공학박사학위논문

낸드 플래시 저장장치의 성능 및 수명 향상을 위한 프로그램 컨텍스트 기반 계층 교차 최적화 기법

Program Context based Cross-Layer Optimization Techniques for Improving Performance and Lifetime of NAND Flash-Based Storage Devices

서울대학교 대학원
전기·컴퓨터 공학부
김태진

낸드 플래시 저장장치의 성능 및 수명 향상을 위한 프로그램 컨텍스트 기반 계층 교차 최적화 기법

Program Context based Cross-Layer Optimization Techniques for Improving Performance and Lifetime of NAND Flash-Based Storage Devices

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 I/O context of an application to analyze data lifetime patterns or duplicate data contents patterns.

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. The key motivation behind the proposed technique was that data lifetimes should be estimated at a higher abstraction level than LBAs, so 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 improves the likelihood of eliminating redundant data by introducing sub-page chunk

for partial updates or partial sortings. Based on the data contents pattern analysis, we can avoid unnecessary deduplication work by employing a write program context as a unit of selective deduplication. 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

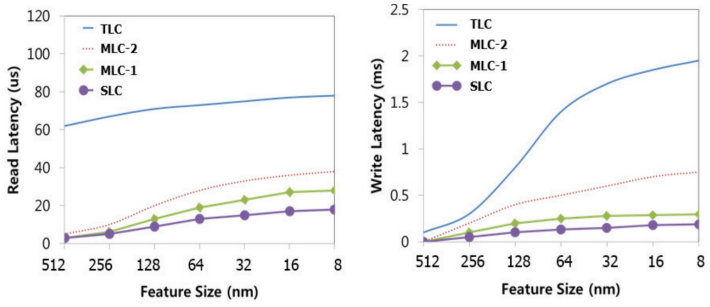
Chapter 1

Introduction

1.1 Motivation

Recently, NAND flash memory is widely used as a storage device from embedded systems to high-end enterprise servers. Because of its many attractive characteristics for mobile storage devices such as light weight, low power consumption, durability, and high performance, it has been widely used for mobile embedded systems. In addition to the advantages, as the cost per byte is falling while the storage capacity is increased, large-capacity NAND flash memory devices such as solid state drives (SSDs) are more commonly employed for high-end desktops and enterprise storage servers.

However, as NAND flash memory technology scales down to 10-nm and below, performance of NAND flash memory is also getting worse. Fig-



(a) A read latency (b) A write latency

Figure 1: Trends of read and write speeds under various NAND flash chips.

ure 1 shows a tendency of read and write speeds of NAND block under various feature sizes. In Figures 1(a) and (b), the x-axis denotes a feature size of NAND chip, and the y-axis represents the latencies of read and write operations, respectively. As shown in Figures 1(a) and (b), both read and write operations are slower as the density of NAND flash memory is increased. In the future, as NAND flash memory is scaled down and the number of bits per cell is increased, the data reliability and performance degradation problems will be even more critical to NAND-based storage systems.

Moreover, the limited endurance of NAND flash memory, which have declined further as a side effect of the recent advanced device technologies, is emerging as major barrier to the wide adoption of SSDs. For example, although the NAND capacity per die doubles every two years, the actual lifetime of SSDs does not increase as much as projected because the maximum number of program/erase cycles has declined [2]. In order for SSDs to be widely adopted, the issues concerning NAND endurance should be properly resolved.

1.1.1 Garbage Collection Problem

Because of the performance degradation of high-density NAND flash memory, the overhead of garbage collection (GC) is increased. In NAND flash-based storage systems, garbage collection is required when there are not enough free blocks to write new data because NAND flash memory does not allow an in-place update operation. If data are updated in NAND flash memory, an FTL stores the newly requested data in another page. Since the previously written data remain as invalidated data in the NAND flash

memory, a garbage collection process is triggered by the FTL in order to reclaim the blocks with the invalidated data so that new data can be stored into NAND flash memory. A garbage collection procedure involves several read, write, and erase operations. Since each operation of NAND flash memory is atomic, the FTL cannot process the next request from a file system while an operation is processed during a garbage collection process. Because of the performance degradation of read and write operations in high-density NAND flash memory, the response time of each I/O request can be elevated when the request is conflicted with a garbage collection process, thus decreasing file system performance. In other words, the whole system is likely to be delayed for a longer time compared to lowdensity NAND flash memory due to the slow extra copies and erases during garbage collection processes. Since such problem can be aggravated in the future high-density NAND flash memory, an efficient garbage collection algorithm is becoming more and more important.

In order to minimize the garbage collection overhead, many techniques have been proposed [1]. Regardless of garbage collection algorithms used, moving valid data from selected victim blocks to new blocks during garbage collection takes a significant portion in the total execution time of a garbage collection algorithm. Therefore, reducing the total number of copied data from the victim blocks is a key factor in improving the performance of a garbage collection algorithm. To reduce the amount of copied data from the victim blocks, a common approach is to separate data based on their characteristics so that the number of dead blocks (which have no valid data) or near-dead blocks (which have few valid data) can be increased. The more

dead or near-dead blocks are generated, the more likely that they can be selected as victim blocks during garbage collection, thus reducing the garbage collection overhead.

One of the most widely used data separation heuristics is to classify data based on their update frequency. This data separation technique classifies data based on their write temporal locality, and it treats data with different temporal locality in a different way [3]. The assumption of this technique is that data with high write temporal locality are likely to be updated soon by successive update requests, and hence the number of dead blocks increases if data with high locality are clustered in the same block. The simplest version of this locality-based data separator divides data into two groups, hot data and cold data according to the number of updates in a given time period. By storing hot data in hot blocks, they are more likely to be dead blocks.

1.1.2 Limited Endurance Problem

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.) 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 propor-

tional 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 [4](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, such as the I/O context of dominant I/O activities, 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. 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, we present a write traffic reduction technique to improve the lifetime of SSD by exploiting data similarity of I/O context of an application. We analyze the likelihood of duplicate data for each I/O context and selectively apply deduplication technique to highly duplicated context. By avoiding disturbance of unique context, we can increase the chance of finding duplicate data and decrease the write latency. With the decreased fingerprinting overhead, we propose new deduplication technique to increase the overall deduplication ratio[1] by introducing sub-page chunk. It resolves

[1] The percentage of identified duplicate writes

technical difficulties caused by its finer granularity, i.e., increased memory requirement and read response time.

1.3 Contributions

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

- We showed that data lifetime and duplicate data have different characteristics depending on the I/O activity. We also implemented program context extraction to effectively use the I/O activity for optimization.
- We propose a fully automatic stream management technique, PCStream, which can work efficiently for general I/O workloads with heterogeneous write characteristics. PCStream is based on the key insight that stream allocation decisions should be made on dominant I/O activities. By identifying dominant I/O activities using program contexts, PCStream fully automates the whole process of stream allocation within the kernel
- We propose a selective deduplication using program contexts with 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). Moreover, FineDedup analyzes the likelihood of finding duplicate data for each PCs. For low dedup ratio PCs, FineDedup skips the deduplication step to avoid unnecessary fingerprinting overhead.

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

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 background for multi-streamed SSDs and data deduplication techniques as well as 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.

Chapter 3 presents the definition of program contexts and how to extract them. Also, distinct lifetime patterns for PCs and duplicate data patterns are shown.

In Chapter 4, 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. By exploiting distinct lifetime patterns of I/O activities, we can achieve high reduction in WAf.

Chapter 5 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

Background

2.1 NAND Flash Memory System Software

In order to overcome the physical limitations of NAND flash memory, such as the *erase-before-write* restriction and the limited P/E cycles, a special software layer, called a *flash translation layer* (FTL), is usually used in NAND flash memory-based storage systems [1]. The FTL emulates a normal block device on top of NAND flash memory, thus enabling users to use NAND flash memory as if they use block device such as hard disk drives. The FTL is charge of address mapping, garbage collection, and wear-leveling. The address mapping function maps a logical block address (LBA) from a host system to a physical block address (PBA) in NAND flash memory. When an update request occurs, the FTL newly allocates a new free page to the request in NAND flash memory. This update process is called out-place update. The location information of the newly allocated page are maintained in the page mapping table which keeps track of mapping information between LBA and PBA. The old versions of newly written data remain invalid in the original location. In order to maintain free space in NAND flash memory, The FTL has to perform a garbage collection process which reclaims the invalid pages in NAND flash memory. Finally, the wear-leveling procedure induces all blocks in NAND flash memory to be evenly

erased, thus preventing frequently erased blocks from being repidly worn out than other blocks.

2.2 NAND Flash-Based Storage Devices

Theoretically, a NAND flash chip with an 8-bit serial bus provides only 40 MB/s for reads and 13 MB/s for writes. This means that the bandwidth of a single flash chip is seriously limited. Moreover, this performance is further reduced with MLC NAND flash memory. In order to overcome the limited performance of a single flash chip, flash-based storage devices utilize the parallelism of multiple NAND flash chips.

Figure 2 shows a simplified diagram of typical NAND flash-based storage devices, consisting of a processor, flash controller, several flash memory packages, and host interface logic. The FTL running on the processor receives host commands (e.g., reads and writes) through the host interface module from the host system, and then issues several flash I/O commands to the flash controller. The flash controller handles multiple I/O commands si-

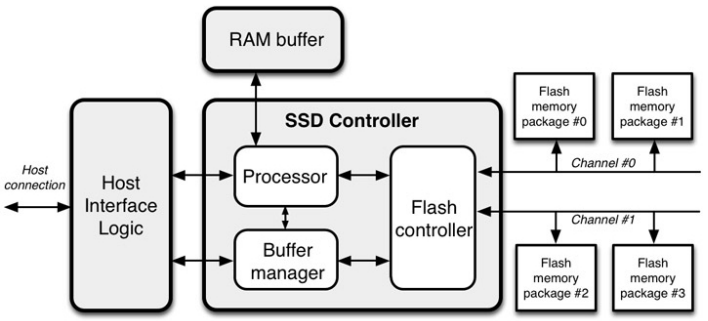


Figure 2: A simplified diagram of typical SSD.

multaneously. Thus, it is possible to achieve much higher performance than utilizing a single NAND flash chip, providing the aggregate bandwidth of multiple flash chips with the host system.

2.3 Multi-stream Interface

At the heart of the garbage collection problems of SSD are the issues of how to predict the lifetime of data written to the SSD and how to ensure that data with similar lifetime are placed in the same erase unit. Kang et. al. [20], proposed multi-streaming, an interface that directly guides data placement within the SSD, separating the two issues. Authors argue that the host system should (and can) provide adequate information about data lifetime to the SSD. It is the responsibility of the SSD, then, to place data with similar lifetime (as dictated by the host system) into the same erase unit.

The design introduces the concept of stream. A stream is an abstraction of SSD capacity allocation that stores a set of data with the same lifetime expectancy. An SSD that implements the proposed multi-stream interface allows the host system to open or close streams and write to one of them. Before writing data, the host system opens streams (through special SSD commands) as needed. Both the host system and the SSD share a unique stream ID for each open stream, and the host system augments each write with a proper stream ID. A multi-streamed SSD allocates physical capacity carefully, to place data in a stream together and not to mix data from different streams.

Figure 3 shows the example of block allocation with and without multi-

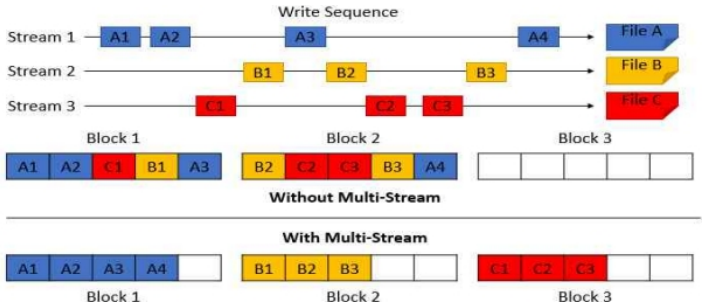


Figure 3: Block allocation with and without multi-stream.

stream. Without Multi-stream, data are written in the order in which they are received. As a result, different lifetimes of data are mixed in the same block. It incurs high valid page copy overhead for garbage collection due to the different invalidation time. However, with Multi-stream, data with same lifetime are separated in a different block so that they are likely to be invalidated together that results low garbage collection overhead. In order to maximize the effect of the multi-streamed SSD, identifying similar lifetime data is the most important and difficult.

