, and is inversely proportional to the total written data W_{day} per day, L_C (in days) can be expressed as follows [2](assuming a perfect wear leveling):

$$L_C = \frac{MAX_{P/E} \times C}{W_{day} \times WAF}$$

, where WAF is a write amplification factor which represents the efficiency of an FTL algorithm. Since $MAX_{P/E}$ and C is determined when the device is manufactured, we should reduce the W_{day} and WAF to improve the lifetime of SSDs. Many existing lifetime-enhancing techniques have been focused on reducing WAF by increasing the efficiency of an FTL algorithm. For example, by avoiding unnecessary data copies during garbage collection using the multi-stream feature, WAF can be reduced. In order to reduce W_{day}, various system-level techniques were proposed. For example, data deduplication, data compression, and write traffic throttling are such techniques.

Most existing studies, however, are based on the the single I/O layer such as block I/O, device driver, and SSD firmware so their effectiveness is limited. In order for NAND flash-based storage devices to be broadly adopted in various computing environments, therefore, new approaches that properly address the lifetime problem of recent high-density NAND flash memory are highly required.

1.2 Dissertation Goals

In this dissertation, we propose system-level approaches that improve the lifetime of NAND flash-based storage devices, which overcomes the limitations of the existing techniques. More specifically, our primary goal is to understand high-level information of applications or file systems, such as the I/O context of dominant I/O activities and data contents of the file system, and then develop the lifetime improvement approaches that efficiently exploit such high-level information at various system levels ranging from a system call layer to a flash controller.

First, we propose a system-level approach to reduce WAF that exploits the I/O context of an application to increase the data lifetime prediction for the multi-streamed SSDs. By extracting program contexts during runtime, data-to-stream mapping is done automatically. Thus, it can effectively separate data with short lifetimes from data with long lifetimes to improve the efficiency of garbage collection. Moreover, when data mapped to the same stream show large differences in their lifetimes, long-lived data of the current stream are moved to its substream during garbage collection.

Second, present a write traffic reduction approach which reduces the amount of write traffic sent to a storage by eliminating redundant data, therby improving the lifetime of storage devices. In particular, we increase the overall deduplication ratio[1] by introducing sub-page chunk based on the understanding of file system behavior. It resolves technical difficulties caused by its finer granularity, i.e., increased memory requirement and read

[1] The percentage of identified duplicate writes

response time.

1.3 Contributions

In this dissertation, we present two system-level techniques to improve the lifetime of NAND flash-based storage devices. The contributions of this dissertation can be summarized as follows:

- We present a fully automatic stream management technique, called PCStream, for multi-streamed SSDs based on program contexts (PCs). Since the key motivation behind PCStream was that data lifetimes should be estimated at a higher abstraction level than LBAs, PCStream employed a write program context as a stream management unit. A program context [11, 12], which represents a particular execution phase of a program, is known to be an effective hint in separating data with different lifetimes [8]. PCStream automatically maps an identified program context to a stream. Since program contexts can be computed during runtime, PCStream does not need any manual work.

- We propose a fine-grained deduplication technique for flash-based SSDs, called *FineDedup*. The proposed FineDedup technique is different from other existing deduplication techniques in that it increases the likelihood of finding duplicates by using a finer deduplication unit which is smaller than a single page (e.g., one fourth of a single page). With a smaller deduplication unit, many data segments within a page

can be detected as a duplicate one, so the amount of data written to flash memory can be reduced regardless of a physical page size.

- We implement the proposed techniques in the Linux kernel and our in-house flash storage prototype. Then, we evaluate their effects on lifetime using various real-world applications.

1.4 Dissertation Structure

This dissertation is composed of five chapters. The first chapter is the introduction of theh dissertation, while the last chapter serves as conclusions with a summary and future work. The three intermediate chapters are organized as follows:

Chapter 2 provides the overall architecture of NAND flash-based storage devices. We also describe the existing lifetime improvement techniques for flash-based devices, focusing on multi-streamed SSDs and deduplication techniques which are highly related to our proposed techniques.

In Chapter 3, we present a new data separation technique, called PC-Stream, for multi-streamed SSDs. We explain the relationship of dominant I/O activities with data lifetime patterns.

Chapter 4 introduces a fine-grained deduplication technique, called FineDedup, for NAND flash-based storage devices. We describe the patterns of duplicate data with in a page. Finally, we show how effective the proposed technique is in terms of write traffic reduction.

Chapter 2

Related Work

2.1 Data Separation Techniques for Multi-streamed SSDs

Research exists to detect data temperature. Park, et al. uses multiple bloom filters to identify hot data in the device layer. Stoica, et al. propose new data placement algorithms to improves flash write performance by estimating data update frequencies. Luo, et al. observe high temporal write locality in different workloads and design a write-hotness aware retention management policy to improve flash memory life time. Most research is on a simulation or mathematical modeling basis and those lack of real world system and performance analysis. It is hard to guarantee the benefit of these algorithms in a dynamic I/O intensive datacenter workloads. In adition, as multi-stream SSDs become available, there is a need to identify data temperature and separate them to multiple levels (usually more than three levels - hot, cold and warm) to fully utilize such devices.

In order to achieve high performance on multi-streamed SSDs, data with similar *future* update times [8] should be allocated to the same stream, so that the copy cost of GC can be minimized. However, since it is difficult to know the future update times *a priori* when they are written, stream allocation decisions are often *manually* made by programmers based on

their expertise on the application [4, 5] or the file system [6]. Furthermore, these manual techniques assume that the number of streams in an SSD is not changing, thus requiring manual modifications whenever the number of streams in the SSD changes.

To the best of our knowledge, AutoStream [7] is the only automatic stream management technique without additional manual work. However, since AutoStream predicts data lifetimes using the update frequency of the logical block address (LBA), it does not work well with modern append-only workloads such as RocksDB [9] or Cassandra [10]. Unlike conventional update workloads where data written to the same LBAs often show strong update locality, append-only workloads make it impossible to predict data lifetimes from LBA characteristics (such as access frequency or access patterns). For example, as shown in Fig. 1(b), data written to a fixed LBA range over time in RocksDB show widely varying data lifetimes, thus making it difficult to allocate streams based on LBA characteristics.

2.2 Write Traffic Reduction Techniques

In order to extend the lifetime of flash-based SSDs, data deduplication techniques have been used in recent SSDs because they are effective in reducing the amount of data written to flash memory by preventing duplicate data from being written again [23, 24]. As a result, only non-duplicate data, i.e., unique data, are stored in SSDs effectively decreasing the total amount of data written to SSDs. In most deduplication schemes proposed for SSDs, the unit of data deduplication is same as the flash page size which is usually

4 KB or 8 KB. Using a page as a deduplication unit seems to be reasonable because the unit of a read or write operation of flash memory is also a page. However, this page-based deduplication technique misses many chances of eliminating duplicate data, especially when two pages are *almost* identical. For example, in our experimental analysis of an existing 4 KB page-based deduplication technique, we observed that up to 34% mostly identical data. If the unit of deduplication were smaller than 4 KB, about 23% more data could be identified as duplicate data. Furthermore, it is expected that the effectiveness of the page-based deduplication technique would get even worse in future NAND flash memory as the page size of flash memory is expected to increase to a bigger size such as 16 KB [19].

Because of the "erase-before-write" nature of NAND flash memory, flash storage devices employ a flash translation layer (FTL) that supports address mapping, garbage collection, and wear-leveling algorithms [20]. These firmware algorithms incur a lot of extra write/erase operations, seriously shortening the overall lifetime of a storage device. For this reason, a large number of studies have been focused on reducing such extra operations to improve the storage lifetime. However, considering the decreasing lifetime of recent high-density NAND flash memory such as TLC NAND flash memory [18], more aggressive lifetime management solutions are required.

Data deduplication techniques, which are originally developed for backup systems, are regarded as one of the promising approaches for extending the storage lifetime because of their ability that reduces the amount of write traffic sent to a storage device. In deduplication techniques, a chunk is used

as an unit of identification and elimination of duplicate data. Depending on their chunking strategies, deduplication techniques can be categorized into two types, fixed-size deduplication and variable-size deduplication. Fixed-size deduplication divides an input data stream into fixed-size chunks (e.g., pages) [23, 24]. Then, it decides if each chunk data is duplicate and prevents duplicate chunks from being rewritten to flash memory. Unlike fixed-size deduplication, the chunk size of variable-size deduplication is not fixed. Instead, it decides a cut point between chunks using a content-defined chunking (CDC) algorithm which divides the data stream according to the contents [27, 28].

In general, variable-size deduplication techniques can identify more data as duplicate data than the fixed-size deduplication technique. Since variable-size deduplication adaptively changes the size of chunks by analyzing the contents of input stream, duplicate data are more effectively found regardless of their locations. In spite of its advantages, variable-size deduplication is not commonly used in SSDs because of the following practical limitations.

First, the CDC algorithm often requires relatively high computational power and a large amount of memory space. Thus, variable-size deduplication is not appropriate to be employed at the level of storage devices where computing and memory resources are constrained. Second, the size of remaining unique data after deduplication may vary in variable-size deduplication. When writing those data, a complicated scheme for data size management is required to form sub-page data chunks to fit in a flash page size, preventing an internal fragmentation. For those reasons, most existing

deduplication techniques for SSDs employ fixed-size deduplication, which is relatively simple and does not require a significant amount of hardware resources.

