
# Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts

Taejin Kim and Duwon Hong, *Seoul National University;* Sangwook Shane Hahn, *Western Digital;* Myoungjun Chun, *Seoul National University;* Sungjin Lee, *DGIST;* Jooyoung Hwang and Jongyoul Lee, *Samsung Electronics;* Jihong Kim, *Seoul National University*

https://www.usenix.org/conference/fast19/presentation/kim-taejin

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

# Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts

Taejin Kim, Duwon Hong, Sangwook Shane Hahn[†], Myoungjun Chun,
Sungjin Lee[‡], Jooyoung Hwang[*], Jongyoul Lee[*], and Jihong Kim

*Seoul National University, [†]Western Digital, [‡]DGIST, [*]Samsung Electronics*

## Abstract

Multi-streamed SSDs can significantly improve both the performance and lifetime of flash-based SSDs when their streams are properly managed. However, existing stream management solutions do not adequately support the multi-streamed SSDs for their wide adoption. No existing stream management technique works in a fully automatic fashion for general I/O workloads. Furthermore, the limited number of available streams makes it difficult to effectively manage streams when a large number of streams are required. In this paper, 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 with no manual work. In order to overcome the limited number of supported streams, we propose a new type of streams, internal streams, which can be implemented at low cost. PCStream can effectively double the number of available streams using internal streams. Our evaluations on real multi-streamed SSDs show that PCStream achieves the same efficiency as highly-optimized manual allocations by experienced programmers. PCStream improves IOPS by up to 56% over the existing automatic technique by reducing the garbage collection overhead by up to 69%.

## 1 Introduction

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 [1, 2, 3, 4, 5, 6]. 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 [7, 8].

Multi-streamed SSDs provide a special interface mechanism for a host system, called streams[1]. With the stream interface, data separation decisions *on the host level* can be delivered to SSDs [9, 10]. When the host system assigns two data $D_1$ and $D_2$ to different streams $S_1$ and $S_2$, respectively, a multi-streamed 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 [10, 11, 12, 13, 14]. 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

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

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 [10, 11, 12, 13] require programmers to assign streams at the application level with manual code modifications. AutoStream [14] 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 [15] or Cassandra [16]) and write-once workloads (such as a Linux kernel build). Unlike conventional in-place update workloads where data are 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 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) [17, 18], which uniquely represents an execution path of a program up to a write system call, is known to be effective in representing dominant I/O activities [19]. 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 [20]. 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 number 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 write amplification factor (WAF) by 49%.

The rest of this paper is organized as follows. In Section 2, we review existing stream management techniques. Before describing PCStream, its two core components are presented in Sections 3 and 4. Section 5 describes PCStream in detail. Experimental results follow in Section 6, and related work is summarized in Section 7. Finally, we conclude with a summary and future work in Section 8.

## 2 Limitations of Current Practice in Multi-Streamed SSDs

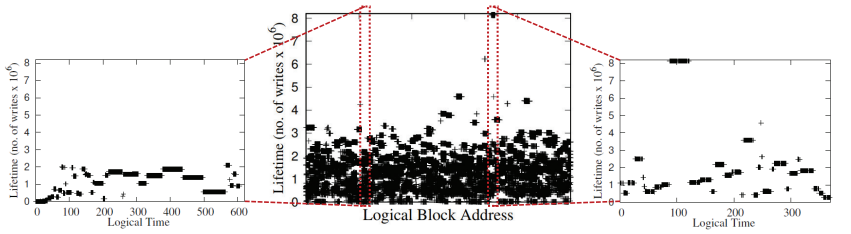
In this section, we review the key weaknesses of existing stream management techniques as well as stream implementation methods. PCStream was motivated to overcome these weaknesses so that multi-streamed SSDs can be widely employed in practice.

### 2.1 No Automatic Stream Management for General I/O Workloads

Most existing stream management techniques [10, 11, 12] require programmers to manually allocate streams for their applications. For example, in both ManualStream[3] [10] and [11], there is no systematic guideline on how to allocate

[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.

[3] For brevity, we denote the manual stream allocation method used in [10] by ManualStream.




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

Fig. 1: Lifetime distributions of append-only workload over addresses and times.