2.4 Inline Data Deduplication Technique

Inline deduplication saves the lifetime of SSDs more than offline deduplication [5]. The lifetime of SSD flash cells is limited to a specified small number of writes and block erasures (e.g., around 5,000 times in MLC SSD). Inline deduplication minimizes the number of writes to the SSDs by writing only unique data blocks. However, offline deduplication requires writing all duplicate and unique blocks to the SSDs first. Then during the idle time, stored data are read, and the unique ones are written back to respective

SSDs.

Inline data deduplication typically consists of four steps. First, the deduplication module receives a write request with data ($Data_a$) and logical block address (LBA_a). Second, $Data_a$ are chunked into fixed-sized blocks or variable-sized blocks. In this dissertation, we use fixed-sized chunking which has a lower CPU utilization requirement and has been commercially used in primary storage [5]. Third, the module calculates a signature (Sig_a) for the data. Fourth, the module looks up Sig_a in the deduplication tables. If the signature matches another signature (Sig_b) that was already stored in the deduplication tables, the received write request is considered duplicate ($Data_a = Data_b$). Then Dataa will not be written to the storage devices and the module only updates the deduplication tables. The update includes recording a mapping from LBA_a to the physical block address (PBA_b) of the matched signature. For cases that step four cannot find Sig_a in the table, the write request is considered unique. Therefore, in addition to updating the deduplication tables, $Data_a$ are written to the storage devices.

2.5 Related Work

2.5.1 Data Separation Techniques for Multi-streamed SSDs

There are several research for detecting data temperature. Park, et al. [6] uses multiple bloom filters to identify hot data in the device layer. Stoica, et al. [7] propose new data placement algorithms to improves flash write performance by estimating data update frequencies. Luo, et al. [8] 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.

There have been many studies for multi-streamed SSDs [20, 21, 22, 23, 24, 46]. Kang *et al.* first proposed a multi-streamed SSD that supported manual stream allocation for separating different types of data [20]. Yang *et al.* showed that a multi-streamed SSD was effective for separating data of append-only applications like RocksDB [21]. Yong *et al.* presented a virtual stream management technique that allows logical streams, not physical streams, to be allocated by applications. Unlike these studies that involve modifying the source code of target programs, PCStream automates the stream allocation with no manual code modification.

Rho *et al.* proposed a stream management technique, called FStream, at the file system layer [23]. In FStream, metadata, journal data, and user data that may have different lifetime characteristics were allocated to separate streams. Since FStream was implemented as a part of a file system, it was not able to directly detect application's I/O behaviors. Also, it may be hard to be deployed in practice due to a strong dependence on file system-specific implementation details. PCStream, on the other hand, efficiently exploits programs' I/O behaviors using PCs with no file system-specific modifica-

tions.

Yang *et al.* presented an automatic stream management technique at the block device layer [24]. Similar to hot-cold data separation technique used in FTLs, it approximates the data lifetime of data based on update frequencies of LBAs. The applicability of this technique is, however, quite limited to in-place update workloads only. PCStream has no such limitation on the workload characteristics, thus effectively working for general I/O workloads including append-only, write-once as well as in-place update workloads.

Ha *et al.* proposed an idea of using PCs to separate hot data from cold one in an FTL layer [29]. Kim *et al.* extended it for multi-streamed SSDs [46]. Unlike these work, our study treats the PC-based stream management problem in a more complete fashion by (1) pinpointing the key weaknesses of existing multi-streamed SSD solutions, (2) extending the effectiveness of PCs for more general I/O workloads including write-once patterns, and (3) introducing internal streams as an effective solution for outlier PCs. Furthermore, PCStream exploits the globally unique nature of a PC signature for supporting short-lived applications that runs frequently.

2.5.2 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 [54, 55]. 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 [50].

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 [51]. 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 [49], 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) [54, 55]. 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 [58, 59].

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 [54] proposed CAFTL to enhance the endurance of SSDs with a set of acceleration techniques to reduce runtime overhead. A. Gupta [55] 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 [56] proposed OrderMergeDedup which orders and merges the deduplication metadata with data writes to realize failure-consistent storage with deduplication. W. Li [57] 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 elimi-

nating duplicate data. This approach can complement the limitation of existing fixed-size deduplication techniques, which exhibit a relatively low amount of removed writes in comparison with variable-size deduplication.

2.5.3 Program Context based Optimization Techniques for Operating Systems

History-based prediction techniques exploit the principle that most programs exhibit certain degrees of repetitive behavior. For example, subroutines in an application are ckalled multiple times, and loops are written to process a large amount of data. The challenge in making an accurate prediction is to link the past behavior (event) to its future reoccurrence. In particular, predictors need the program context of past events so that future events about to occur in the same context can be identified. The more accurate context information the predictor has about the past and future events, the more accurate prediction it can make about future program behavior.

A key observation made in computer architecture is that a particular instruction usually performs a very unique task and seldom changes behavior, and thus program instructions provide a highly effective means of recording the context of program behavior. Since the instructions are uniquely described by their program counters (PCs) which specify the location of the instructions in memory, PCs offer a convenient way of recording the program context.

One of the earliest predictors to take advantage of the information provided by PCs is branch prediction. The PC of the branch instruction uniquely

identifies the branch in the program and is associated with a particular behavior, for example, to take or not to take the branch. Branch prediction techniques correlate the past behavior of a branch instruction and predict its future behavior upon encountering the same instruction.

The success in using the program counter in branch prediction was noticed and the PC information has been widely used in other predictor designs in computer architecture. Numerous PC-based predictors have been proposed to optimize energy [9], cache management [10] , and memory prefetching [11]. For example, PCs have been used to accurately predict the instruction behavior in the processor's pipeline which allows the hardware to apply power reduction techniques at the right time to minimize the impact on performance [9] . In Last Touch Predictor [10], PCs are used to predict which data will not be used by the processor again and free up the cache for storing or prefetching more relevant data. In PC-based prefetch predictors [11] , a set of memory addresses or patterns are linked to a particular PC and the next set of data is prefetched when that PC is encountered again.

Moreover, PCC [27] considers the opportunity and viability of PC-based prediction techniques in operating systems design. In particular, they consider the buffer cache management problem, which shares common characteristics with hardware cache management as they essentially deal with different levels of the memory hierarchy. The basic idea is to separate access streams by the program context (identified by the function call stack) when the I/O access is made, with the assumption that a single program context is likely to access disk files with the same pattern in the future.

However, none of the existing PC-based approach focus of the storage devices. They use iterative access characteristics according to the PC at the operating system level, e.g., cache management. Based on our analysis on PCs, program context can be a good candidate for optimizing performance and lifetime of SSDs.

Chapter 3

Capturing Dominant I/O Activities of an Application by Program Contexts

3.1 Definition and Extraction of Program Context

In developing an efficient optimization technique for general I/O workloads, our key insight was that in most applications, the overall I/O behavior of applications is decided by a few dominant I/O activities (*e.g.*, logging and flushing in RocksDB). Moreover, data written by dominant I/O activities tend to have distinct I/O patterns. Therefore, if such dominant I/O activities of applications can be automatically detected and distinguished each other in an LBA-*oblivious* fashion, an optimization technique can be developed for varying I/O workloads including append-only workloads.

In this dissertation, we argue that a program context can be used to build an efficient general-purpose classifier of dominant I/O activities.Here, a PC represents an execution path of an application which invokes write-related system call functions such as `write()` and `writev()`. There could be various ways of extracting PCs, but the most common approach [27, 28] is to represent each PC with its PC signature which is computed by summing program counter values of all the functions along the execution path

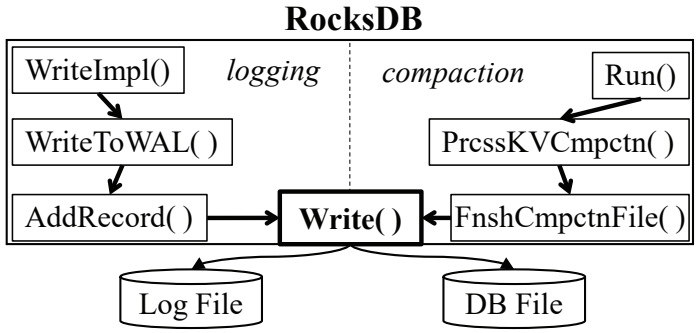


Figure 4: An illustration of (simplified) execution paths of two dominant I/O activities in RocksDB.

which leads to a write system call.

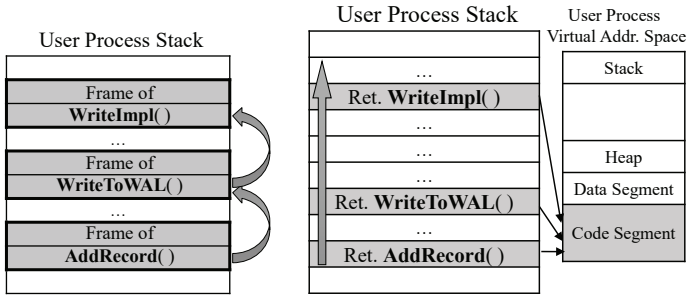
In RocksDB, dominant I/O activities include logging, flushing and compaction. Since these I/O activities are invoked through different function-call paths, we can easily identify dominant I/O activities of RocksDB using PCs. For example, Fig. 4 shows (simplified) execution paths for logging and compaction in RocksDB. The sum of program counter values of the execution path `WriteImpl()`→`WriteToWAL()`→ `AddRecord()` is used to represent a PC for the logging activity while that of the execution path `Run()`→`ProcessKeyValueCompaction()`→`FinishCompactionFile()` is used for the compaction activity. In SQLite, there exist two dominant I/O activities which are logging and managing database tables. Similar to the RocksDB, SQLite writes log files and database files using different execution paths. In GCC, there exist many dominant I/O activities of creating various types of temporal files and object files.

As mentioned earlier, a PC signature, which is used as a unique ID of each program context, is defined to be the sum of program counters

along the execution path of function calls that finally reaches a write-related system function. In theory, program counter values in the execution path can be extracted in a relatively straightforward manner. Except for inline functions, every function call involves pushing the address of the next instruction of a caller as a return address to the stack, followed by pushing a frame pointer value. By referring to frame pointers, we can back-track stack frames of a process and selectively get return addresses for generating a PC signature. Fig. 5(a) illustrates a stack of RocksDB corresponding to Fig. 4, where return addresses are pushed before calling `write()`, `AddRecord()` and `WriteToWAL()`. Since frame pointer values in the stack hold the addresses of previous frame pointers, we can easily obtain return addresses and accumulate them to compute a PC signature.

The frame pointer-based approach for computing a PC signature, however, is not always possible because modern C/C++ compilers often do not use a frame pointer for improving the efficiency of register allocation. One example is a `-fomit-frame-pointer` option of GCC [36]. This option enables to use a frame pointer as a general-purpose register for performance, but makes it difficult for us to back-track return addresses along the call chains.

We employ a simple but effective workaround for back-tracking a call stack when a frame pointer is not available. When a write system call is made, we scan every word in the stack and checks if it belongs to process's code segment. If the scanned stack word holds a value within the address range of the code segment, it assumes that it is a return address. Fig. 5(b) shows the scanning process. Since scanning the entire stack may takes too



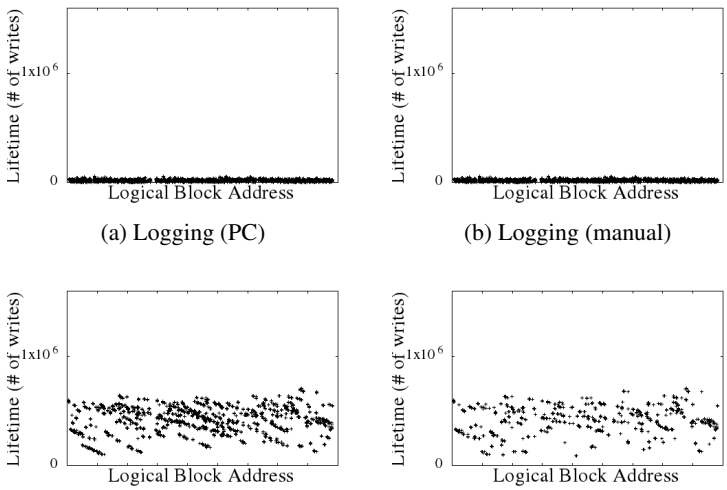
(a) with the frame pointer. (b) without the frame pointer.