There are several existing studies for fixed-size deduplication for SSDs. F. Chen [23] proposed CAFTL to enhance the endurance of SSDs with a set of acceleration techniques to reduce runtime overhead. A. Gupta [24] also proposed CA-SSD to improve the reliability of SSDs by exploiting the value locality, which implies that certain data items are likely to be accessed preferentially. In these studies, authors focused on the feasibility of deduplication at SSD level and proved its effectiveness rather than improving deduplication itself.

Recently, several deduplication techniques for flash-based storage are proposed. Z. Chen [25] proposed OrderMergeDedup which orders and merges the deduplication metadata with data writes to realize failure-consistent storage with deduplication. W. Li [26] proposed CacheDedup which integrates deduplication with caching architecture to address limited endurance of flash caching by managing data writes and deduplication metadata together, and proposing duplication-aware cache replacement algorithms. These studies focus on systematic approach such as block layer or flash caching. However, this study improves the effect of deduplication in the device-specific domain, so the approach of this study is quite different.

Similar to the existing deduplication techniques, the proposed FineDedup technique is also based on fixed-size deduplication. Using a smaller deduplication unit, however, FineDedup improves the likelihood of eliminating duplicate data. This approach can complement the limitation of ex-

isting fixed-size deduplication techniques, which exhibit a relatively low amount of removed writes in comparison with variable-size deduplication.

Chapter 3

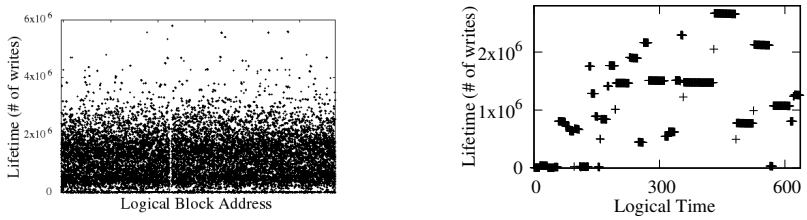
Automatic Stream Allocation Using Program Contexts

3.1 Motivation

3.1.1 Fallacy: LBA-based lifetime prediction

Multi-streamed SSDs provide a special mechanism, called streams, for a host system to prevent data with different lifetimes from being mixed into the same block [3, 4]. When the host system maps two data D_1 and D_2 to different streams S_1 and S_2, a multi-streamed SSD guarantees that D_1 and D_2 are placed in different blocks. Since streams, when properly managed, can be very effective in minimizing the copy cost of garbage collection (GC), they can significantly improve both the performance and lifetime of flash-based SSDs [4, 5, 6, 7].

In order to achieve high performance on multi-streamed SSDs, data with similar *future* update times [8] should be allocated to the same stream, so that the copy cost of GC can be minimized. However, since it is difficult to know the future update times *a priori* when they are written, stream allocation decisions are often *manually* made by programmers based on their expertise on the application [4, 5] or the file system [6]. In this paper, our goal is to develop a *fully automatic* stream management technique.



(a) Lifetime patterns over LBAs (b) Lifetime patterns over time

Figure 1: Lifetime distributions over addresses and times.

Many existing data separation techniques such as [7, 13] estimate the data lifetime based on the update frequency of LBAs. For example, AutoStream [7] assumes that, if some LBAs are frequently rewritten by applications, those LBAs hold hot data. This LBA-based lifetime prediction approach, however, does not work well with recent data-intensive applications where a majority of new data are written in an append-only manner.

In order to illustrate a mismatch between an LBA-based predictor and append-only workloads, we analyzed the write pattern of RocksDB [9], which is a popular key-value store based on the LSM-tree algorithm [14]. Fig. 1(a) shows how LBAs may be related to data lifetimes[1] in RocksDB [9]. As shown in Fig. 1(a), there exists no strong correlation between the LBAs and lifetimes in RocksDB. This scatter plot is in sharp contrast with one for update workloads where a few distinct LBA regions have short lifetimes while others have very long lifetimes.

We also analyzed if the lifetimes of LBAs change under some predictable patterns over time although the overall lifetime distribution shows

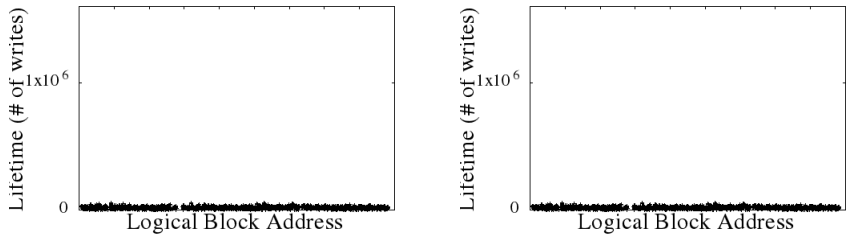
[1]The lifetime of data is defined by the logical time which is the number of writes to the device between when the data is first written and when the data is invalidated by an overwrite or a TRIM command.

large variances. Fig. 1(b) shows a scatter plot of data lifetimes over the logical time for a specific 1-MB chunk with 256 pages. As shown in Fig. 1(b), for the given chuck, data lifetimes vary in a random fashion (although some temporal locality is observed). Our illustration using RocksDB strongly suggests that under append-only workloads, LBAs are not useful in deciding data lifetimes.

3.1.2 Program context as a lifetime predictor

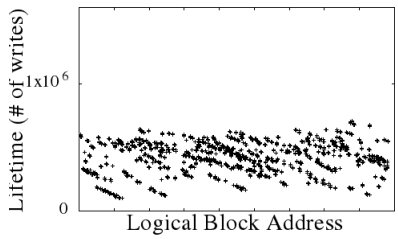
In developing PCStream, our key insight was that in most applications, a few dominant I/O activities exist and each dominant I/O activity represents the application's important I/O context (e.g., for logging or for flushing). Furthermore, most dominant I/O activities tend to have distinct data lifetime patterns. In order to distinguish data by their lifetimes, therefore, it is important to effectively distinguish dominant I/O activities from each other. For example, in update workloads, LBAs alone were effective in separating dominant I/O activities.

In this paper, we argue that a program context is an efficient general-purpose indicator for separating dominant I/O activities regardless of the type of I/O workloads. Since a PC represents an execution path of an application which invokes write-related system functions such as `write()` and `writev()` in the Linux kernel, we represent the PC by summing program counter values of all the functions along the execution path which leads to a write system call. In RocksDB, for example, dominant I/O activities include logging, flushing and compaction. Since they are invoked through different function-call paths, we can easily identify dominant I/O activities

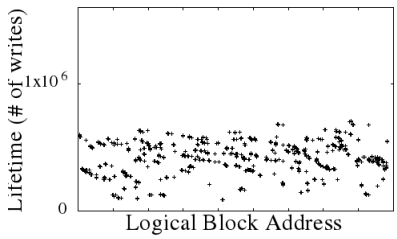


(a) Logging (PC)

(b) Logging (manual)



(c) Flushing (PC)



(d) Flushing (manual)

Figure 2: Data lifetime distributions of different PCs.

of RocksDB using PCs. For example, Fig. 4(a) shows an execution path for flushing in RocksDB. The sum of program counter values of `Run()`, `WriteLevel0Table()`, and `BuildTable()` is used to represent the PC for the flushing activity. Note that using the program context to distinguish data lifetimes is not new. For example, Ha *et al.* proposed a data separation technique based on the program context [8]. However, their work was neither designed for append-only workloads nor for modern multi-streamed SSDs.

In order to validate our hypothesis that PCs can be useful for predicting lifetimes by distinguishing dominant I/O activities, we conducted experiments using RocksDB, comparing the accuracy of identifying dominant I/O activities using two different methods. First, we manually identified

dominant I/O activities by inspecting the source code. Second, we automatically decided dominant I/O activities by extracting PCs for write-related system functions. Fig. 2 illustrates two dominant I/O activities matched between two methods. As shown in Fig. 2(a) and 2(b), the logging activity of RocksDB is correctly identified by two methods. Furthermore, from the logging-activity PC, we can clearly observe that data written from the PC are short-lived. Similarly, from Fig. 2(c) and 2(d), we observe that data written from the flushing-activity PC behave in a different fashion. For example, data from the flushing-activity PC remain valid a lot longer than those from the logging-activity PC.

3.2 Design of PCStream

In this section, we describe in detail the proposed automatic stream management technique, PCStream. We first explain how we automatically extract PCs during runtime and describe how multiple PCs are mapped to streams in an SSD. In order to mitigate the side effect of a few outlier PCs with large lifetime variances, we introduce 'substreams' based on a two-phase stream assignment technique.

Fig. 3 shows an overall organization of PCStream. *The PC extractor module*, which is implemented in the Linux kernel as part of a system call handler, computes a PC signature, which is used as a unique ID for each program context. We use the signature program counter [11] as a PC signature by summing program counter values along the execution path to a write-related system function (e.g., `write()`). With the PC signature, we

can monitor the data lifetime of each write at the program context level. A PC signature value is stored in an inode data structure of a file system (modified for PCStream) and is delivered to *the lifetime analyzer module* which estimates expected lifetimes of data belonging to a given PC in the block device level. In order to efficiently detect the end of data lifetime in append-only workloads, the lifetime analyzer also intercepts TRIM [15] requests. Based on the lifetime information, *the PC-to-stream mapper module* clusters PCs with similar lifetimes and maps them together to the same stream ID. This mapping is required because the number of streams in an SSD is generally less than the number of PCs in host applications.