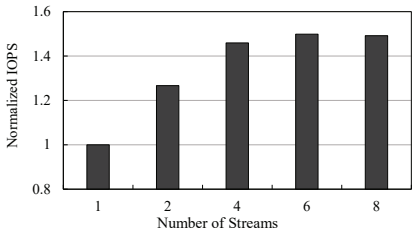



Fig. 2: IOPS changes over the number of streams.

streams for a given application. The efficiency of stream allocations largely depends on the programmer's understanding and expertise 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 [12] 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 [13] and AutoStream [14] may be considered as automatic stream management techniques, their applicability is quite limited. FStream [13] can be useful for separating file system metadata but it does not work for the user data separation. AutoStream [14] 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 [15] and Cassandra [16], the 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 [15], which is a popular key-value store based on the LSM-tree algorithm [21]. Fig. 1(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 command [22]. As shown in Fig. 1(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. 1(b) and 1(c) show scatter plots of data lifetimes over the logical time for two specific 1-MB chunks with 256 pages. As shown in Figs. 1(b) and 1(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. 1(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 does not depend on lower-level details such as write patterns based on LBAs.

## 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. 2 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. 2, 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 [9, 23]. However, the number of streams supported in commercial SSDs is quite limited, say, 4 to 16 [10, 11, 14], 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

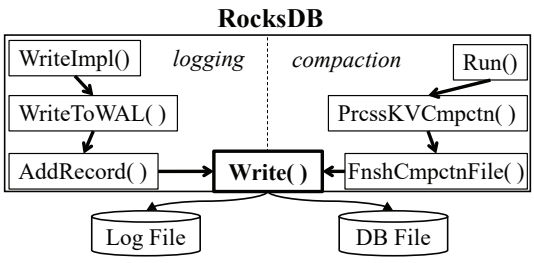



Fig. 3: An illustration of (simplified) execution paths of two dominant I/O activities in RocksDB.

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 the 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 [24] 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 the 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. The limited size of fast memory, unfortunately, restricts the number of supported streams quite severely.

# 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.

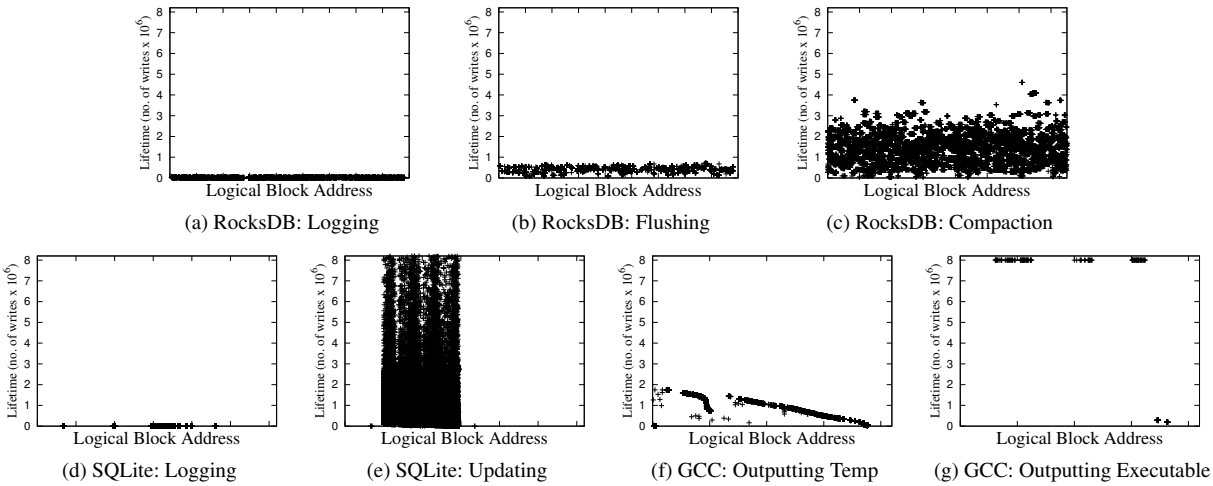
In this paper, we argue that a program context can be used to build an efficient general-purpose classifier of dominant I/O activities with different data lifetimes. Here, a PC represents an execution path of an application which invokes write-related system call functions such as `write()` and `writev()`. There could be various ways of extracting PCs, but the most common approach [17, 18] 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.

## 3.1 PC as a Unit of Lifetime Classification

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 [15], SQLite [25], and GCC [26]. 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).

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. 3 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.

To confirm our hypothesis that data lifetimes can be distinguished by tracking dominant I/O activities and a PC is




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

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

Fig. 4: Data lifetime distributions of dominant I/O activities in RocksDB, SQLite and GCC.

a useful unit of classification for different I/O activities, we have analyzed how well PCs work for RocksDB, SQLite and GCC. Fig. 4 shows data lifetime distributions of dominant I/O activities which were distinguished by computed PC values. As expected, Fig. 4 validates that dominant I/O activities show distinct data lifetime distributions over the logical address space. For example, as shown in Figs. 4(a)∼4(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. 4(d)∼4(g). As shown in Fig. 4(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. 4(f), because of the write-once pattern of temporary files. But, unlike the other graphs in Fig. 4, data lifetime distributions of Figs. 4(c) and 4(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.