Figure 5: Examples of PC extraction methods.

long, we stop the scanning step once a sufficient number of return address candidates are found. In the current implementation, the scanning process stops early once five return address candidates are identified. Even though it is quite ad-hoc, this restricted scan is quite effective in distinguishing different PCs because it is very unlikely that two different PCs reach the same `write()` system call through the same execution subpath that covers five proceeding function calls. In our evaluation on a PC with 3.4 GHz Intel CPU, the overhead of the restricted scan was almost negligible, taking only 300$\sim$400 $n$sec per `write()` system call.

3.2 Data Lifetime Patterns of I/O Activities

In developing automatic stream management technique, our key insight was that in most applications, (regardless of their I/O workload characteristics) 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 life-



(a) Logging (PC) (b) Logging (manual)

(c) Flushing (PC) (d) Flushing (manual)

Figure 6: Data lifetime distributions of different PCs.

time 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 dissertation, 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.

In order to validate our hypothesis that PCs can be useful for estimating 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. 6 illustrates two dominant I/O activities matched be-

tween two methods. As shown in Fig. 6(a) and 6(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. 6(c) and 6(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.

3.3 Duplicate Data Patterns of I/O Activities

We analyzed the relationship between I/O activities and the duplicate data patterns of several applications such as RocksDB and GCC. Although it was difficult to find an I/O activity which is likely to have duplicate data, we can find the relative difference in duplication rates (i.e., the percentage of duplicate requests in total requests) for I/O activities for two reasons.

First, some activity reads and manipulates data stored in the device and write them, e.g., compaction activity of RocksDB. Compaction activity merges multiple files which incurs reading multiple files from the device and writing them to the device after sorting. If the key range is skewed, we may find untouched data sequence (same sequence as in read file) after sorting. The untouched data are regarded as duplicate data in the device. Unlike compaction activity, the probability of finding duplicate data for logging and flushing activites is not very high because data contents of logging and flushing activities are decided by users. In summary, it is better to focus on compaction rather than logging and flushing to find duplicate data.

Second, activities with short-lived data are hard to be referenced, e.g., temporary files during compiling. Most deduplication techniques work in the same way when duplicate data is not found. As a regular write, an updated or trimmed page is invalidated. However, the existing deduplication techniques for SSDs does not consider handling a fingerprint of an invalidated page. The fingerprint value in the dedup table should be removed when the invalid page is erased. We assume the fingerprint is removed right after the page invalidation because removing all the fingerprints of invalid pages before erasing a block make GC overhead more severe. Then, fingerprint of short-lived data can not stay long enough to be deduplicated. In summary, we can avoid deduplication for PCs with short-lived data.

In order to validate our analysis, we measured duplicate rate of aforementioned applications. Table 1 shows the duplicate rates of I/O activities for RocksDB and GCC. For RocksDB, Yahoo! Cloud Serving Benchmark (YCSB) [44] with 12-million keys was used to generate update-heavy workloads. For GCC, a Linux kernel was built 30 times. For each build, 1/3 of source files, which were selected randomly, were modified and recompiled. As explained, logging and flushing activities of RocksDB showed very low

Application	I/O activity	Duplicate requests	Total requests	Duplicate rate
RocksDB	logging	0	39181	0%
	flushing	1092	102831	1%
	compaction	32201	398107	8%
GCC	outputting temp files	0	193834	0%

Table 1: Duplicate rates for I/O activities.

duplicate rates while compaction activity shows higher duplicate rates. Duplicate rate of outputting temporary file activity is also very low. We can exploit this duplicate data patterns of I/O activities in designing an efficient deduplication technique.

Chapter 4

Fully Automatic Stream Management For Multi-Streamed SSDs Using Program Contexts

4.1 Overview

In flash-based SSDs, garbage collection (GC) is inevitable because NAND flash memory does not support in-place updates. Since the efficiency of garbage collection significantly affects both the performance and lifetime of SSDs, garbage collection has been extensively investigated so that the garbage collection overhead can be reduced [12, 13, 14, 15, 16, 3]. For example, hot-cold separation techniques are commonly used inside an SSD so that quickly invalidated pages are not mixed with long-lived data in the same block. For more efficient garbage collection, many techniques also exploit host-level I/O access characteristics which can be used as useful hints on the efficient data separation inside the SSD [17, 18].

Multi-streamed SSDs provide a special interface mechanism for a host system, called streams[1], which data separation decisions *on the host level* can be delivered to SSDs [19, 20]. When the host system assigns two data D_1 and D_2 to different streams S_1 and S_2, respectively, a multi-streamed

[1] In this paper, we use "streams" and "external streams" interchangeably.

SSD places D_1 and D_2 in different blocks, which belong to S_1 and S_2, respectively. When D_1 and D_2 have distinct update patterns, say, D_1 with a short lifetime and D_2 with a long lifetime, allocating D_1 and D_2 to different streams can be helpful in minimizing the copy cost of garbage collection by separating hot data from cold data. Since data separation decisions can be made more intelligently on the host level over on the SSD level, when streams are properly managed, they can significantly improve both the performance and lifetime of flash-based SSDs [20, 21, 22, 23, 24]. We assume that a multi-streamed SSD supports m+1 streams, S_0, ..., S_m.

In order to maximize the potential benefit of multi-streamed SSDs in practice, several requirements need to be satisfied both for stream management and for SSD stream implementation. First, stream management should be supported in a fully automatic fashion over general I/O workloads without any manual work. For example, if an application developer should manage stream allocations *manually* for a given SSD, multi-streamed SSDs are difficult to be widely employed in practice. Second, stream management techniques should have no dependency on the number of available streams. If stream allocation decisions have some dependence on the number of available streams, stream allocation should be modified whenever the number of streams in an SSD changes. Third, the number of streams supported in an SSD should be sufficient to work well with multiple concurrent I/O workloads. For example, with 4 streams, it would be difficult to support a large number of I/O-intensive concurrent tasks.

Unfortunately, to the best of our knowledge, no existing solutions for multi-streamed SSDs meet all these requirements. Most existing techniques [20,

21, 22, 23] require programmers to assign streams at the application level with manual code modifications. AutoStream [24] is the only known automatic technique that supports stream management in the kernel level without manual stream allocation. However, since AutoStream predicts data lifetimes using the update frequency of the logical block address (LBA), it does not work well with append-only workloads (such as RocksDB [25] or Cassandra [26]) and write-once workloads (such as a Linux kernel build). Unlike conventional in-place update workloads where data written to the same LBAs often show strong update locality, append-only or write-once workloads make it impossible to predict data lifetimes from LBA characteristics such as the access frequency.

In this paper, we propose a *fully-automatic* stream management technique, called PCStream, which works efficiently over general I/O workloads including append-only, write-once as well as in-place update workloads. The key insight behind PCStream is that stream allocation decisions should be made at a higher abstraction level where *I/O activities*, not LBAs, can be meaningfully distinguished. For example, in RocksDB, if we can tell whether the current I/O is a part of a logging activity or a compaction activity, stream allocation decisions can be made a lot more efficiently over when only LBAs of the current I/O is available.

In PCStream, we employ a write program context[2] as such a higher-level classification unit for representing I/O activity regardless of the type of I/O workloads. A program context (PC) [27, 28], which uniquely represents

[2]Since we are interested in write-related system calls such as `write()` in Linux, we use *write program contexts* and *program contexts* interchangeable where no confusion arises.

an execution path of a program up to a write system call, is known to be effective in representing dominant I/O activities [29]. Furthermore, most dominant I/O activities tend to show distinct data lifetime characteristics. By identifying dominant I/O activities using program contexts during run time, PCStream can automate the whole process of stream allocation within the kernel with no manual work. In order to seamlessly support various SSDs with different numbers of streams, PCStream groups program contexts with similar data lifetimes depending on the number of supported streams using the k-means clustering algorithm [30]. Since program contexts focus on the semantic aspect of I/O execution as a lifetime classifier, not on the low-level details such as LBAs and access patterns, PCStream easily supports different I/O workloads regardless of whether it is update-only or append-only.

Although many program contexts show that their data lifetimes are narrowly distributed, we observed that this is not necessarily true because of several reasons. For example, when a single program context handles multiple types of data with different lifetimes, data lifetime distributions of such program contexts have rather large variances. In PCStream, when such a program context PC_j is observed (which was mapped to a stream S_k), the long-lived data of PC_j are moved to a different stream $S_{k'}$ during GC. The stream $S_{k'}$ prevents the long-lived data of the stream S_k from being mixed with future short-lived data of the stream S_k.

When several program contexts have a large variance in their data lifetimes, the required number of total streams can quickly increase to distinguish data with different lifetimes. In order to effectively increase the num-

ber of streams, we propose a new stream type, called an internal stream, which can be used only for garbage collection. Unlike external streams, internal streams can be efficiently implemented at low cost without increasing the SSD resource budget. In the current version of PCStream, we create the same number of internal streams as the external streams, effectively doubling the number of available streams.

In order to evaluate the effectiveness of PCStream, we have implemented PCStream in the Linux kernel (ver. 4.5) and extended a Samsung PM963 SSD to support internal streams. Our experimental results show that PCStream can reduce the GC overhead as much as a manual stream management technique while requiring no code modification. Over AutoStream, PCStream improves the average IOPS by 28% while reducing the average WAF by 49%.

4.2 Motivation

4.2.1 No Automatic Stream Management for General I/O Workloads

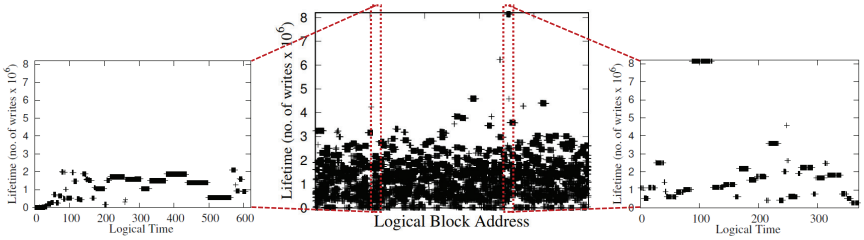
Most existing stream management techniques [20, 21, 22] require programmers to manually allocate streams for their applications. For example, in both ManualStream[3] [20] and [21], there is no systematic guildeline on how to allocate streams for a given application. The efficiency of stream allocations largely depends on the programmer's understanding and expertise

[3]For brevity, we denote the manual stream allocation method used in [20] by ManualStream.

on data temperature (*i.e.*, frequency of updates) and internals of database systems. Furthermore, many techniques also assume that the number of streams is known *a priori*. Therefore, when an SSD with a different number of streams is used, these techniques need to re-allocate streams manually. vStream [22] is an exception to this restriction by allocating streams to virtual streams, not external streams. However, even in vStream, virtual stream allocations are left to programmer's decisions.

Although FStream [23] and AutoStream [24] may be considered as automatic stream management techniques, their applicability is quite limited. FStream [23] can be useful for separating file system metadata but it does not work for the user data separation. AutoStream [24] is the only known technique that works in a fully automatic fashion by making stream allocation decisions within the kernel. However, since AutoStream predicts data lifetimes using the access frequency of the same LBA, AutoStream does not work well when no apparent *locality* on LBA accesses exists in applications. For example, in recent data-intensive applications such as RocksDB [25] and Cassandra [26], majority of data are written in an append-only manner, thus no LBA-level locality can be detected inside an SSD.

In order to illustrate a mismatch between an LBA-based data separation technique and append-only workloads, we analyzed the write pattern of RocksDB [25], which is a popular key-value store based on the LSM-tree algorithm [31]. Fig. 7(a) shows how LBAs may be related to data lifetimes in RocksDB. We define the lifetime of data as the interval length (in terms of the logical time based on the number of writes) between when the data is first written and when the data is invalidated by an overwrite or a TRIM



(b) Lifetime patterns for chunk A (a) Lifetime patterns over LBAs (c) Lifetime patterns for chunk B

Figure 7: Lifetime distributions of append-only workload over addresses and times.

command [32]. As shown in Fig. 7(a), there is no strong correlation between LBAs and their lifetimes in RocksDB.