3.2.1 Automatic PC computation

As mentioned earlier, a PC is represented by a PC signature which is defined as the sum of program counter values along the execution path of a function call that reaches a write-related system function. A function call involves pushing the next program counter, which is used as a return address, to the stack followed by pushing a frame pointer value. In general, by using frame pointer values, we are able to back-track the stack frames of the process and selectively get return addresses for generating a PC signature. For example, Fig. 4(a) shows the abstracted execution path for flushing data in RocksDB and Fig. 4(b) illustrates how a PC signature is obtained by back-tracking the stack. Since a frame pointer value in the stack holds the address of the previous frame pointer, the PC extractor can obtain return addresses and accumulate them to compute a PC signature.

The frame pointer-based approach for computing PC signatures, how-

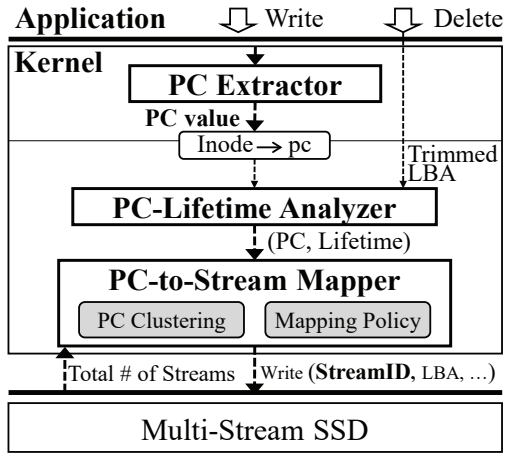
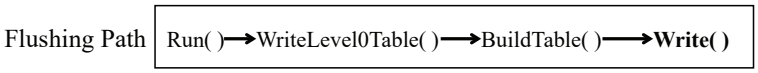


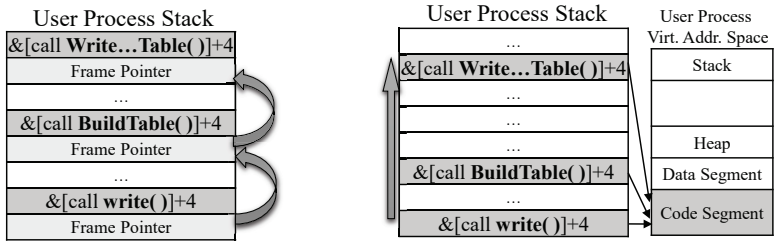
Figure 3: An overall architecture of PCStream.

ever, is not always possible because modern C/C++ compilers often do not use the frame pointer for improving the efficiency of register allocation. One example is a `-fomit-frame-pointer` option of GCC [16]. Although this option allows the frame pointer to be used as a general-purpose register for high performance, it makes very difficult for us to back-track return addresses along the call chains.

In PCStream, we employ a simple but effective workaround for back-tracking the call stack when the frame pointer is not used. When a write system call is made, we scan every word in the stack and check if it belongs to the process's code segment. If the scanned stack word holds a value within the address range of the code segment, we assume that it is a return address. Since scanning the full stack takes too long, we stop the stack scanning procedure when a sufficient number of return address candidates are found. In the current version, we stop when 5 return address candidates are



(a) An abstracted execution path for flushing data.



(b) with the frame pointer. (c) without the frame pointer.

Figure 4: An example execution path and its PC extraction.

found. Although quite ad-hoc, a restricted scan is effective in distinguishing different PCs because two different PCs cannot follow the same execution path to write system functions. (If they do, they are the same PC.) In our evaluation with a 3.4 GHz CPU machine, the performance overhead of the restricted scan was almost negligible, taking only 300-400 $n$sec per write system call.

3.2.2 PC lifetime prediction

The prediction of PC lifetimes is rather complicated. The data lifetime of the append-only workload is defined from when a write request is issued until the TRIM command [15] is issued to the corresponding address. In order to measure the lifetime of data, the lifetime analyzer records the write time and PC value for each write request using its LBA. Upon receiving the TRIM command, the lifetime analyzer can compute the lifetime of the corresponding data using the recorded information. Note that, the same PC

may generate multiple data streams with different lifetimes. We take the average lifetime as the PC's lifetime.

3.2.3 Mapping PCs to SSD streams

The last step in PCStream is to map a group of PCs with similar lifetimes to an SSD stream. This is because each SSD supports a limited number of stream IDs. For example, SSDs used in FStream [6] and AutoStream [7] support only 9 and 16 streams, respectively. To properly group multiple PCs, the PC-to-stream mapper employs a simple 1-D clustering algorithm. In order to cluster PCs with similar lifetimes, the mapper calculates the lifetime difference between PCs. Then, PCs with the smallest lifetime difference are clustered into the same PC group. The mapper repeats this clustering step until all the PCs are assigned to their PC groups. For adapting to changing workloads, reclustering operations should be regularly performed. Since the number of PCs created by applications is not limited, the clustering algorithm must be efficient enough to quickly handle many PCs. Our goal in this work is to confirm the feasibility of using PCs, so we leave those issues as our future work.

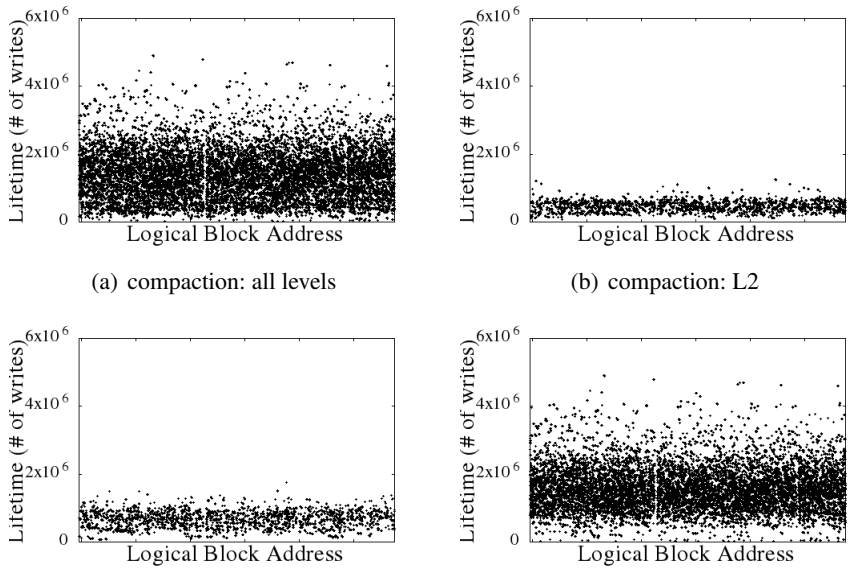
3.2.4 Two-phase stream assignment

For most PCs, their lifetime distributions tend to have small variances. However, we observed that a few outlier PCs which have large lifetime variations. For example, when multiple I/O contexts are covered by the same write system function, the corresponding PC may represent several I/O contexts

whose data lifetimes are quite different. Such a case occurs in the compaction module of RocksDB. RocksDB maintains several levels, L1, ..., Ln, in the persistent storage, except for L0 (or a memtable) stored in DRAM. Once one level, say L2, becomes full, all the data in L2 is compacted to a lower level, i.e., L3. It involves moving data from L2 to L3, along with the deletion of the old data in L2. In the LSM tree [14], a higher level is smaller than a lower level (i.e., the size of (L2) $<$ the size of (L3)). Thus, data stored in a higher level is invalidated more frequently than those kept in lower levels, thereby having shorter lifetimes.

Unfortunately, in the current RocksDB implementation, the compaction step is supported by the same execution path (i.e., the same PC) regardless of the level. Therefore, the PC for the compaction activity cannot effectively separate data with short lifetimes from one with long lifetimes. Fig. 5(a) shows the lifetime distribution collected from the compaction-activity PC. Since this distribution includes lifetimes of data written from all the levels, its variance is large. When we manually separate the single compaction step into several per-level compaction steps, as shown in Figs. 5(b) and 5(c), the lifetime distributions of per-level compaction steps show smaller variances. In particular, L2 and L3 show distinct lifetime distributions from that of L4. Data from L2 and L3 are likely to have shorter lifetimes, while L4 has generally long-lived data as shown in Fig. 5(d).

Since it is difficult to separate data with different lifetimes within the same PC (as in the compaction-activity PC), we devised a two-phase method that decides SSD streams in two levels: the main stream ID in a host level and the substream ID in an SSD level. Conceptually, long-lived data in the



(a) compaction: all levels (b) compaction: L2

(c) compaction: L3 (d) compaction: L4

Figure 5: Lifetime distributions of the compaction activity at different levels.

main stream are moved to its substream to separate from (future) short-lived data of the main stream. Although moving data to the substream may increase WAF, the overhead can be hidden if we restrict the substream move during GC only. Since long-lived data (i.e., valid pages) in a victim block are moved to a free block during GC, they can be moved to the substream by changing the target block. For instance, PCStream assigns the compaction-activity PC *pID* to a main stream *sID* for the first phase. To separate the long-lived data of *pID* (e.g., L4 data) from future short-lived data of *pID* (e.g., L1 data), valid pages of the *sID* are assigned to its substream for the second phase during GC.

3.3 Experimental Results

For our experiments, we have implemented PCStream in the Linux kernel 4.5. For an objective evaluation, we compared PCStream with three existing schemes: Baseline, Manual [4], and AutoStream [7]. Baseline stands for a legacy SSD that does not support a multi-stream feature. Manual is a RocksDB implementation which is manually optimized for multi-streamed SSDs. AutoStream is an LBA-based data separation technique which is implemented at the device driver layer. To understand the impact of the two-phase assignment, in addition, we compared PCStream with PCStream* which excluded the two-phase assignment feature.