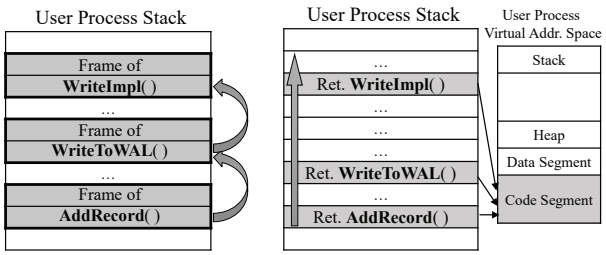
Note that if we used an LBA-based data separator instead of the proposed PC-based scheme, most of data lifetime characteristics shown in Fig. 4 could not have been known. Only the data lifetime distribution of the logging activity of SQLite, as shown in Fig. 4(d), can be accurately captured by the LBA-based data separator. For example, the LBA-based data separator 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.

## 3.2 Extracting PCs

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. 3, 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 [26]. 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 check 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 take too long, we stop the scanning step once a sufficient number of return address candidates are found. The larger the return address candidates, the longer the com-




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

Fig. 5: Examples of PC extraction methods.

putation time. On the other hand, if the number of return addresses is too small, two different paths can be regarded as the same path. When the minimum number of addresses that can distinguish the two paths of the program is found, the scanning should be stopped to minimize the scanning overhead. In our evaluation, five return addresses were enough to distinguish execution paths.

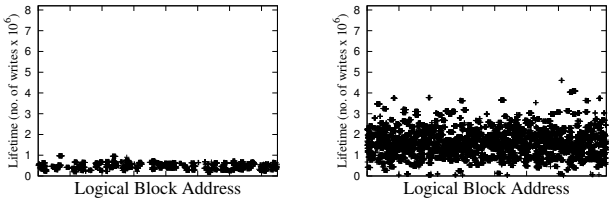
Even though it is quite ad-hoc, this restricted scan is quite effective in distinguishing different PCs because it is very unlikely that two different PCs reach the same `write()` system call through the same execution subpath that covers five proceeding function calls. In our evaluation on a PC with 3.4 GHz Intel CPU, the overhead of the restricted scan was almost negligible, taking only 300∼400 *n*sec per `write()` system call.

## 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.

### 4.1 PCs with Large Lifetime Variances

For most PCs, their lifetime distributions tend to have small variances (*e.g.*, Figs. 4(a), 4(d), and 4(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.




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

Fig. 6: Lifetime distributions of the compaction activity at different levels.

RocksDB maintains several levels, L1, ..., L*n*, 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 [21], a higher level is smaller than a lower level (*i.e.*, the size of (L2) $<$ the size of (L3)). Thus, data stored in a higher level is invalidated more frequently than those kept in lower levels, thereby having shorter lifetimes.

Unfortunately, in the current RocksDB implementation, the compaction step is supported by the same execution path (*i.e.*, the same PC) regardless of the level. Therefore, the PC for the compaction activity cannot effectively separate data with short lifetimes from one with long lifetimes. Fig. 6(a) and 6(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. 4(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. 4(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.

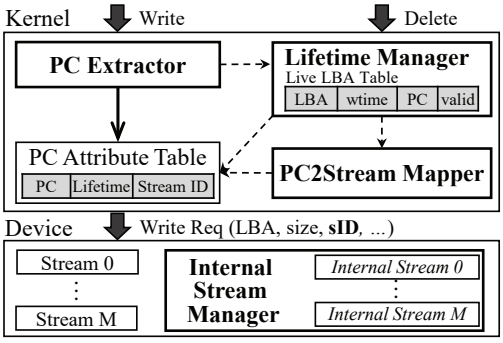



Fig. 7: An overall architecture of PCStream.

## 4.2 Implementation of Internal Streams

As described in Section 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 and most GC can be handled as background tasks, internal streams have 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.

# 5 Design and Implementation of PCStream

In this section, we explain the detailed implementation of PCStream. Fig. 7 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 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 firmware, is responsible for handling internal streams associated with external streams.

## 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 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 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. For example, for a 256 GB SSD, 4 KB-granularity

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

requires 4 billion entries while 1 MB-granularity requires 16 million entries. For a 9 byte-sized entry, LBA table requires about 144 MB memory. Due to the coarse-grained mapping, multiple requests within an address unit are considered as requests to the same address, which are updates. Therefore, the data lifetime can be recognized shorter than the real lifetime. However, even if long-lived data are misallocated to the short lifetime stream, the internal stream effectively suppresses the increase in WAF.