We also analyzed if the lifetimes of LBAs change under some predictable patterns over time although the overall lifetime distribution shows large variances. Figs. 7(b) and 7(c) show scatter plots of data lifetimes over the logical time for two specific 1-MB chunks with 256 pages. As shown in Figs. 7(b) and 7(c), for the given chunk, the lifetime of data written to the chunk varies in an unpredictable fashion. For example, at the logical time 10 in Fig. 7(b), the lifetime was 1 but it increases about 2 million around the logical time 450 followed by a rapid drop around the logical time 500. Our workload analysis using RocksDB strongly suggests that under append-only workloads, LBAs are not useful in predicting data lifetimes reliably. In practice, the applicability of LBA-based data separation techniques is quite limited to a few cases only when the LBA access locality is obvious in I/O activities such as updating metadata files or log files. In order to support *general* I/O workloads in an automatic fashion, stream management decisions should be based on higher-level information which do not depend on

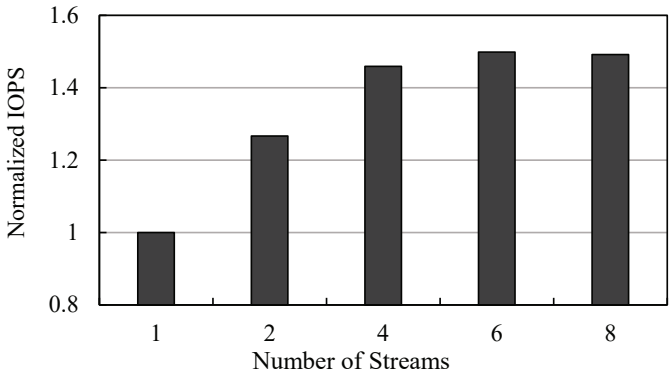


Figure 8: IOPS changes over the number of streams.

lower-level details such as write patterns based on LBAs.

4.2.2 Limited Number of Supported Streams

One of the key performance parameters in multi-streamed SSDs is the number of available streams in SSDs. Since the main function of streams is to separate data with different lifetimes so that they are not mixed in the same block, it is clear that the higher the number of streams, the more efficient the performance of multi-streamed SSDs. For example, Fig. 8 shows how IOPS in RocksDB changes as the number of streams increases on a Samsung PM963 multi-streamed SSD with 9 streams. The `db_bench` benchmark was used for measuring IOPS values with streams manually allocated. As shown in Fig. 8, the IOPS is continuously improving until 6 streams are used when dominant I/O activities with different data lifetimes are sufficiently separated. In order to support a large number of streams, both the SBC-4 and NVMe revision 1.3, which define the multi-stream related specifications, allow up to 65,536 streams [19, 33]. However, the number of streams

supported in commercial SSDs is quite limited, say, 4 to 16 [20, 21, 24], because of several implementation constraints on the backup power capacity and fast memory size.

These constraints are directly related to a write buffering mechanism that is commonly used in modern SSDs. In order to improve the write throughput while effectively hiding the size difference between the FTL mapping unit and the flash program unit, host writes are first buffered before they are written to flash pages in a highly parallel fashion for high performance. Buffering host writes temporarily inside SSDs, however, presents a serious data integrity risk for storage systems when a sudden power failure occurs. In order to avoid such critical failures, in data centers or storage servers where multi-streamed SSDs are used, SSDs use tantalum or electrolytic capacitors as a backup power source. When a main power is suddenly failed, the backup power is used to write back the buffered data reliably. Since the capacity of backup power is limited because of the limited PCB size and its cost, the maximum amount of buffered data is also limited. In multi-streamed SSDs where each stream needs its own buffered area, the amount of buffered data increases as the number of streams increases. The practical limit in the capacity of backup power, therefore, dictates the maximum number of streams as well.

The limited size of fast memory, such as TCM [34] or SRAM, is another main hurdle in increasing the number of streams in multi-streamed SSDs. Since multi-stream related metadata which includes data structures for write buffering should be accessed quickly as well as frequently, most SSD controllers implement data structures for supporting streams on fast

memory over more common DRAM. Since the buffered data is the most recent one for a given LBA, each read request needs to check if the read request should be served from the buffered data or not. In order to support a quick checkup of buffered data, probabilistic data structures such as a bloom filter can be used along with other efficient data structures, for accessing LBA addresses of buffered data and for locating buffer starting addresses. Since the latency of a read request depends on how fast these data structures can be accessed, most SSDs place the buffering-related data structure on fast memory. Similarly, in order to quickly store buffered data in flash chips, these data structure should be placed on fast memory as well. However, most SSD manufacturers are quite sensitive in increasing the size of fast memory because it may increase the overall SSD cost. A limited size of fast memory, unfortunately, restricts the number of supported streams quite severely.

4.3 Automatic I/O Activity Management

In developing an efficient data lifetime separator for general I/O workloads, our key insight was that in most applications, the overall I/O behavior of applications is decided by a few dominant I/O activities (*e.g.*, logging and flushing in RocksDB). Moreover, data written by dominant I/O activities tend to have distinct lifetime patterns. Therefore, if such dominant I/O activities of applications can be automatically detected and distinguished each other in an LBA-*oblivious* fashion, an automatic stream management technique can be developed for widely varying I/O workloads including append-

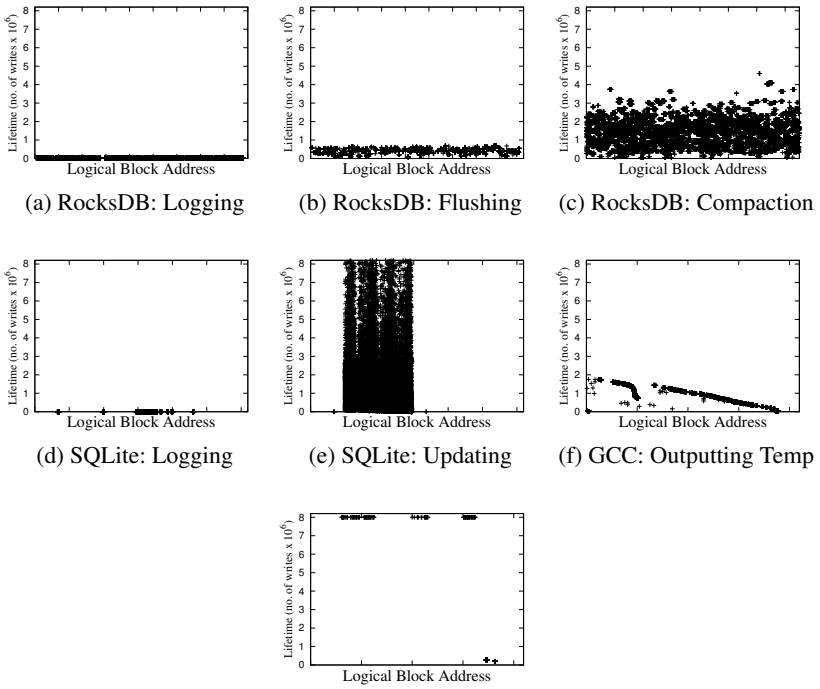
only workloads.

In this paper, we argue that a program context can be used to build an efficient general-purpose classifier of dominant I/O activities with different data lifetimes. Here, a PC represents an execution path of an application which invokes write-related system call functions such as `write()` and `writev()`. There could be various ways of extracting PCs, but the most common approach [27, 28] is to represent each PC with its PC signature which is computed by summing program counter values of all the functions along the execution path which leads to a write system call.

4.3.1 PC as a Unit of Lifetime Classification for General I/O Workloads

In order to illustrate that using PCs is an effective way to distinguish I/O activities of an application and their data lifetime patterns, we measured data lifetime distributions of PCs from various applications with different I/O workloads. In this section, we report our evaluation results for three applications with distinct I/O activities: RocksDB [25], SQLite [35], and GCC [36]. RocksDB shows the append-only workload while SQLite shows a workload that updates in place. Both database workloads are expected to have distinct I/O activities for writing log files and data files. GCC represents an extensive compiler workload (*e.g.*, compiling a Linux kernel) that generates many short-lived temporary files (*e.g.*, `.s`, `.d`, and `.rc` files) as well as some long-lived files (*e.g.*, object files and kernel image files).

To confirm our hypothesis that data lifetimes can be distinguished by



(a) RocksDB: Logging (b) RocksDB: Flushing (c) RocksDB: Compaction

(d) SQLite: Logging (e) SQLite: Updating (f) GCC: Outputting Temp

(g) GCC: Outputting Executable

Figure 9: Data lifetime distributions of dominant I/O activities in RocksDB, SQLite and GCC.

tracking dominant I/O activities and a PC is a useful unit of classification for different I/O activities, we have analyzed how well PCs work for RocksDB, SQLite and GCC. Fig. 9 shows data lifetime distributions of dominant I/O activities which were distinguished by computed PC values. As expected, Fig. 9 validates that dominant I/O activities show distinct data lifetime distributions over the logical address space. For example, as shown in Figs. 9(a)$\sim$9(c), the logging activity, the flushing activity and the compaction activity in RocksDB clearly exhibit quite different data lifetime distributions. While the logged data written by the logging activity have short

lifetimes, the flushed data by the flushing activity have little bit longer lifetimes. Similarly, for SQLite and GCC, dominant I/O activities show quite distinct data lifetime characteristics as shown in Figs. 9(d)∼9(g). As shown in Fig. 9(d), the logging activity of SQLite generates short-lived data. This is because SQLite overwrites logging data in a small and fixed storage space and then removes them soon. Lifetimes of temporary files generated by GCC are also relatively short as shown in Fig. 9(f), because of the write-once pattern of temporary files. But, unlike the other graphs in Fig. 9, data lifetime distributions of Figs. 9(c) and 9(e), which correspond to the compaction activity of RocksDB and the updating activity of SQLite, respectively, show large variances. These *outlier I/O activities* need a special treatment, which will be described in Section 4.4.

Note that if we used an LBA-based data separator instead of the proposed PC-based scheme, most of data lifetime characteristics shown in Fig. 9 could not have been known. Only the data lifetime distribution of the logging activity of SQLite, as shown in Fig. 9(d), can be accurately captured by the LBA-based data separator. For example, the LBA-based data separtor cannot decide that the data lifetime of data produced from the outputting temp activity of GCC is short because temporary files are not overwritten each time they are generated during the compiling step.

4.4 Support for Large Number of Streams

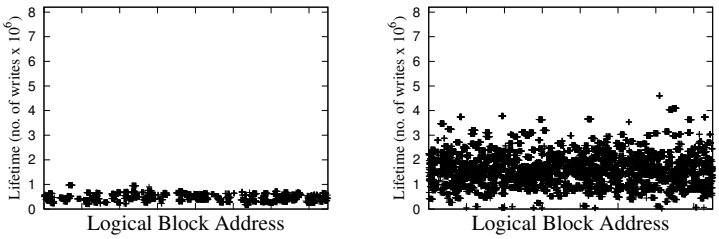
The number of streams is restricted to a small number because of the practical limits on the backup power capacity and the size of fast memory. Since

the number of supported streams critically impacts the overall performance of multi-streamed SSDs, in this section, we propose a new type of streams, called *internal streams*, which can be supported without affecting the capacity of a backup power as well as the size of fast memory in SSDs. Internal streams, which are restricted to be used only for garbage collection, significantly improve the efficiency of PC-based stream allocation, especially when PCs show large lifetime variances in their data lifetime distributions.

4.4.1 PCs with Large Lifetime Variances

For most PCs, their lifetime distributions tend to have small variances (*e.g.*, Figs. 9(a), 9(d), and 9(f)). However, we observed that it is inevitable to have a few PCs with large lifetime variances because of several practical reasons. For example, when multiple I/O contexts are covered by the same execution path, the corresponding PC may represent several I/O contexts whose data lifetimes are quite different. Such a case occurs, for example, in the compaction job 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 [31], 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



(a) RocksDB: L2 Compaction (b) RocksDB: L4 Compaction

Figure 10: Lifetime distributions of the compaction activity at different levels.

of the level. Therefore, the PC for the compaction activity cannot effectively separate data with short lifetimes from one with long lifetimes. Fig. 10(a) and 10(b) show distinctly different lifetime distributions based on the level of compaction: data written from the level 4 have a large lifetime variance while data written from the level 2 have a small lifetime variance.

Similarly, in SQLite and GCC, program contexts with large lifetime variations are also observed. Fig. 9(e) shows large lifetime variances of data files in SQLite. Since client request patterns will decide how SQLite updates its tables, the lifetime of data from the updating activity of SQLite is distributed with a large variance. Similarly, the lifetime of data from the outputting temporary files of GCC can significantly fluctuate as well depending on when the next compile step starts. Fig. 9(g) shows long lifetimes of object files/executable files after a Linux build was completed (with no more re-compiling jobs). However, the lifetime of the same object files/executable files may become short when if we have to restart the same compile step right after the previous one is finished (*e.g.*, because of code changes).