For benchmarks, we have used three scenarios of `db_bench` of RocksDB: Update-Random (`UR`), Append-Random (`AR`), and Fill-Random (`FR`) scenarios. For key-value pairs already stored in the SSD, `UR` updates values for random keys, creating many read-modify-writes in the SSD. `AR` is similar to `UR`, except that it performs the update of values for growing keys. `FR` writes key-value pairs to the SSD in a random key order.

3.3.1 Experiments with an SSD emulator

We carried out a set of experiments using an SSD emulator which is based on the open flash development platform [17]. In the SSD emulator, the internal workings of an SSD are simulated using the host's DRAM memory in the kernel level. For our evaluations, we extended the SSD emulator to support a multi-streamed feature (up to 8 streams). Furthermore, we enhanced the garbage collection module of the SSD firmware to support the two-phase

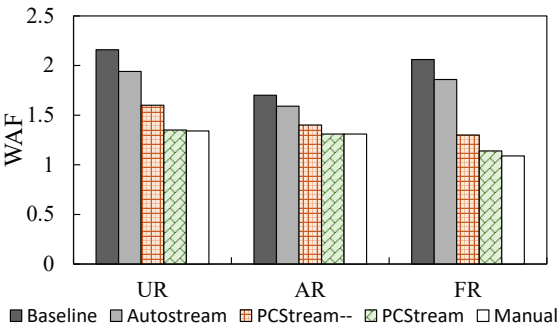


Figure 6: A comparison of WAF on the SSD emulator.

stream management technique. The SSD emulator provided 12 GB capacity with 4 channels and 4 ways, and there were 8192 flash blocks, each of which was composed of 384 4-KB pages.

We compared WAF of the existing techniques with PCStream for the three scenarios, and the result is shown in Fig. 6. PCStream was quite effective in reducing WAF, thus achieving an equivalent level of the WAF reduction as in Manual. For example, both PCStream and Manual reduced WAF by 38% over Baseline for the UR case. PCStream outperformed AutoStream by reducing WAF by 35% on average. Fig. 6 also indicates that the two-phase stream assignment technique is effective. PCStream outperformed PCStream* by 12% on average in the WAF reduction. This additional gain of PCStream over PCStream* came from isolating long- and short-lived data in separate blocks by moving the long-lived data to substreams during GC at the SSD.

In order to better understand how PCStream achieved a high reduction in WAF, we measured per-stream lifetime distributions under each technique

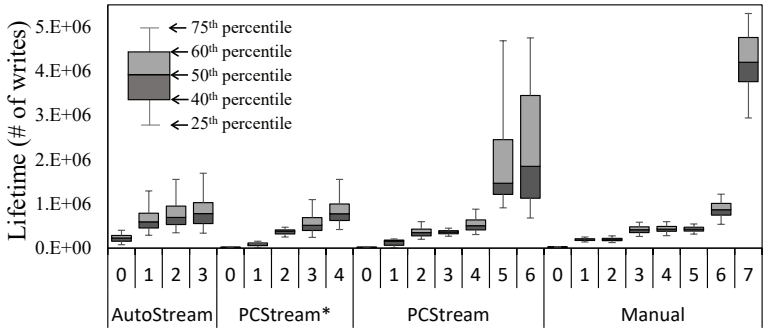


Figure 7: A comparison of per-stream lifetime distributions.

for the UR scenario. Fig. 7 shows a box plot of data lifetimes from the 25th percentile to the 75th percentile.

As shown in Fig. 7, streams in PCStream are divided into two groups, $G1 = \{0, 1, 2, 3, 4\}$ and $G2 = \{5, 6\}$, where $G1$ includes streams with short lifetimes and small variances and $G2$ includes streams with large lifetimes and large variances. Since the GC copy cost is affected by how data in $G1$ and $G2$ are mixed into the same block, PCStream can significantly reduce the GC overhead by avoiding such data mixtures in the same block by separating $G1$ and $G2$ into different streams. On the other hand, in AutoStream, three streams (i.e., streams 1, 2, and 3) show similar lifetime distributions with large variances without a distinct data separation pattern. In Fig. 7, we can also observe the effect of substreams. Streams 3 and 4 of PCStream*, which have large variances in lifetimes, are split into two substreams 5 and 6 in PCStream. This split reduces variances of streams 3 and 4, thus reducing the GC copy cost.

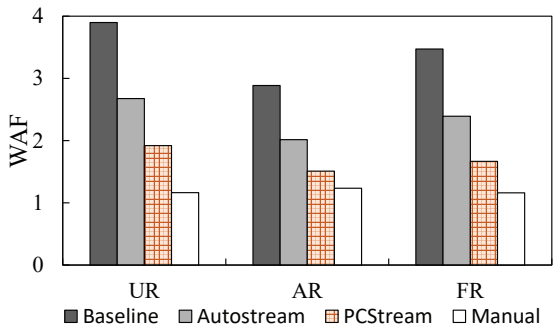


Figure 8: A comparison of WAF on Samsung PM963 SSD.

3.3.2 Experiments with a real-world SSD

In order to evaluate the effect of PCStream in a real SSD, we have conducted experiments using Samsung's PM963 480GB SSD that supports 8 streams. Since it was impossible to implement the two-phase stream assignment in the commercial SSD firmware, we evaluated PCStream* only. To warm up the SSD before running benchmarks, we filled up 90% of the SSD capacity with valid data.

As illustrated in Fig. 8, PCStream* reduced WAF by 28% over AutoStream on average. Note that although PCStream* still outperformed AutoStream in PM963, but a performance gap was smaller over that in the emulated SSD environment. It was difficult to pinpoint why AutoStream worked better in PM963 over in the emulated SSD, but we suspect that some internal features of PM963 (such as a large block size or some implementation details of streams) might have affected the performance of AutoStream.

Chapter 4

Fine-grained Deduplication Technique

4.1 Motivations

The write-requested page is identified whether the contents of the page have already been written and is written to flash memory only if there is no existing duplicate page. When a write-requested page is the exact duplicate of a previously written page, the requested page is not written to flash memory; only the corresponding entry for a mapping table (between the logical address and physical address) is updated. On the other hand, if there is no existing page duplicate in flash memory whose contents are the same as those of requested one, the requested page has to be written to flash memory. However, even for these unique pages, if their redundancy is checked at a sub-page level, say at a quarter of the page size, many sub-pages of these unique pages can be identified as redundant data. In existing techniques based on page-level deduplication, therefore, many duplicate data are written to flash memory even though the same data chunks have already been written.

In order to better understand the effect of fine-grained deduplication on the amount of identified duplicate data, we analyzed how many more chunks can be identified as redundant when the chunk size gets smaller than a single page. For our evaluation, we used five I/O traces, `PC`, `Sensor`,

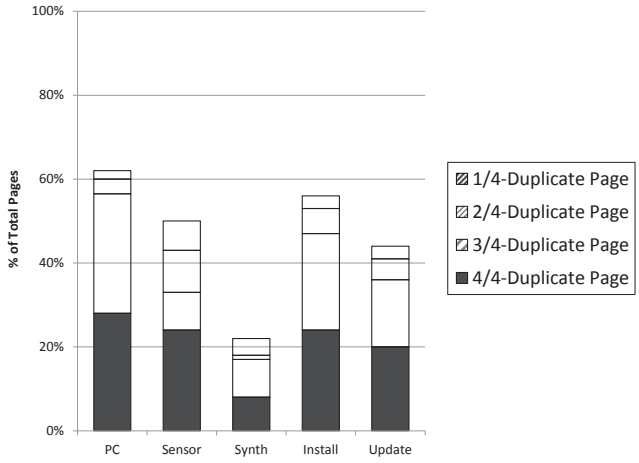


Figure 9: The percentage of pages according to their partial duplicate patterns.

`Synth`, `Install`, and `Update` which are explained in Section 4.3. In our evaluation, the page size was 4 KB and the chunk size was set to 1 KB.

Fig. 9 shows the percentage of the page writes from host, classified by their partial duplicate patterns. We denote that a page is a n/4-duplicate page when n chunks of the page are duplicate chunks. A 4/4-duplicate page is a duplicate page at the page level. In the existing page-based deduplication, only 4/4-duplicate pages can be identified as a duplicate page. As shown in Fig. 9, 4/4-duplicate pages account for only 8% - 28% of total requested pages. For partially duplicate pages, i.e., 1/4-, 2/4- and 3/4-duplicate pages, the page-based deduplication technique is useless. As shown in Fig. 9, pages with 1-3 duplicate chunks account for 14% - 34%. This means that many duplicate data are unnecessarily written to flash memory due to the large chunk size.

We also investigated the amount of data that can be eliminated by data

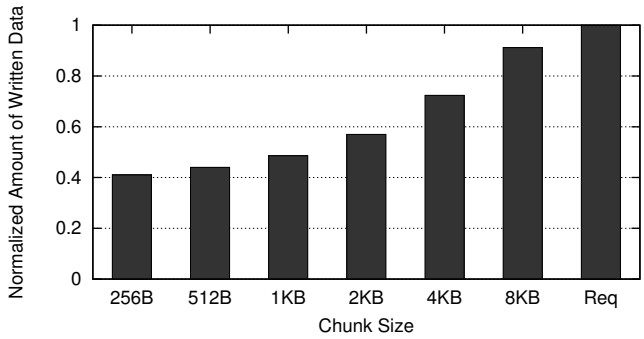


Figure 10: The amount of written data under varying chunk sizes in PC workload.

deduplication while varying the chunk size from 256 B to 8 KB. As shown in Fig. 10, when the chunk size is 1 KB, the amount of data written to flash memory is reduced by 33% over when the chunk size is 4 KB. In particular, when the size of a chunk is 8 KB (i.e., when the physical page size is assumed to be 8 KB), only 10% of requested data are eliminated by data deduplication. This effectively shows that, as the size of a page increases, the overall deduplication ratio, i.e., the percentage of identified duplicate writes, decreases significantly. Since the physical page size of NAND flash memory is expected to increase as the semiconductor process is scaled down [18, 19], it is expected that the deduplication ratio of the existing deduplication technique will be significantly decreased in near future. In order to resolve this problem, the deduplication chunk size of deduplication techniques needs to be smaller than a page size. As depicted in Fig. 10, the deduplication ratio is saturated when the chunk size is 1 KB. Thus, we use it as a default chunk size in the rest of this dissertation.

4.2 Fine-Grained Deduplication

In order to effectively incorporate fine-grained deduplication into flash-based SSDs, two key technical issues must be addressed properly. First, fine-grained deduplication requires a larger memory space than a coarse-grained one because it needs to keep more metadata in memory to find small-size duplicate data. Second, in fine-grained deduplication, unique data segments from partially duplicated pages can be scattered across several physical pages, which may seriously degrade the overall read performance. The proposed FineDedup technique is designed to take full advantage of fine-grained deduplication with small memory overhead as well as a low read performance penalty.

In this section, we describe our proposed FineDedup technique in detail. We first explain the overall architecture of FineDedup and describe how FineDedup handles read and write requests in Section 4.2.1. In Section 4.2.2 and 4.2.3 we introduce a read performance penalty and memory overheads caused by FineDedup, respectively, and explain how these problems can be resolved in FineDedup.

4.2.1 Overall Architecture of FineDedup

Fig. 11 shows an overall architecture of FineDedup with its main components. Upon the arrival of a write request, FineDedup stores requested data temporarily in an on-device buffer, which is managed by an LRU algorithm. When the requested data are evicted from the buffer, FineDedup divides the data into several chunks. (Note that the chunk size is 1 KB in this work, but a different size of chunks can be used as well in FineDedup.)

For each chunk, FineDedup computes a fingerprint, using a collision-resistant hash function. In this work, we use an MD6 hash function [21], which is one of the well-known cryptographic hash functions. A fingerprint is used as a unique ID that represents the contents of a chunk. FineDedup has to compute more fingerprints than the existing deduplication schemes because of its small chunk size. To reduce the hash calculation time, FineDedup uses multiple hardware-assisted hash engines for parallel hash calculations. In our FPGA (ML605) implementation of the MD6 hash function, it took about 10 μs to compute a fingerprint using a hardware accelerator. Considering a long write latency (e.g., 1.2 ms) of NAND flash memory, the time overhead of computing fingerprints can be considered negligible.

After fingerprinting, each fingerprint is looked up in the dedup table which maintains the fingerprints of the unique chunks previously written to flash memory. Each entry of the dedup table is composed of a key-value pair, $\{$*fingerprint*, *location*$\}$, where the location indicates a physical address in which the unique chunk is stored. If the same fingerprint is found, it is not necessary to write the chunk data because the same chunk is already stored in flash memory. Instead, FineDedup updates the mapping table so that the corresponding mapping entry points to the unique chunk previously written. Unlike existing page-based deduplication techniques, FineDedup handles all the data in the unit of a chunk. For this reason, FineDedup must maintain a chunk-level mapping table that maps a logical chunk address to a physical chunk in flash memory. Because of its finer mapping granularity, the chunk-level mapping table is much larger than the existing page-level mapping table. To reduce the memory space for maintaining the chunk-

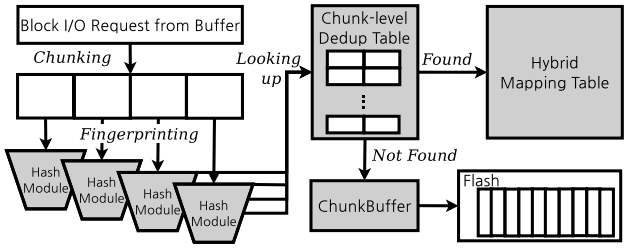


Figure 11: An overview of the proposed FineDedup technique.

level mapping table, FineDedup uses a hybrid mapping strategy, which is described in Section 4.2.3 in detail.

If there is no matched fingerprint in the dedup table, FineDedup stores the chunk data in a *chunk buffer* temporarily. This temporary buffering is necessary because the unit of I/O operations of flash memory is a single page. The chunk buffer stores the incoming chunk data until there are four chunks, and evicts them to flash memory at once. FineDedup then updates the mapping table so that the corresponding mapping entries indicate newly written chunks. The new fingerprints of the evicted chunks are finally inserted into the dedup table with their physical location.

When a read request arrives, FineDedup reads all the chunks that belong to the requested page from flash memory, and then transfers the read data to the host system. The physical addresses of the chunks can be obtained by consulting to the mapping table. In FineDedup, four chunks in the same logical page can be scattered across different physical pages. In that case, multiple read operations are required to form the original page data, which in turn significantly increases the overall read response time. We explain how FineDedup resolves this problem in the following subsection.

4.2.2 Read Overhead Management

FineDedup effectively reduces the number of pages written to flash memory by using a small-size chunk for deduplication, but it incurs two types of additional overheads, i.e., a read performance overhead and a memory space overhead, which are not observed in the existing deduplication techniques. In this subsection, we first introduce why the read performance overhead occurs in FineDedup, and then explain how FineDedup resolves this problem. In the following subsection, we describe our memory space overhead reduction technique in detail.

The main cause of the read performance degradation is data fragmentation which occurs when data chunks belonging to the same logical page are broken up into several physical pages. Fig. 12 illustrates why data fragmentation occurs in FineDedup. There are two page write requests, *Req 1* and *Req 2*, in Fig. 12. *Req 1* consists of four chunks, 'A', 'B', 'C', and 'D', and *Req 2* is also composed of four chunks, 'E', 'F', 'G', and 'H'. Since 'A' and 'B' of *Req 1* are duplicate chunks, only 'C' and 'D' need to be written to flash memory. Suppose that there is a read request for the page data written by *Req 1*. In that case, FineDedup has to read three pages, i.e., *page 1*, *page 2*, and *page 3*, from flash memory to form the requested data. The read performance penalty can also occur even when there are no duplicate chunks in the requested page. For example, in Fig. 12, *Req 2* has no duplicate chunks in flash memory, thus all the chunks belonging to *Req 2* being written to flash memory. Because a single page write requires four data chunks, 'E' and 'F' of *Req 2* are written to *page 3* together with 'C' and 'D', and 'G'

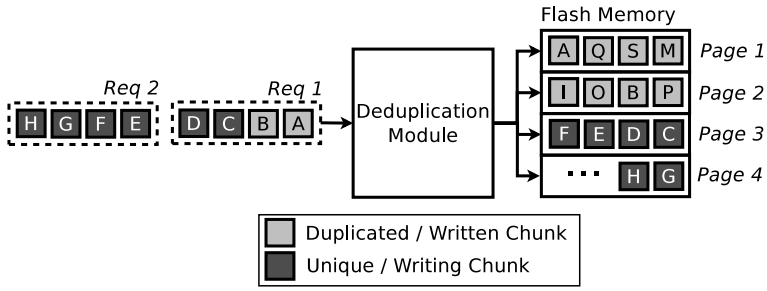


Figure 12: Data fragmentation caused by FineDedup.

and 'H' will be written to *page 4* with other data chunks, as shown in Fig. 12. Thus, when the data written by *Req 2* are read later, both *page 3* and *page 4* must be read from flash memory.

One of the feasible approaches that mitigate the read performance overhead is to employ a chunk read buffer. In our observation, the access frequencies of unique chunks are greatly skewed; that is, a small number of popular chunks account for a large fraction of the total accesses to unique chunks in flash memory. For example, according to our analysis under real-world workloads, top 10% of the unique chunks serve more than 70% of the total data read by a host system. By keeping frequently accessed chunks in a chunk read buffer, therefore, FineDedup can reduce a large number of page read operations sent to flash memory.

On the other hands, we have observed that about 39% of read requests to unique pages actually requires two page read operations. In order to further reduce this read performance penalty, FineDedup uses a chunk packing scheme. The key idea of this scheme is to group chunks belonging to the same logical page in a chunk buffer and then write them to the same phys-

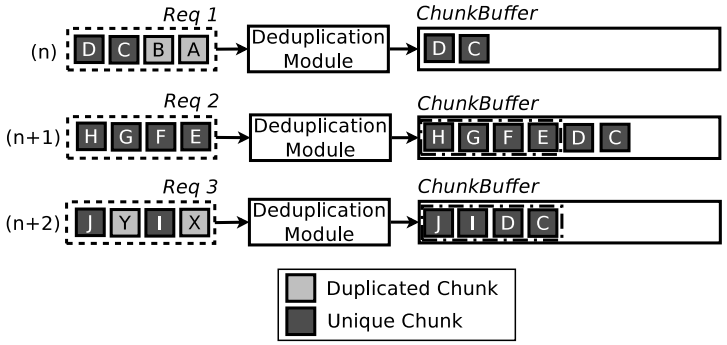


Figure 13: A packing scheme in the *chunk buffer*.

ical page together. Fig. 13 shows an example of our chunk packing scheme when three page write requests, *Req 1*, *Req 2*, and *Req 3*, are consecutively issued from a host system. *Req 1* contains two duplicate chunks 'A' and 'B' and two unique chunks 'C' and 'D'. As expected, only 'C' and 'D' out of four chunks are sent to the chunk buffer. The next request *Req 2* does not have any duplicate chunks, so all of them are moved to the chunk buffer. As depicted in Fig. 13, the chunks 'E', 'F', 'G', and 'H' belong to the same logical page and form single page data. Thus, FineDedup writes them to flash memory together, leaving the chunks 'C' and 'D' in the chunk buffer. When *Req 3* is issued with two more unique chunks 'I' and 'J', 'C' and 'D' along with 'I' and 'J' are written to flash memory. All those chunks can be written to the same physical page together because every chunk of each request is not broken up into two pages.