For these *outlier* PCs with large lifetime variations, it is a challenge to

allocate streams in an efficient fashion unless there are more application-specific hints (*e.g.*, the compaction level in RocksDB) are available. As an ad-hoc (but effective) solution, when a PC shows a large variance in its data lifetime, we allocate an additional stream, called an internal stream, to the PC so that the data written from the PC can be better separated between the original stream and its internal stream. In order to support internal streams, the total number of streams may need to be doubled so that each stream can be associated with its internal stream.

4.4.2 Implementation of Internal Streams

As described in Section 4.2.2, it is difficult to increase the number of (normal) streams. However, if we restrict that internal streams are used only for data movements during GC, they can be quite efficiently implemented without the constraints on the backup power capacity and fast memory size. The key difference in the implementation overhead between normal streams and internal streams comes from a simple observation that data copied during GC do not need the same reliability and performance support as for host writes. Unlike buffered data from host write requests, valid pages in the source block during garbage collection have no risk of losing their data from the sudden power-off conditions because the original valid pages are always available. Therefore, even if the number of internal streams increases, unlike normal streams, no higher-capacity backup capacitor is necessary for managing buffered data for internal streams.

The fast memory requirement is also not directly increased as the number of internal streams increases. Since internal streams are used only for GC

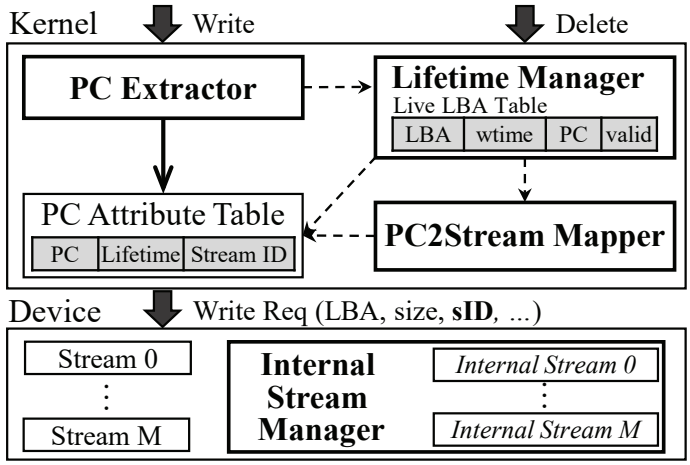


Figure 11: An overall architecture of PCStream.

and most GC can be handled as background tasks, internal streams has a less stringent performance requirement. Therefore, data structures for supporting internal streams can be placed on DRAM without much performance issues. Furthermore, for a read request, there is no need to check if a read request can be served by buffered data as in normal streams because the source block always has the most up-to-date data. This, in turn, allows data structures for internal streams to be located in slow memory. Once an SSD reaches the fully saturated condition where host writes and GC are concurrently performed, the performance of GC may degrade a little because of the slow DRAM used for internal streams. However, in our evaluation, such cases were rarely observed under a reasonable overprovisioning storage capacity.

4.5 Design and Implementation of PCStream

In this section, we explain the detailed implementation of PCStream. Fig. 11 shows an overall architecture of PCStream. The *PC extractor* is implemented as part of a kernel's system call handler as already described in Section 4.3, and is responsible for computing a PC signature from applications. The PC signature is used for deciding the corresponding stream ID[4] from the PC attribute table. PCStream maintains various per-PC attributes in the PC attribute table including PC signatures, expected data lifetimes, and stream IDs. In order to keep the PC attribute table updated over changing workloads, the computed PC signature with its LBA information is also sent to the *lifetime manager*, which estimates expected lifetimes of data belonging to given PCs. Since commercial multi-streamed SSDs only expose a limited number of streams to a host, the *PC2Stream mapper* groups PCs with similar lifetimes using a clustering policy, assigning PCs in the same group to the same stream. Whenever the lifetime manager or the PC2Stream mapper are invoked, the PC attribute table is updated with new outputs from these modules. Finally, the *internal stream manager*, which was implemented inside an SSD as a firmware, is responsible for handling internal streams associated with external streams.

4.5.1 PC Lifetime Management

The responsibility of the lifetime manager is for estimating the lifetime of data associated with a PC. Except for outlier PCs, most data from the same

[4] We call *i* the stream ID of S_i.

PC tend to show similar data lifetimes with small variances.

Lifetime estimation: Whenever a new write request *R* arrives, the lifetime manager stores the write request time, the PC signature, PC_i, and the LBA list of R into the live LBA table. The live LBA table, indexed by an LBA, is used in computing the lifetime of data stored at a given LBA which belongs to PC_i. Upon receiving TRIM commands (that delete previously written LBAs) or overwrite requests (that update previously written LBAs), the lifetime manager searches the live LBA table for a PC signature PC_{found} with the LBA list which includes the deleted/updated LBAs. The new lifetime l_{new} of PC_{found} is estimated using the lifetime of the matched LBA from the live LBA table. The weighted average of the existing lifetime l_{old} for PC_{found} and l_{new} is used to update the PC_{found} entry in the PC attribute table. Note that the written time entry of the live LBA table is updated differently depending on TRIM commands or overwrite requests. The written time entry becomes invalid for TRIM while it is updated by the current time for an overwrite request.

Maintaining the live LBA table, which is indexed by an LBA unit, in DRAM could be a serious burden owing to its huge size. In order to mitigate the DRAM memory requirement, the lifetime manager slightly sacrifices the accuracy of computing LBA lifetime by increasing the granularity of LBA lifetime prediction to 1 MB, instead of 4 KB. The live LBA table is indexed by 1 MB LBA, and each table entry holds PC signatures and written times over a 1 MB LBA range. Because of the coarse-grained indexing, each entry could have multiple signatures and written times, and the same PC could span across multiple entries. If the same PC has different lifetimes, we take

the arithmetic mean as the PC lifetime. To limit the table size, if the table reaches a threshold size, the least recently referenced entry is evicted from the LBA table. Currently, the threshold is set to 64 MB.

PC attribute table: The PC attribute table keeps PC signatures and its expected lifetimes. To quickly retrieve the expected lifetime of a requested PC signature, the PC attribute table is managed through a hash data structure. Each hash entry requires only 12 bytes: 64-bit for a PC signature and 32-bit for a predicted lifetime. The table size is thus small so that it can be entirely loaded in DRAM. From our evaluations, the DRAM size of the PC attribute table was sufficient with several tens or hundreds of KB.

In addition to the main function of the PC attribute table that maintains the data lifetime for a PC, the $memory - resident$ PC attribute table has another interesting benefit for the efficient stream management. Since a PC signature of an I/O activity is virtually guaranteed to be *globally unique* across *all* applications (the uniqueness property), and a PC signature does not change over different executions of the same application (the consistency property), the PC attribute table can capture a long-term history of programs' I/O behaviors. Because of the uniqueness and consistency of a PC signature, PCStream can exploit the I/O behavior of even short-lived processes (*e.g.*, `cpp` and `cc1` for GCC) that are launched and terminated frequently. When short-lived processes are frequently executed, the PC attribute table can hold their PC attributes from their previous executions, thus enabling quick but accurate stream allocation for short-lived processes.

The consistency property is rather straightforward because each PC signature is determined by the sum of return addresses inside a process's

virtual address space. Unless a program's binary is changed after recompilation, those return addresses remain the same, regardless of program's execution. The uniqueness property is also somewhat obvious from the observation that the probability that distinct I/O activities that take different function-call paths have the same PC signature is extremely low. This is even true for multiple programs. Even though they are executed in the same virtual address space, it is very unlikely that I/O activities of diverged programs taking different function-call paths have the same PC. Consequently, this immutable property of the PC signature for a given I/O activity makes it possible for us to characterize the given I/O activity in a long-term basis without a risk of PC collisions.

4.5.2 Mapping PCs to SSD streams

After estimating expected lifetimes of PC signatures, the PC2Stream mapper attempts to group PCs with similar lifetimes into an SSD stream. This grouping process is necessary because while commercial SSDs only support a limited number of streams (*e.g.,* 9), the number of unique PCs can be larger (*e.g.*, 30). For grouping PCs with similar lifetimes, the PC2Stream mapper module uses the k-means algorithm [30] which is widely used for similar purposes. In PCStream, we use the difference in the data lifetime between two PCs as a clustering distance and generates m clusters of PCs for m streams. This algorithm is particularly well suited for our purpose because it is lightweight in terms of the CPU cycle and memory requirement. To quickly assign a proper stream to incoming data, we add an extra field to the PC attribute table which keeps a stream ID for each PC signature. More

specifically, when a new write request comes, a designated SSD stream ID is obtained by referring to the PC attribute table using request's PC value as an index. If there is no such a PC in the table, or a PC does not have a designated stream ID, the request gets default stream ID, which is set to 0.

For adapting to changing workloads, re-clustering operations should be performed regularly. This re-clustering process is done in a straightforward manner. The PC2Stream mapper scans up-to-date lifetimes for all PCs in the PC attribute table. Note that PC's lifetimes are updated whenever the lifetime manager gets new lifetimes while handling overwrites or TRIM requests, as explained in Section 4.5.1. With the scanned information, the PC2Stream mapper recomputes stream IDs and updates stream fields of the PC attribute table. In order to minimize unnecessary overhead of frequent re-clustering operations, re-clustering is triggered when 10% of the PC lifetime entries in the PC attribute table is changed.

4.5.3 Internal Stream Management

As explained in Section 4.4.1, there are a few outlier PCs with large lifetime variances. In order to treat these PCs in an efficient fashion, we devise a two-phase method that decides SSD streams in two levels: the main stream in the host level and its internal stream in the SSD level. Conceptually, long-lived data in the main stream are moved to its internal stream so that (future) short-lived data will not be mixed with long-lived data in the main stream. Although moving data to the internal stream may increase WAF, the overhead can be hidden if we restrict data copies to the internal stream during GC only. Since long-lived data (*i.e.*, valid pages) in a victim

block are moved to a free block during GC, blocks belong to an internal stream tend to contain long-lived data. For instance, PCStream assigns the compaction-activity PC_1 to a main stream S_1 in the first phase. To separate the long-lived data of PC_1 (*e.g.*, L4 data) from future short-lived data of the same PC_1 (*e.g.*, L1 data), valid pages of the S_1 are assigned to its internal stream for the second phase during GC.

We have implemented the internal stream manager with the two-phase method in Samsung's PM963 SSD [38]. To make it support the two-phase method, we have modified its internal FTL so that it manages internal streams while performing GC internally. Since the internal stream manager assigns blocks for an internal stream and reclaims them inside the SSD, no host interface changed is required.

4.5.4 PC Extraction for Indirect Writes

One limitation of using PCs to extract I/O characteristics is that it only works with C/C++ programs that *directly* call write-related system calls. Many programs, however, often invoke write system calls *indirectly* through intermediate layers, which makes it difficult to track program contexts.

The most representative example may be Java programs, such as Cassandra, that run inside a Java Virtual Machine (JVM). Java programs invoke write system calls via the Java Native Interface (JNI) [39] that enables Java programs to call a native I/O library written in C/C++. For Java programs, therefore, the PC extractor shown in Fig. 11 fails to capture Java-level I/O activities as it is unable to inspect the JVM stack from the native write system call which is indirectly called through the JNI. Another exam-

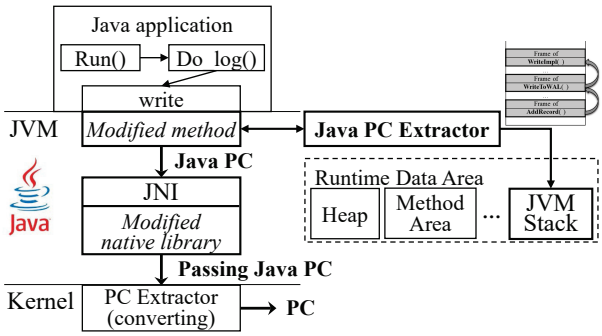


Figure 12: Extracting PCs for JVM.

ple is a program that maintains a write buffer that is dedicated to dealing with all the writes from an application. For example, in MySQL [40] and PostgreSQL [41], every write is first sent to a write buffer. Separate flush threads later materialize buffered data to persistent storage. In that case, the PC extractor only captures PCs of flush threads, not PCs of I/O activities that originally generate I/Os, because the I/O activities were executed in different threads using different execution stacks.