Note that the main objective of this scheme is to prevent chunks of a unique request to be scattered across multiple pages avoiding unnecessary data fragmentation. In order to directly insert a incoming unique request to

chunk buffer, page-sized free space is managed to be always available in the buffer. When there is no free space for the next request and no suitable chunks of requests to form a single page, the chunks of a partially duplicate request is broken up into two pages. Most partially duplicated requests, however, are 3/4-Duplicate pages as shown in Fig. 9, which means there are many requests of one unique chunk in the chunk buffer. Therefore, we can expect that most requests will be written to the same page even when the size of the chunk buffer is not large since it is not quite difficult to find an appropriate chunk to fit a flash page. In the above example, if we assume the chunk buffer can contain 8 chunks and *Req 3* has three unique chunks, only two chunks of *Req 3* will be written along with existing chunks, 'C' and 'D', leaving the other chunk in chunk buffer. A large chunk buffer provides more chance to avoid the request scattering.

Remaining data in the chunk buffer could be lost when a power failure occurs. Recent enterprise SSDs, such as SM825 model manufactured by Samsung, have a large SDRAM cache (e.g., 256 - 512 MB) and use it as a device buffer. Moreover, they support internal cache power protection through the use of capacitors to flush out information in DRAM to flash memory at the event of power failure [22]. In order to keep the reliability in FineDedup, the remaining data in the chunk buffer can be stored to flash memory during power protection procedure as well as the mapping information of the written page. In conclusion for chunk buffer design, there is a trade-off between potential read performance and reliability depending on the chunk buffer size. The size of the chunk buffer, hence, should be determined according to the characteristics of workloads.

4.2.3 Memory Overhead Management

As mentioned in Section 4.2.1, FineDedup handles requested data in the unit of a chunk. Therefore, FineDedup must maintain a chunk-level mapping table that maps a logical chunk address to a physical chunk address in flash memory. Since the size of a chunk is smaller than that of a page, a chunk-level mapping table is much larger than the page-level mapping table. For example, suppose that the page size is 4 KB and the chunk size is 1 KB. In that case, the size of a chunk-level mapping table is four times larger than that of a page-level mapping table.

In order to reduce the amount of memory space required for a mapping table, FineDedup employs a hybrid mapping table which is composed of two types of mapping tables: a page-level mapping table and a chunk-level mapping table. As depicted in Fig. 9, duplicate pages and unique pages still account for a considerable proportion of the total written pages. For these pages, the page-level mapping table is more appropriate because they can be directly mapped to corresponding pages in flash memory. The chunk-level mapping table is required only for partially duplicate pages.

Fig. 14 shows the overall architecture of the hybrid mapping table used in FineDedup. The primary mapping table (PMT) is maintained in the page level while the secondary mapping table (SMT) is maintained in the chunk level. The entry of the PMT is either a physical page address (PPA in Fig. 14) in flash memory or an index of the SMT (chunk address (CA) in Fig. 14).

If the chunk-level mapping is not necessary for a requested page, for example, unique page or duplicate page, the corresponding entry of the PMT

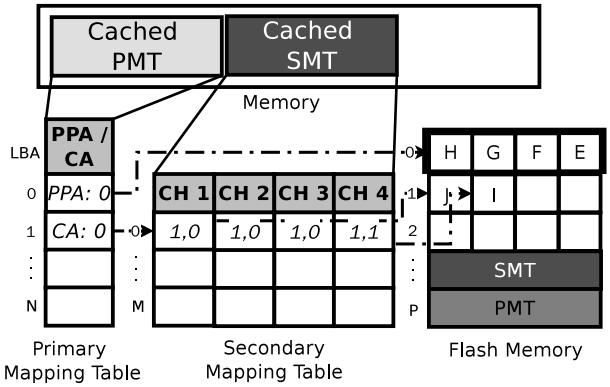


Figure 14: An overview of the demand-based hybrid mapping table.

directly points to the physical address of the newly written page or existing unique page in flash memory, respectively. On the other hand, if a partially duplicate page is requested for writing, FineDedup allocates a new entry in the SMT. As depicted in Fig. 14, each entry of SMT is composed of four fields, each of which points to the physical chunk address in flash memory. FineDedup then updates the new entry so that each field points to the physical chunk address. The corresponding entry of the PMT indicates the newly allocated entry of the SMT.

Using the hybrid mapping table, FineDedup can reduce the amount of memory space for keeping a mapping table. However, the problem of this hybrid mapping approach is that the size of a mapping table can greatly vary according to the characteristics of workloads. For example, if some workloads have many partially duplicate pages, the size of the SMT gets too big. On the other hand, if workloads mostly have unique pages or duplicate pages, the it can be very small. Thus, the hybrid mapping table cannot be directly adopted in real SSD devices whose DRAM size is usually

fixed. To overcome such a limitation, FineDedup adopts a demand-based mapping strategy in which the entire chunk-level mapping table is stored in flash memory while caching only a fixed number of popular entries in DRAM memory. The *Cached PMT* and *Cached SMT* in Fig. 14 represents the cached versions of the PMT and SMT, respectively.

It has been known that the demand-based mapping requires extra page read and write operations [29]. For instance, if a mapping entry for a chunk to be read is not found in the in-memory mapping table, that entry must be read from flash memory while evicting a victim entry to flash memory. The temporal locality present in workload, however, helps keep the number of extra operations small. The mapping information of requests issued in similar times will be stored in the same flash page. Once a mapping page is loaded in memory, hence, most requests issued in similar times are serviced from the mappings in memory.

4.3 Experimental Results

4.3.1 Experimental Settings

In order to evaluate the effectiveness of FineDedup, we performed our experiments using a trace-driven simulator with the I/O traces collected under various applications. The trace-driven simulator modeled the basic operations of NAND flash memory, such as page read, page write and block erase operations, and included several flash firmware algorithms, such as garbage collection and wear-leveling. The proposed FineDedup technique and the existing deduplication techniques were also implemented in our simulator.

Trace	Description	Amount of Writes	Amount of Reads
PC	Web surfing, emailing and editing document, etc.	3.1 GB	40 MB
Sensor	Storing semiconductor fabrication sensor data	2.6 GB	66 KB
Synth	Synthesizing hardware modules	2.5 GB	70 MB
Install	Installing & executing programs (office, DB)	4.9 GB	119 MB
Update	Updating & downloading software packages	3.5 GB	103 MB
M-media	Downloading & playing multimedia files	3.2 GB	36 MB

Table 1: A summary of traces used for experimental evaluations.

For trace collection, we modified the Linux kernel 2.6.32 and collected I/O traces at the level of a block device driver. All the I/O traces include detailed information about the I/O commands sent to a storage device (e.g., the type of requests, logical block addresses (LBA), the size of requests, etc.) as well as the contents of the data sent to or read from a storage device. We recorded I/O traces while running various real-world applications. The detailed descriptions of these I/O traces are summarized in Table 1.

4.3.2 Effectiveness of FineDedup

Fig. 15 shows the amount of data written to flash memory by FineDedup over the existing scheme. The results shown in Fig. 15 are normalized to *RAW_req*, which represents the total amount of data written to flash memory without data deduplication. We assume the page-based deduplication technique as a baseline case. The baseline is denoted by *BL_4KB* for a 4 KB

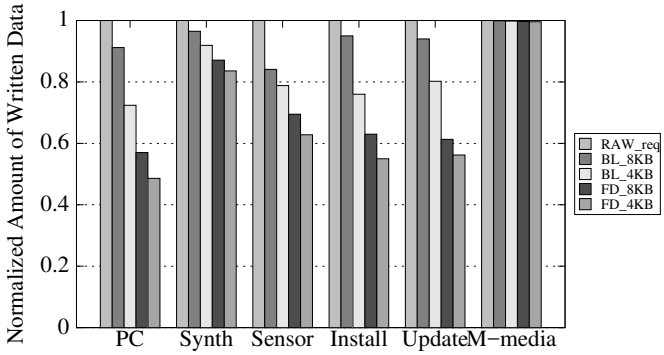


Figure 15: The amount of written data under various schemes.

flash page and *BL_8KB* for a 8 KB flash page. Our FineDedup technique is denoted by *FD_4KB* and *FD_8KB* for a 4 KB flash page and a 8 KB flash page, respectively. The chunk size in FineDedup is set to 1 KB for a 4 KB flash page and 2 KB for a 8 KB flash page.