The problem of indirect writes can be addressed by collecting PC signatures *at the front-end interface* of an intermediate layer that accepts write requests from other parts of the program. In case of Java programs, a native I/O library can be modified to capture write requests and computes their PC signatures. Once a native library is modified, PCStream can automatically gather PC signatures without modifying application programs. Fig. 12 illustrates how PCStream collects PC signatures from Java programs. We have modified the OpenJDK [42] source to extract PC signatures for most of write methods in write related classes, such as `OutputStream`. The stack area in the `Runtime Data Areas` of JVM is used to calculate PC

signatures. The calculated PC is then passed to the write system call of the kernel via the modified native I/O libraries.

Unlike Java, there is no a straightforward way to collect PCs from applications with write buffers. This is because the implementation of write buffering is different depending on applications. Additional efforts to manually modify code are unavoidable. However, the scope of this manual modification is limited only to the write buffering code, and application logics themselves don't need to be edited or annotated.

4.6 Experimental Results

4.6.1 Experimental Settings

In order to evaluate PCStream, we have implemented it in the Linux kernel (version 4.5) on a PC host with Intel Core i7-2600 8-core processor and 16 GB DRAM. As a multi-streamed SSD, we used Samsung's PM963 480 GB SSDs. The PM963 SSD supports up to 9 streams; 8 user-configurable streams and 1 default stream. When no stream is specified with a write request, the default stream is used. To support internal streams, we have modified the existing PM963 FTL firmware. For a detailed performance analysis, we built a modified `nvme-cli` [43] tool that can retrieve the internal profiling data from PCStream-enabled SSDs. Using the modified `nvme-cli` tool, we can monitor WAF values and per-block data lifetimes from the extended PM963 SSD during run time.

We compared PCStream with three existing schemes: Baseline, ManualStream [20], and AutoStream [24]. Baseline indicates a legacy SSD that

does not support multiple streams. ManualStream represents a multi-streamed SSD with manual stream allocation. AutoStream represents the LBA-baed stream management technique proposed in [24].

We have carried out experiments with various benchmark programs which represent distinct write characteristics. RocksDB [25] and Cassandra [26] have append-only write patterns. SQLite [35] has in-place update write patterns and GCC [36] has write-once patterns. For more realistic evaluations, we also used mixed workloads running two different benchmark programs simultaneously.

In both RocksDB and Cassandra experiments, Yahoo! Cloud Serving Benchmark (YCSB) [44] with 12-million keys was used to generate update-heavy workloads (workload type A) which consists of 50/50 reads and writes. Since both RocksDB and Cassandra are based on the append-only LSM-tree algorithm [31], they have three dominant I/O activities (such as logging, flushing, and compaction). Cassandra is written in Java, so its PC is extracted by the modified procedure described in Section 4.4. In SQLite evaluations, TPC-C [45] was used with 20 warehouses. SQLite has two dominant I/O activities such as logging and updating tables. In GCC experiments, a Linux kernel was built 30 times. For each build, 1/3 of source files, which were selected randomly, were modified and recompiled. Since GCC creates many temporary files (*e.g.*, `.s`, `.d`, and `.rc`) as well as long-lived files (*e.g.*, `.o`) from different compiler tools, there are more than 20 dominant PCs. To generate mixed workloads, we run RocksDB and GCC scenarios together (denoted by Mixed 1), and run SQLite and GCC scenarios at the same time (denoted by Mixed 2). In order to emulate an aged SSD

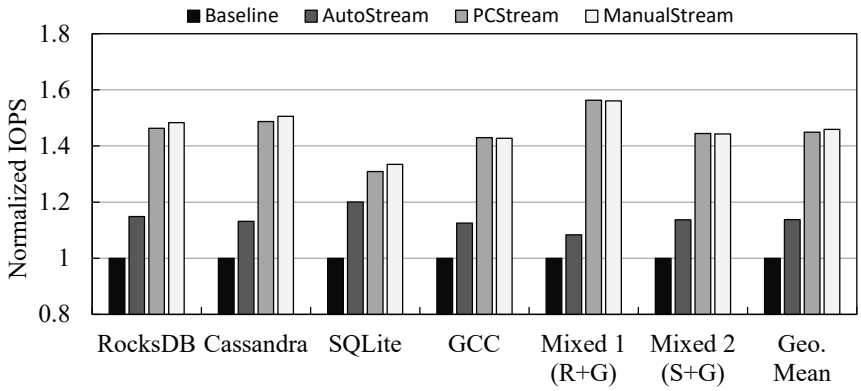


Figure 13: A comparison of normalized IOPS.

in our experiments, 90% of the total SSD capacity was initially filled up with user files before benchmarks run.

4.6.2 Performance Evaluation

We compared IOPS values of three existing techniques with PCStream. Fig. 13 shows normalized IOPS for six benchmarks with four different techniques. For all the measured IOPS values[5], PCStream improved the average IOPS by 45% and 28% over Baseline and AutoStream, respectively. PCStream outperformed AutoStream by up to 56% for complex workloads (*i.e.*, GCC, Mixed1 and Mixed 2) where the number of extracted PCs far exceeds the number of supported streams in PM963. The high efficiency of PCStream under complex workloads comes from two novel features of PCStream: (1) LBA-oblivious PC-centric data separation and (2) a large num-

[5]For RocksDB, Cassandra, and SQLite, the YCSB benchmark and TPC-C benchmark compute IOPS values as a part of the benchmark report. For GCC, where an IOPS value is not measured during run time, we computed the IOPS value as a ratio between the total number of write requests (measured at the block device layer) and the total elapsed time of running GCC.

ber of streams supported using internal streams. AutoStream, on the other hands, works poorly except for SQLite where the LBA-based separation can be effective. Even in SQLite, PCStream outperformed AutoStream by 10%.

4.6.3 WAF Comparison

Fig. 14 shows WAF values of four techniques for six benchmarks. Overall, PCStream was as efficient as ManualStream; Across all the benchmarks, PCStream showed similar WAF values as ManualStream. PCStream reduced the average WAF by 63% and 49% over Baseline and AutoStream, respectively.

As expected, Baseline showed the worst performance among all the techniques. Owing to the intrinsic limitation of LBA-based data separation, AutoStream performs poorly except for SQLite. Since PCStream (and ManualStream) did not depend upon LBAs for stream separations, they performed well consistently, regardless of write access patterns. As a result, PCStream reduced WAF by up to 69% over AutoStream.

One interesting observations in Fig. 14 is that PCStream achieved a lower WAF value than even ManualStream for GCC, Mixed 1, and Mixed 2 where more than the maximum number of streams in PM963 are needed. In ManualStream, DB applications and GCC were manually annotated at offline, so that write system calls were statically bound to specific streams during compile time. When multiple programs run together as in three complex workloads (*i.e.*, GCC, Mixed 1 and Mixed 2), static stream allocations are difficult to work efficiently because they cannot adjust to dynamically changing execution environments. However, unlike ManualStream,

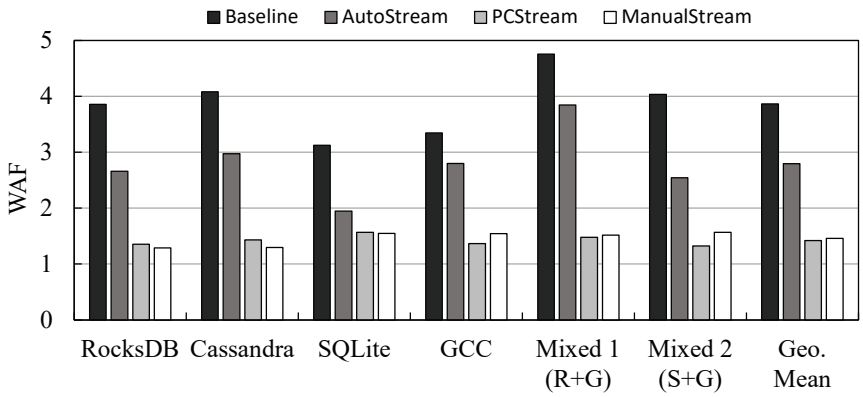


Figure 14: A comparison of WAF under different schemes.

PCStream continuously adapts its stream allocations during run time, thus quickly responding to varying execution environments.

4.6.4 Per-stream Lifetime Distribution Analysis

To better understand the benefit of PCStream on the WAF reduction, we measured per-stream lifetime distributions for the Mixed 1 scenario. Fig. 15 shows a box plot of data lifetimes from the 25th to the 75th percentile. As shown in Fig. 15, streams in both PCStream and ManualStream are roughly categorized as two groups, $G1$ = $\{S_1, S_2, S_3, S_4, S_5\}$ and $G2$ = $\{S_6, S_7, S_8\}$, where $G1$ includes streams with short lifetimes and small variances (*i.e.*, S_1, S_2, S_3, S_4, and S_5) and $G2$ includes streams with large lifetimes and large variances (*i.e.*, S_6, S_7, and S_8). The S_0 does not belong to any groups as it is assigned to requests whose lifetimes are unknown. Even though the variance in the S_0 is wider than that in ManualStream, PCStream showed similar per-stream distributions as ManualStream. In par-

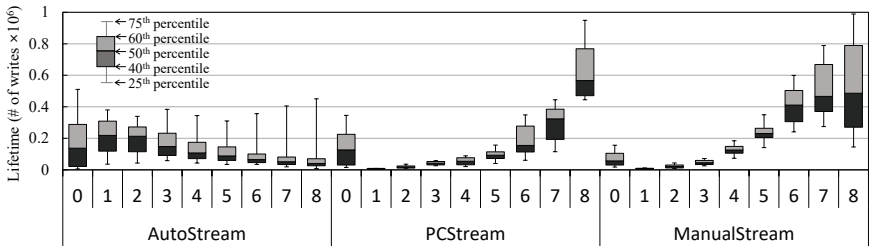


Figure 15: A Comparison of per-stream lifetime distributions.

ticular, for the streams in $G2$, PCStream exhibited smaller variance than ManualStream, which means that PCStream separates cold data from hot data more efficiently. Since PCStream moves long-lived data of a stream to its internal stream, the variance of streams with large lifetimes tend to be smaller over ManualStream.

AutoStream was not able to achieve small per-stream variances as shown in Fig. 15 over PCStream and ManualStream. As shown in Fig. 15, all the streams have large variances in AutoStream because hot data are often mixed with cold data in the same stream. Since the LBA-based data separation technique of AutoStream does not work well with both RocksDB and GCC, all the streams include hot data as well as cold data, thus resulting in large lifetime variances.

4.6.5 Impact of Internal Streams

In order to understand the impact of internal streams on different stream management techniques, we compared the two versions of each technique, one with internal streams and the other without internal streams. Since internal streams are used only for GC, they can be combined with any existing

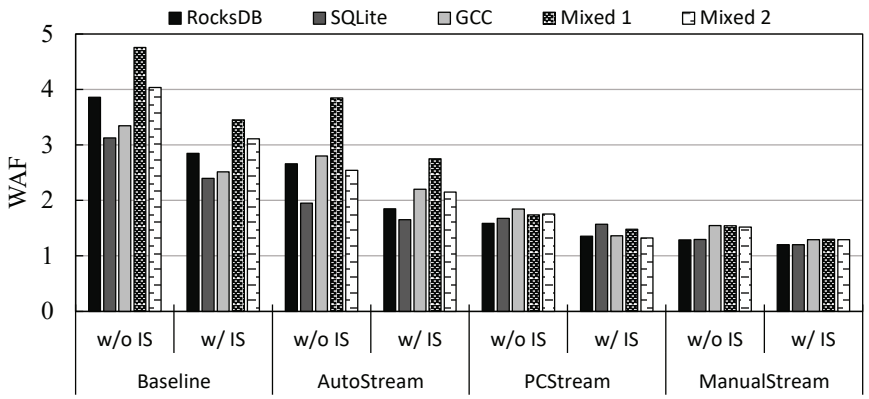


Figure 16: The effect of internal streams on WAF.

stream management techniques. Fig. 16 shows WAF values for five benchmarks with four techniques. Overall, internal streams worked efficiently across the four techniques evaluated. When combined with internal streams, Baseline, AutoStream, PCStream and ManualStream reduced the average WAF by 25%, 22%, 17%, and 12%, respectively. Since the quality of initial stream allocations in Baseline and AutoStream was relatively poor, their WAF improvement ratios with internal streams were higher over PCStream and ManualStream. Although internal streams were effective in separating short-lived data from long-lived data in both Baseline and AutoStream, the improvement from internal streams in these techniques are not sufficient to outperform PCStream and ManualStream. Poor initial stream allocations, which keep putting both hot and cold data to the same stream, unfortunately, offset a large portion of benefits from internal streams.

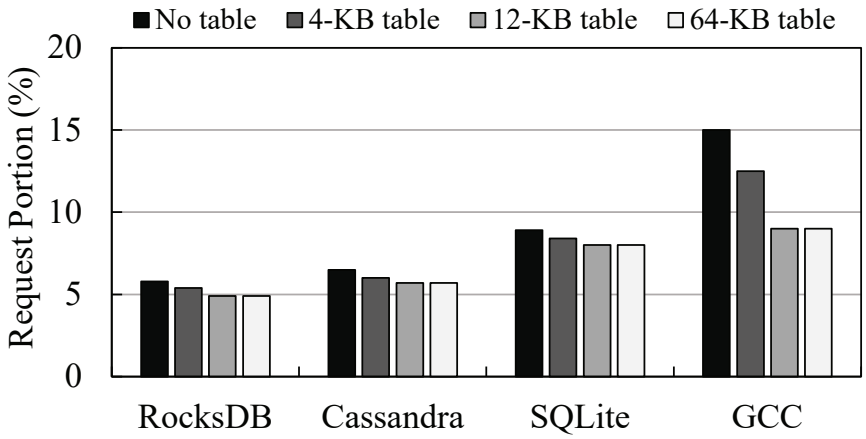


Figure 17: The effect of the PC attribute table.

4.6.6 Impact of the PC Attribute Table

As explained in Section 4.5.1, the PC attribute table is useful to maintain a long-term history of applications' I/O behavior by exploiting the uniqueness of a PC signature across different applications. To evaluate the effect of the PC attribute table on the efficiency of PCStream, we modified the implementation of the PC attribute table so that the PC attribute table can be selectively disabled on demands when a process terminates its execution. For example, in the kernel compilation scenario with GCC, the PC attribute table becomes empty after each kernel build is completed. That is, the next kernel build will start with no existing PC to stream mappings.

Fig. 17 show how many requests are assigned to the default S_0 stream over varying sizes of the PC attribute table. Since S_0 is used when no stream is assigned for an incoming write request, the higher the ratio of requests assigned to S_0, the less effective the PC attribute table. As shown in Fig. 17,

in RocksDB, Cassandra, and SQLite, the PC attribute table did not affect much the ratio of writes on S_0. This is because these programs run continuously for a long time while performing the same dominant activities repeatedly. Therefore, although the PC attribute table is not maintained, they can quickly reconstruct it. On the other hand, the PC attribute table was effective for GCC, which frequently creates and terminates multiple processes (*e.g.*, `cc1`). When no PC attribute table was used, about 16% of write requests were assigned to S_0. With the 4-KB PC attribute table, this ratio was reduced to 12%. With the 12-KB PC attribute table, only 9% of write requests were assigned to S_0. This reduction in the S_0 allocation ratio reduced the WAF value from 1.96 to 1.54.

Chapter 5

Fine-grained Deduplication Technique using Program Contexts

5.1 Overview

As the price-per-byte of NAND flash memory is rapidly decreasing, NAND flash-based solid-state drives (SSDs) are emerging as a viable high-performance storage solution for laptops, desktop PCs and high-performance enterprise systems. However, as NAND flash memory technology scales down to 20-nm and below [49], storing data reliably in NAND flash memory gets a key design challenge of NAND-based storage systems. Particulary, the reduction in the number of P/E cycles of NAND flash memory seriously limits the overall lifetime of flash-based SSDs, making it difficult for SSDs to be used in write-intensive applications.

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 [54, 55]. As a result, only non-duplicate data, i.e., unique data, are stored in SSDs, thus effectively decreasing the total amount of data written to SSDs. In most deduplication schemes proposed for SSDs, the unit of data deduplication is the same as the flash page

size which is usually 4 KB or 8 KB. However, this page-based deduplication technique misses many chances of eliminating duplicate data, especially when two pages are *almost* identical. 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 [50].

In this dissertation, 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.

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. Our evaluation results show that FineDedup prolongs the lifetime of SSDs by up to 24% over page-based deduplication

while requiring a negligible memory space increase. This improvement comes with a less than 5% read performance penalty over page-based deduplication.

5.2 Selective Deduplication using Program Contexts

As analyzed in Section 3.3, data duplicate patterns and I/O activities seems to have relationship. Particulary, some writes are known a priori to be likely to be unique. Applications might generate data that should not or cannot be deduplicated. For example, some applications write random, compressed, or encrypted data; others write complex formats (e.g., virtual disk images) with internal metadata that tends to be unique.

Attempting to deduplicate unique writes wastes CPU time on hash computation and I/O bandwidth on maintaining the hash index. Unique hashes also increase the index size, requiring more RAM space and bandwidth for lookup, insertion, and garbage collection.

In summary, if program context can help to know when it is unwise to deduplicate a write, we can optimize its performance and reliability. We implemented a selective deduplication based on the program context.

Since deduplication uses computational resources and may increase latency, it should only be performed when there is a potential benefit. The program context hint instructs the device not to deduplicate a particular writes. It has two use cases: (1) unique data: there is no point in wasting resources on deduplicating data that is unlikely to have duplicates, such as short-lived

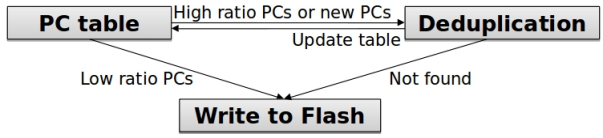


Figure 18: Diagram of selective deduplication.

data; (2) reliability: maintaining multiple copies of certain writes may be necessary, e.g., superblock replicas in many file systems.

Figure 18 shows a diagram of the proposed selective deduplication technique. When a write request comes, it first query its PC value to the PC table which holds duplicate rate per PCs. If the PC has high duplicate ratio or a new one, we apply deduplication on the request. After fingerprinting and searching, the result is updated to the PC table. If the PC has low duplicate ratio or there is no written data for a new PC, we handle data as a normal reqeust.

5.3 Exploiting Small Chunk Size

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

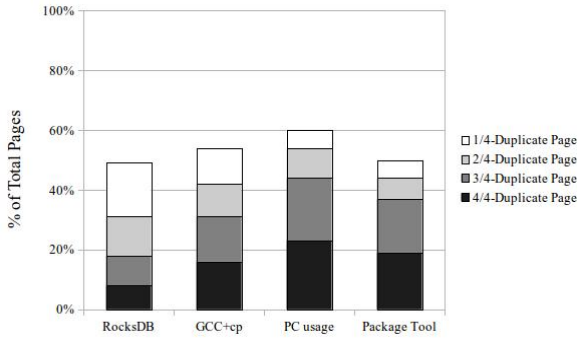


Figure 19: The percentage of pages according to their partial duplicate patterns.

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 four I/O traces, `RocksDB`, `GCC+cp`, `PC usage`, and `Package Tool` which are explained in Section 5.5. In our evaluation, the page size was 4 KB and the chunk size was set to 1 KB.

Fig. 19 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

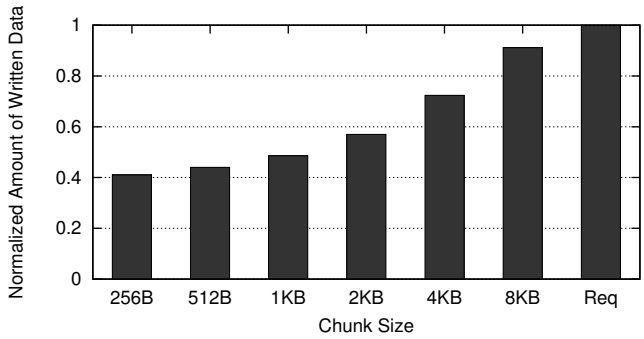


Figure 20: The amount of written data under varying chunk sizes in PC workload.

Fig. 19, 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. 19, 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 deduplication while varying the chunk size from 256 B to 8 KB. As shown in Fig. 20, 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 [49, 50],

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. 20, 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.

5.4 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 5.4.1. In Section 5.4.2 and 5.4.3 we introduce a read performance penalty and memory overheads caused by FineDedup, respectively, and explain how these problems can be resolved in FineDedup.

5.4.1 Overall Architecture of FineDedup

Fig. 21 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 [52], 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

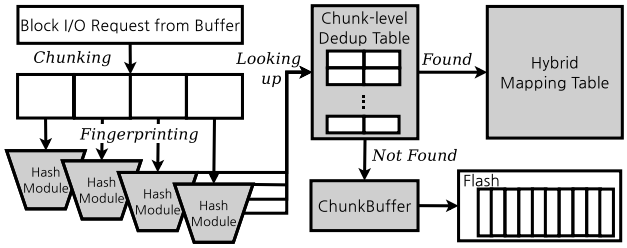


Figure 21: An overview of the proposed FineDedup technique.

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-level mapping table, FineDedup uses a hybrid mapping strategy, which is described in Section 5.4.3 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.

5.4.2 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. 22 illustrates why data fragmentation occurs in FineDedup. There are two page write requests, *Req 1* and *Req 2*, in Fig. 22. *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

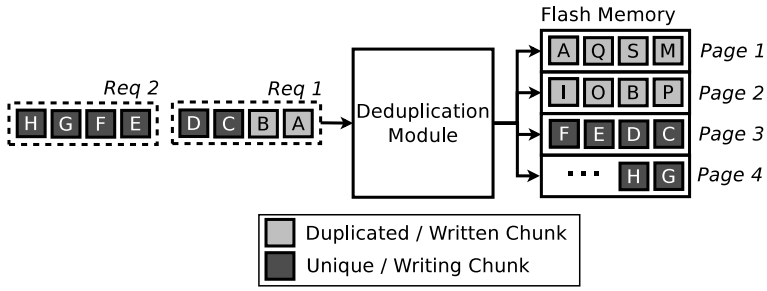


Figure 22: Data fragmentation caused by FineDedup.

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. 22, *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' and 'H' will be written to *page 4* with other data chunks, as shown in Fig. 22. 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

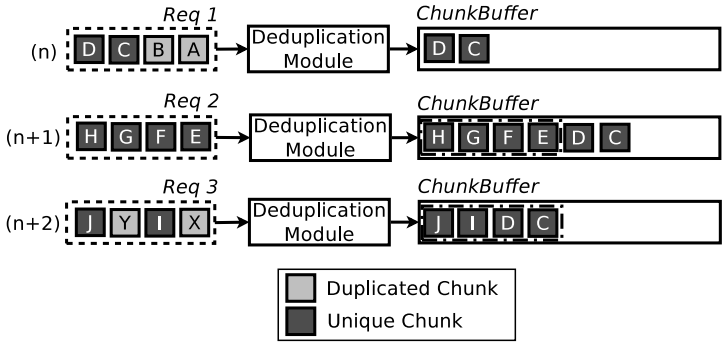


Figure 23: A packing scheme in the *chunk buffer*.

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 physical page together. Fig. 23 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. 23, 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. 19, 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 [53]. 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.

5.4.3 Memory Overhead Management

As mentioned in Section 5.4.1, 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. 19, 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. 24 shows the overall architecture of the hybrid mapping table used

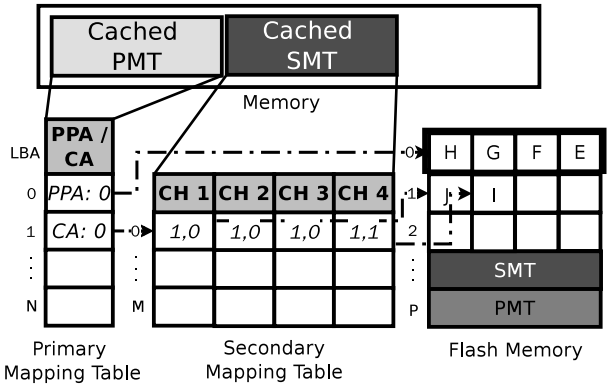


Figure 24: An overview of the demand-based hybrid mapping table.

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. 24) in flash memory or an index of the SMT (chunk address (CA) in Fig. 24).

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 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. 24, 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. 24 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 [60]. 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.

5.5 Experimental Results

In order to evaluate the effectiveness of FineDedup, we performed our experiments using a trace-driven simulator with the I/O traces collected under

Trace	Description	Amount of Writes	Amount of Reads
RocksDB	Benchmarking on the Key-value store	3.1 GB	810 MB
GCC+cp	Developing Kernel modules	2.6 GB	66 MB
PC usage	Web surfing, emailing and editing document, etc.	2.5 GB	70 MB
Package Tool	Installing & upgrading software packages	4.9 GB	119 MB

Table 2: A summary of traces used for experimental evaluations.