As we can see in Fig. 15, the effectiveness of deduplication techniques is highly workload-dependent. The amount of data eliminated by the deduplication technique notably increases when FineDedup is applied in most of the traces except `M-media`. When we set the chunk size to one fourth of the flash page size, FineDedup removes on average 16% more duplicate data over *BL_4KB* for a 4 KB flash page. For a 8 KB flash page, it removes more duplicate data, on average by 23% over the existing technique. For `PC`, FineDedup saves 37% flash writes over *BL_8K*. As expected, the benefit of FineDedup mainly derives from the decreased chunk size because it increases the probability of finding and eliminating duplicate data. Especially, `PC` trace shows a large number of update requests with little different data, so FineDedup can effectively identify unchanged data as duplicate while

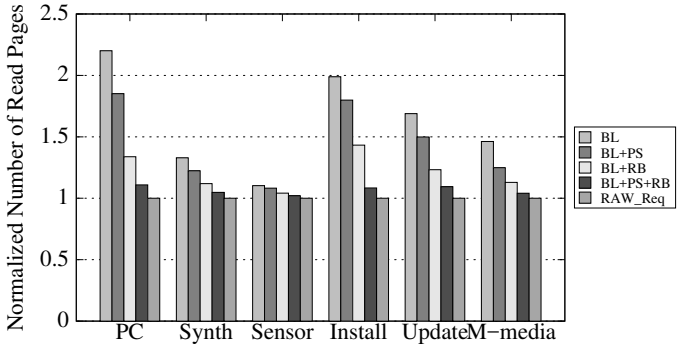


Figure 16: The number of page read operations.

existing deduplication technique regards as unique data. For the `M-media` trace, it is extremely difficult to find duplicate data because the data were already highly compressed. Thus, both the existing deduplication techniques and FineDedup are not effective in reducing the amount of data written to flash memory.

4.3.3 Read Overhead Evaluation

As explained in Section 4.2.2, fine-grained chunking in FineDedup increases the number of page read operations. Fig. 16 shows the normalized number of page read operations compared with the number of read requests in the workloads. *RAW_Req* indicates the number of original page read requests and *BL* refers to the number of page read operations of the baseline FineDedup without employing proposed optimization schemes. *BL+PS*, *BL+RB* and *BL+PS+RB* indicate the number of page reads of FineDedup with the proposed packing scheme, the chunk read buffer, and both, respectively. The size of the chunk read buffer was set to 8 MB and the chunk buffer size was

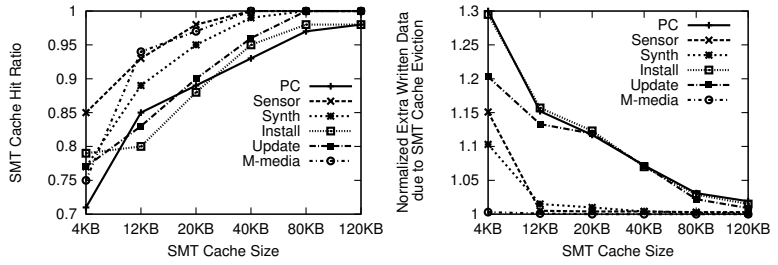
set to 200 KB.

As shown in Fig. 16, employing the chunk read buffer is more effective than the packing scheme for reducing additional page read operations in most workloads. This is because the packing scheme is only effective for the requests containing no duplicate chunks whereas the chunk read buffer can absorb most of the read requests to frequently accessed chunks. FineDedup with both the packing scheme and chunk read buffer incurs on average less than 5% of additional read operations over the existing deduplication technique.

4.3.4 Memory Overhead Evaluation

As explained in Section 4.2.3, chunk level mapping table requires large memory space to handle partially duplicate pages. In FineDedup, we have proposed the demand-based hybrid mapping table to reduce the required memory size for a mapping table without performance degradations. In Fig. 17, the effectiveness of the proposed mapping table is evaluated in terms of the hit ratio and the amount of additional written data with various memory sizes for the cache. Since the PMT of the hybrid mapping table in FineDeup is the same approach as the DFTL [29], which is a well-known demand-based scheme to exploit the page-level mapping, the overhead of PMT can be estimated from the overhead of DFTL. Thus, in order to focus on the overhead of the SMT, we assume that DFTL is employed as the baseline mapping scheme in our evaluation.

Fig. 17(a) shows the hit ratio of the cached SMT. With a 120 KB cache, more than 95% of the mapping table accesses are absorbed. In addi-



(a) Hit ratio of SMT cache

(b) Extra data written due to SMT cache evictions

Figure 17: The effectiveness of the demand-based hybrid mapping table in FineDedup with various cache sizes.

tion, Fig. 17(b) shows extra written data caused by the evicted page entries from the SMT cache. Since mapping table accesses occur in the middle of read/write operations, it is important to reduce the amount of written data from evicted page entries in terms of read/write performance. Similar to the hit ratio, the overhead by the eviction becomes almost negligible when the cache size is set to 120 KB under most workloads.

Note that the memory overhead in the `M-media` trace is not as significant as the other traces when the cache size is very small as shown in Fig. 17, although all of them have a similar number of read requests. It is mainly because the former traces do not benefit from the fine-grained chunking scheme. Since most of data in the `M-media` trace contains unique data, chunk-level mapping table is used only for small amount of data. As a result, FineDedup does not incur a significant memory overhead even when the fine-grained chunking method is not effective.

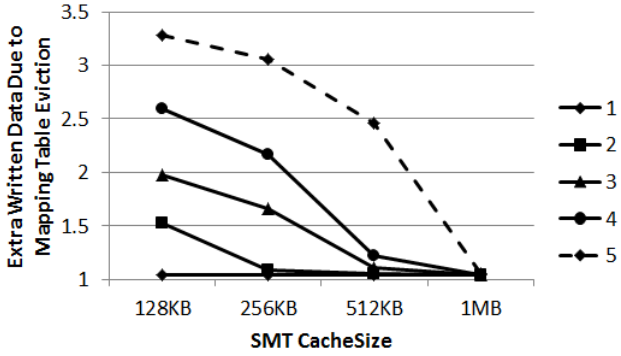


Figure 18: The amount of extra written data due to mapping table eviction for the mixed workload.

4.3.5 Evaluation of the Effectiveness of Cached Mapping Table for Mixed Workloads

The effectiveness of the cached mapping table is evaluated for mixed workloads. Fig. 18 shows the normalized amount of additionally written data due to the mapping table eviction. The mixed traces are composed by accumulating individual traces in the order of `PC`, `Synth`, `Sensor`, `Install`, and `Update`. For example, mixed workload 4 is composed with `PC`, `Synth`, `Sensor`, and `Install`. Although, the cached mapping table is not effective as the number of traces is increased, the performance degradation rate is smaller than the number of workload increasing rate. Moreover, mapping table caching will be effective when the caching memory is big enough to contain the working set of each trace. In the evaluation, the required memory space for the cached mapping table is about 1 MB for the mixed workload. Considering that commercial SSDs have hundreds of GBs of DRAM, the memory overhead of a few MB is not large.

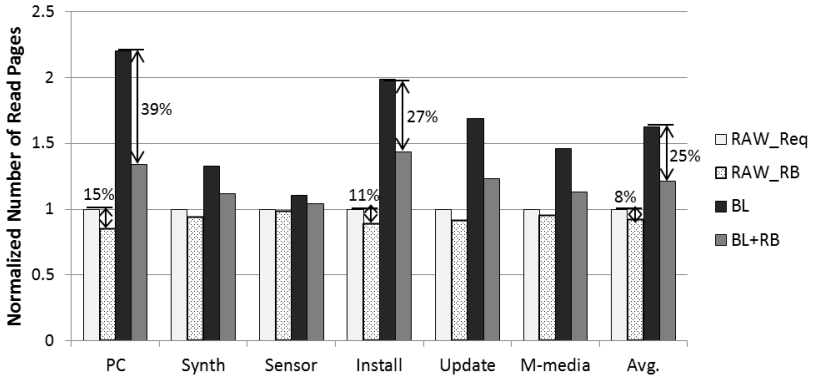


Figure 19: The number of page reads with and without read buffer.

4.3.6 Evaluation of the Effectiveness of the Chunk Read Buffer

The effectiveness of the chunk read buffer between the baseline policy and FineDedup is evaluated. Fig. 19 shows the amount of page reads for the baseline policy and FineDedup with and without the read buffer. The read buffer absorbs about 8% read requests of *RAW_req*, whereas the read requests are reduced by about 25% with the read buffer for FineDedup on average. Based on the analysis, the higher effectiveness of the read buffer in FineDedup is because of the increased memory utilization by deduplication. The read buffer in the baseline policy can absorb read requests for pages that have already been read. Chunk read buffer in FineDedup, however, can also absorb the read requests for deduplicated pages pointed by the hybrid mapping table. Therefore, with the same read buffer size, more read requests can be absorbed in FineDedup.

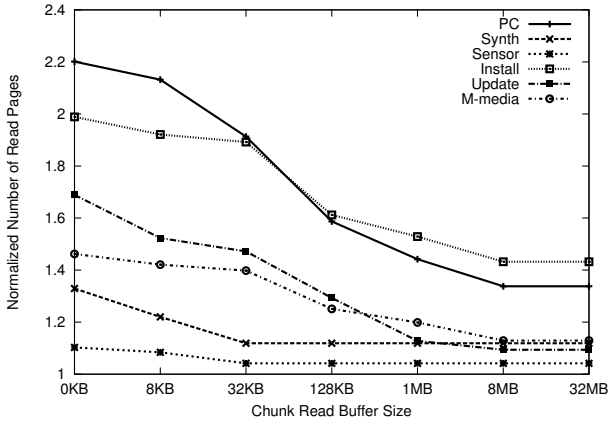


Figure 20: The number of page read operations under varying chunk read buffer sizes.