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.

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 2.

5.5.1 Effectiveness of FineDedup

Fig. 25 shows the amount of data written to flash memory by FineDedup over the existing scheme. The results shown in Fig. 25 are normalized to

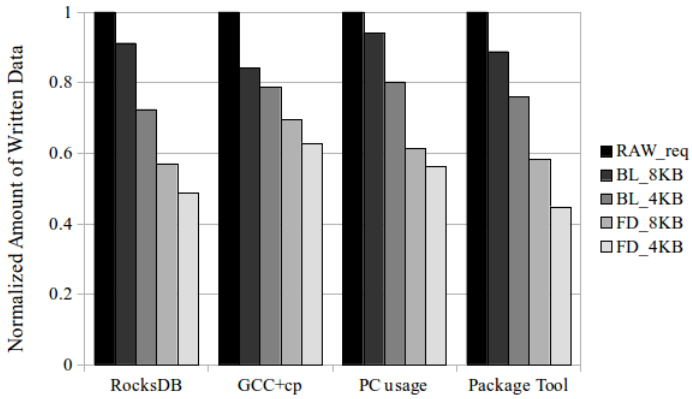


Figure 25: The amount of written data under various schemes.

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 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. 25, 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 `RocksDB`, FineDedup saves 37% flash writes over *BL_8K*. As expected,

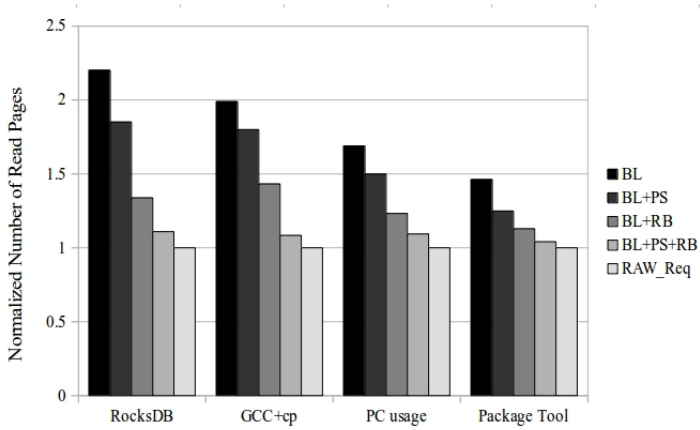


Figure 26: The number of page read operations.

the benefit of FineDedup mainly derives from the decreased chunk size because it increases the probability of finding and eliminating duplicate data. Especially, `RocksDB` trace shows a large number of write requests with little different data during compaction, so FineDedup can effectively identify unchanged data as duplicate while existing deduplication technique regards as unique data.

5.5.2 Read Overhead Evaluation

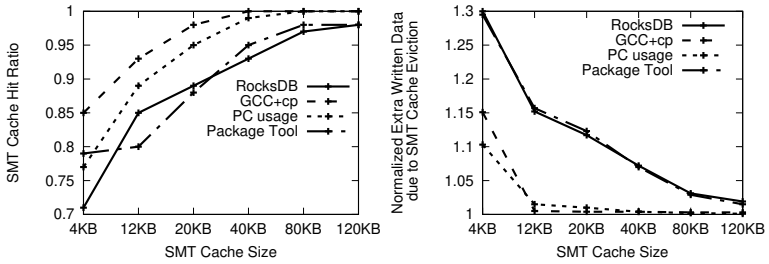
As explained in Section 5.4.2, fine-grained chunking in FineDedup increases the number of page read operations. Fig. 26 shows the normalized number of page read operations compared with the number of read requests in the workloads. *RAW_Req* indicates the number of original page read requests and *BL* 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. 26, 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.

5.5.3 Memory Overhead Evaluation

As explained in Section 5.4.3, 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. 27, 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 [60], 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.



(a) Hit ratio of SMT cache

(b) Extra data written due to SMT cache evictions

Figure 27: The effectiveness of the demand-based hybrid mapping table in FineDedup with various cache sizes.

Fig. 27(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 addition, Fig. 27(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. As a result, FineDedup does not incur a significant memory overhead even when the fine-grained chunking method is not effective.

Chapter 6

Conclusions

6.1 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 have presented a new stream management technique, PCStream, for multi-streamed SSDs. Unlike existing techniques, PCStream fully automates the process of mapping data to a stream based on PCs. Based on observations that most PCs are effective to distinguish lifetime characteristics of written data, PCStream allocates each PC to a different stream. When a PC has a large variance in their lifetimes, PCStream refines its stream allocation during GC and moves the long-lived data of the current stream to the corresponding internal stream. Our experimental results show that PCStream can improve the IOPS by up to 56% over the existing automatic technique while reducing WAF by up to 69%.

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.

6.2 Future Work

6.2.1 Supporting applications that have unusal program contexts

The current version of PCStream can be extended. PCStream does not support applications that rely on a write buffer (*e.g.,* MySQL). Similarly, we can make PCStream to support thread pools. In a thread pool, I/O is performed by worker threads. so it obviously won't work unless we somehow capture PC when an I/O is queued to the thread pool.

To address this, we plan to extend PCStream interfaces so that developers can easily incorporate PCStream into their write buffering modules with minimal efforts. The key insight on this extension is that we should

collect PC signatures *at the front-end interface* of an intermediate layer that accepts write requests from other parts of the program.

6.2.2 Optimizing read request based on the I/O context

Although PCStream focues on only writes, program contexts are originally used to predict accesses to the cache at the operating system layer. For example, PCs have been used to accurately predict the instruction behavior in the processor's pipeline which allows the hardware to apply power reduction techniques at the right time to minimize the impact on performance [9] . In Last Touch Predictor [10], PCs are used to predict which data will not be used by the processor again and free up the cache for storing or prefetching more relevant data. In PC-based prefetch predictors [11] , a set of memory addresses or patterns are linked to a particular PC and the next set of data is prefetched when that PC is encountered again.

Based on the good performance on cache management, we can optimize read request using the program context. For example, when there are some read pattern for a PC (sequential read or repetitive read), we can copy the target page to other chip or channel to maximize read bandwidth for the future read. Moreover, when there is a program context that read specific address many times, we can handle read disturbance problem in advance.

6.2.3 Exploiting context information to improve fingerprint lookups

One of the most time-consuming operations in a deduplication system is hash lookup, because it often requires extra I/O operations. Worse, hashes are randomly distributed by their very nature. Hence, looking up a hash often requires random I/O, which is the slowest operation in most storage systems. Also, as previous studies have shown [61], it is impractical to keep all the hashes in memory because the hash index is far too large.

When a deduplication system knows what data is about to be written, it can prefetch the corresponding hashes from the index, accelerating future data writes by reducing lookup delays. For example, a copying process first reads source data and then writes it back. If a deduplication system can identify that behavior at read time, it can prefetch the corresponding hash entries from the index to speed up the write path. Another interesting use case for this context is segment cleaning in log-structured file systems (e.g., Nilfs2) that migrate data between segments during garbage collection.

The I/O context is used to inform the deduplication system of I/O operations that are likely to generate further duplicates (e.g., during a file copy) so that their hashes can be prefetched and cached to minimize random accesses. This hint can be set on the read path for applications that expect to access the same data again.

Bibliography

[1] T. Chung, D. Park, S. Park, D. Lee, S. Lee and H. Song, "A Survey of Flash Translation Layer," *Journal of Systems Architecture,* vol. 55, no. 5, pp. 332-343, 2009.

[2] 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.

[3] 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.

[4] 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.

[5] Deepstorage.net, "Storage Efficiency Imperative: An In-Depth Review of Storage Efficiency Technologies and the Solidfire Approach," `http://www.deepstorage.net/NEW/reports/SolidFireStorageEfficiency.pdf`, 2012.

[6] D. Park, and D. Du, "Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters," In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[7] R. Stoica, J. Levandoski, and P. Larson, "Identifying Hot and Cold Data in Main-Memory Databases," In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2013.

[8] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "Warm: Improving Nand Flash Memory Lifetime with Write-Hotness Aware Retention Management," In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[9] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using Dynamic Cache Management Techniques to Reduce Energy in a High-Performance Processor," In *Proceedings of the ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[10] A. Lai, and B.Falsafi, "Selective, Accurate, and Timely Selfinvalidation Using Last-Touch Prediction," In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[11] T. Sherwood, S. Sair, and B. Calder, "Predictor-Directed Stream Buffers," In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[12] M. Chiang, P. Lee, R. and Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software-Practice & Experience*, vol. 29, no. 3, pp. 267-290, 1999.

[13] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-Based Solid State Drives," In *Proceedings*

of the ACM International Systems and Storage Conference (SYSTOR), 2009

[14] W. Bux, and I. Iliadis, "Performance of Greedy Garbage Collection in Flash-Based Solid-State Drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172-1186, 2010.

[15] C. Tsao, Y. Chang, and M. Yang, "Performance Enhancement of Garbage Collection for Flash Storage Devices: An Efficient Victim Block Selection Design", In *Proceedings of the Annual Design Automation Conference (DAC)*, 2013.

[16] S. Yan, H. Li, M. Hao, M. Tong, S. Sundararaman, A. Chien, and H. Gunawi, "Tiny-tail Flash: Near-perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs", In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[17] S. Hahn, S. Lee, and J. Kim, "To Collect or Not to Collect: Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes", In *Proceedings of the Design Automation Conference (DAC)*, 2015.

[18] J. Cui, Y. Zhang, J. Huang, W. Wu, and J. Yang, "ShadowGC: Cooperative Garbage Collection with Multi-Level Buffer for Performance Improvement in NAND flash-based SSDs", In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018.

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

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

[21] 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 International Conference on Scalable Computing and Communications (ScalCom)*, 2015.

[22] H. Yong, K. Jeong, J. Lee, J. Kim, "vStream: Virtual Stream Management for Multi-streamed SSDs", In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[23] 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 USENIX Conference on File and Storage Technologies (FAST)*, 2018.

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

[25] Facebook, https://github.com/facebook/rocksdb.

[26] Apache Cassandra, http://cassandra.apache.org.

[27] C. Gniady, A. Butt, and Y. Hu, "Program-Counter-based Pattern Classification in Buffer Caching", In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

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

[29] 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)*, 2011.

[30] J. Hartigan, and M. Wong, "Algorithm as 136: A k-means Clustering Algorithm", *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100-108, 1979.

[31] 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.

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

[33] NVM Express Revision 1.3, `http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf`.

[34] S. Frank, “Tightly Coupled Multiprocessor System Speeds Memory-Access Times”, *Electronics*, vol. 57, no. 1, 1984.

[35] SQLite, `https://www.sqlite.org/index.html`.

[36] 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`.

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

[38] Samsung, Samsung SSD PM963, `https://www.compuram.de/documents/datasheet/Samsung_PM963-1.pdf`

[39] S. Liang, Java Native Interface: Programmer’s Guide and Specification, 1999.

[40] MySQL, `https://www.mysql.com`.

[41] PostgreSQL, `https://www.postgresql.org`.

[42] OpenJDK, `http://openjdk.java.net/`.

[43] NVM-Express User Space Tooling For Linux, `https://github.com/linux-nvme/nvme-cli`.

[44] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB”, In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[45] The Transaction Processing Performance Council, Benchmark C, http://www.tpc.org/tpcc/default.asp.

[46] T. Kim, S. Hahn, S. Lee, J. Hwang, J. Lee and J. Kim, "PCStream: Automatic Stream Allocation Using Program Contexts", In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[47] MonetDB, https://www.monetdb.org.

[48] Linux Programmer's Manual, mmap(2) - map files or devices into memory, http://man7.org/linux/man-pages/man2/mmap.2.html.

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

[50] 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.

[51] 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.

[52] 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.

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

[54] 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 USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[55] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[56] Z. Chen and K. Shen, "OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[57] W. Li, G. Jean-Baptise, J. Riveros, and G. Narasimhan, "CacheDedup: In-line Deduplication for Flash Caching," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[58] D. Meister and A. Brinkmann, "dedupv1: Improving Deduplication Throughput using Solid State Drives," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[59] W. Dong et al., "Tradeoffs in Scalable Data Routing for Deduplication Clusters," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011

[60] A. Gupta, Y. Kim and B. Urgaaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[61] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the Data Domain deduplication file system," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2008.