4.3.7 Sensitivity Study on Chunk Read Buffer Size

In addition to the evaluation of read overhead management shown in Section 4.3.3, we have evaluated that how the size of the chunk read buffer affects the number of read operations. Fig. 20 shows the normalized number of read operations compared with the number of read requests in the workloads while varying the buffer size from 8 KB to 32 MB. The LRU scheme is used to be a replacement scheme in the chunk read buffer. The chunk buffer is not used in this evaluation in order to focus only on the chunk read buffer. When the buffer size gets larger, the number of read operations decreases.

As explained in Section 4.2.2, the access frequencies of unique chunks are greatly skewed. Thus, it is important for read performance to keep those hot chunks in the chunk read buffer. As shown in Fig. 20, the chunk read buffer can absorb most read requests except for `PC` and `Install` traces. In the `PC` and `Install` traces, the skewness of unique read accesses to

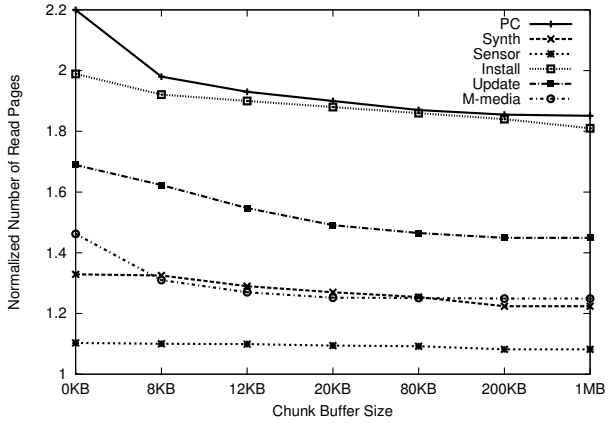


Figure 21: The number of page read operations under varying chunk buffer sizes.

hot chunks was significantly lower than other traces, thus limiting the effectiveness of the chunk read buffer. Since the effectiveness of the chunk read buffer diminishes as its size increases more than 8 MB, in our experiments in Section 4.3.3, a 8 MB chunk read buffer was used.

4.3.8 Sensitivity Study on Chunk Buffer Size

We have also evaluated that how the size of chunk buffer affects the number of read operations. As explained in Section 4.2.2, the chunk buffer separates chunks of unique requests and chunks of partially duplicate requests. Moreover, it also tries not to split chunks of partially duplicate requests into multiple pages. Thus, a larger chunk buffer can reduce more potential extra read operations by preventing chunk splits. Fig. 21 shows the normalized number of read operations compared with the number of read requests in the workloads while varying the chunk buffer size from 8 KB to 1 MB. For this evaluation, the chunk read buffer is not used to focus only on the chunk

buffer.

As shown in Fig. 21, the number of read operations does not decrease much as the chunk buffer size increases. Since most partially duplicated requests in our traces is 3/4-Duplicate pages as explained in Section 4.2.2, it is not difficult to find an appropriate chunk to fit a single flash page. As shown in Fig. 21, however, the benefit of chunk buffer is rather limited. For example, there is no significant saving in the number of read pages for a chunk buffer larger than 200 KB. (In our experiment in Section 4.3.3, we used a 200-KB chunk buffer.)

Chapter 5

Conclusions

5.1 Summary and Conclusions

The cost-per-bit of NAND flash-based solid-state drives (i.e., SSDs) has steadily improved through uninterrupted semiconductor process scaling and multi-leveling so that they are how widely employed in not only mobile embedded systems but also personal computing systems. However, the limited lifetime of NAND flash memory, as a side effect of recent advanced device technologies, is emerging as one of the major concerns for recent high-performance SSDs, especially for datacenter applications.

In this dissertation, we proposed several system-level techniques that improve the lifetime of NAND flash-based storage devices. We first presented data separation technique, called PCStream, for multi-streamed SSDs. Unlike existing stream management techniques, PCStream fully automates the process of mapping data to a stream based on PCs, which work well for append-only workloads as well as update workloads. By exploiting an observation that most PCs are distinguishable from each other in their lifetime characteristics, PCStream allocates each PC to a different stream. When a PC has a large variance in their lifetimes, PCStream refines its stream allocation during garbage collection and moves the long-lived data of the current stream to its substream.

Next, we propose a fine-grained deduplication technique for flash-based SSDs, called FineDedup. By using a fine-grained deduplication unit, the proposed FineDedup technique increases the amount of data eliminated by data deduplication by up to 37% over the existing page-based deduplication technique, extending the SSD lifetime by the same amount. FineDedup inevitably increases the overall read response time because of data fragmentation. By employing a chunk read buffer and a chunk packing scheme, however, the read performance overhead is limited to less than 5% in comparison with the existing deduplication technique. To reduce the memory space required for a chunk-level mapping table, FineDedup adopts a hybrid mapping scheme. Our evaluation results show that FineDedup is effective in improving the SSD lifetime, requiring only about 10 MBs of more memory space in total.

5.2 Future Work

5.2.1 Improving stream mapping method of PCStream

The current version of PCStream can be extended in several directions. For example, we plan to optimize the PC clustering method so that multiple PCs can be better clustered when the number of PCs significantly outnumbers the number of streams. For example, PCStream should be improved in its PC clustering method so that it can work effectively even when there are more PCs than the number of streams. We also plan to evaluate PCStream on real SSDs by implementing the two-phase stream assignment algorithm inside an FTL.

Bibliography

[1] A. Chien and V. Karamcheti, "Moore's Law: The First Ending and a New Beginning," *IEEE Computer Magazine,* vol. 46, no. 12, pp. 48-53, 2013.

[2] J. Jeong, S. Hahn, S. Lee, and J. Kim, "Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[3] SCSI Block Commnads-4 (SBC-4). `http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc4r15.pdf`.

[4] J. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive," in *Proceedings of the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, 2014.

[5] F. Yang, D. Dou, S. Chen, M. Hou, J. Kang, and S. Cho. Optimizing NoSQL DB on Flash: A Case Study of RocksDB. In *Proceedings of IEEE the 15th International Conference on Scalable Computing and Communications (ScalCom'15)*. 2015.

[6] E. Rho, K. Joshi, S. Shin, N. Shetty, J. Hwang, S. Cho. and D. Lee. FStream: Managing Flash Streams in the File System. in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[7] J. Yang, R. Pandurangan, C. Chio, and V. Balakrishnan. AutoStream: Automatic Stream Management for Multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR'17)*, 2017.

[8] K. Ha, and J. Kim. A Program Context-Aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory. In *Proceedings of International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'11)*, 2011.

[9] Facebook. `https://github.com/facebook/rocksdb`.

[10] Apache Cassandra. `http://cassandra.apache.org`.

[11] C. Gniady, A. Butt, and Y. Hu. Program-Counter-Based Pattern Classification in Buffer Caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004.

[12] F. Zhou, J. Behren, and E. Brewer. Amp: Program Context Specific Buffer Caching. In *Proceedings of USENIX Annual Technical Conference (ATC'05)*, 2005.

[13] J. Hsieh, T. Kuo, and L. Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transactions on Storage, vol. 2, no. 1, pp. 22-40*, 2006.

[14] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica, vol. 33, no. 4, pp. 351-385*, 1996.

[15] J. Corbet. Block Layer Discard Requests. https://lwn.net/Articles/293658/.

[16] R. Stallman, and GCC Developer Community. Using the GNU Compiler Collection for GCC version 7.3.0. https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc.pdf.

[17] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[18] M. Goldman et al., "25nm 64Gb 130mm2 3bpc NAND Flash Memory," in *Proc. 3rd International Memory Workshop*, 2011.

[19] Y. Li et al., "128Gb 3b/Cell NAND Flash Memory in 19nm Technology with 18MB/s Write Rate and 400Mb/s Toggle Mode," in *International Solid-State Circuits Conference*, 2012.

[20] S.-W. Lee et al., "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," in *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.

[21] R.L. Rivest et al., "The MD6 hash function - a proposal to NIST for SHA-3," http://groups.csail.mit.edu/cis/md6/, Submission to NIST, 2008.

[22] K. OBrien, "Samsung SSD SM825 Enterprise SSD Review," http://www.storagereview.com/samsung_ssd_sm825_enterprise_ssd_review, 2012.

[23] 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 *Proc. 9th USENIX Conference on File and Storage Technologies*, 2011.

[24] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proc. 9th USENIX Conference on File and Storage Technologies*, 2011.

[25] Z. Chen and K. Shen, "OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash," in *Proc. 14th USENIX Conference on File and Storage Technologies*, 2016.

[26] W. Li, G. Jean-Baptise, J. Riveros, and G. Narasimhan, "CacheDedup: In-line Deduplication for Flash Caching," in *Proc. 14th USENIX Conference on File and Storage Technologies*, 2016.

[27] D. Meister and A. Brinkmann, "dedupv1: Improving Deduplication Throughput using Solid State Drives," in *Proc. IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[28] W. Dong et al., "Tradeoffs in Scalable Data Routing for Deduplication Clusters," in *Proc. 9th USENIX Conference on File and Storage Technologies*, 2011

[29] A. Gupta, Y. Kim and B. Urgaaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Ad-

dress Mappings," in *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.