**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 maximum number of unique PCs was up to 30. So the DRAM size of the PC attribute table was sufficient with 360 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 the 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. In our experiment, there was no alias for the PC value. 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 risk of PC collisions.

## 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 [20] 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 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 the 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.

## 5.3 Internal Stream Management

As explained in Section 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 the 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

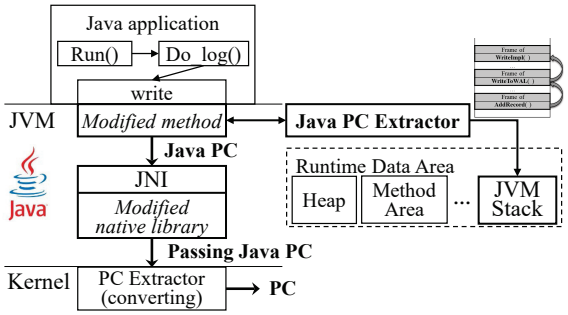



Fig. 8: Extracting PCs for JVM.

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 [27]. 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.

## 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) [28] 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. 7 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 example is a program that maintains a write buffer that is dedicated to dealing with all the writes from an application. For example, in MySQL [29] and PostgreSQL [30], 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 the 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. 8 illustrates how PCStream collects PC signatures from Java programs. We have modified the OpenJDK [31] source to extract PC signatures for most of the 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. For the JIT compilation, the codes are dynamically compiled and optimized, so the computed PC value of the same path can be different. However, if the code cache space is sufficient, the compiled code is reused, so there is no problem in using the PC. In the experiment, there was enough space in the code cache.

Unlike Java, there is no 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. Moreover, in the virtual machine (VM) environment, modification of the VM itself is inevitable. PCStream can get PC of guest OS, but it is difficult to transfer directly to the device. We can transfer PC information to the system call layer of host OS through modification of virtualization layer.

# 6 Experimental Results

## 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 SSD. 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 detailed performance analysis, we built a modified `nvme-cli` [32] 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 [10], and AutoStream [14]. 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-based stream management technique proposed in [14].

We have carried out experiments with various benchmark programs which represent distinct write characteristics. RocksDB [15] and Cassandra [16] have append-only write patterns. SQLite [25] has in-place update write patterns and GCC [26] has write-once patterns. For more realistic evalu-

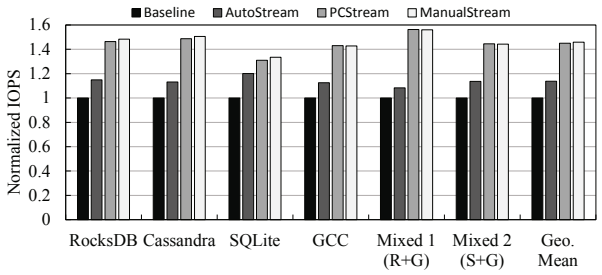



Fig. 9: A comparison of normalized IOPS.

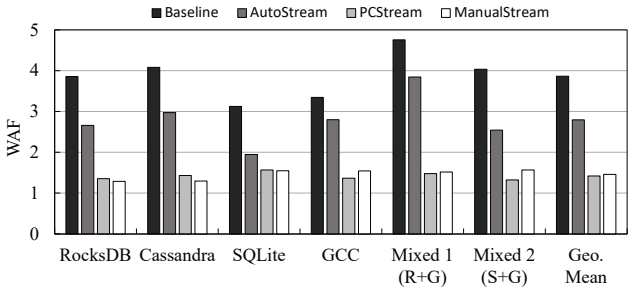



Fig. 10: A comparison of WAF under different schemes.

ations, we also used mixed workloads running two different benchmark programs simultaneously.

In both RocksDB and Cassandra experiments, Yahoo! Cloud Serving Benchmark (YCSB) [33] with 12-million keys was used to generate update-heavy workloads (workload type A) which consist of 50/50 reads and writes. Since both RocksDB and Cassandra are based on the append-only LSM-tree algorithm [21], 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 5.4. In SQLite evaluations, TPC-C [34] 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 in our experiments, 90% of the total SSD capacity was initially filled up with user files before benchmarks run.

## 6.2 Performance Evaluation

We compared the IOPS values of three existing techniques with PCStream. Fig. 9 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 number 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%.

[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.

## 6.3 WAF Comparison

Fig. 10 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 observation in Fig. 10 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, PCStream continuously adapts its stream allocations during run time, thus quickly responding to varying execution environments. For example, 10 PCs out of 25 PCs are remapped by 7 reclustering operations for Mixed 1 workload.

## 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. 11 shows a box plot of data lifetimes from the 25th to the 75th percentile. As shown in

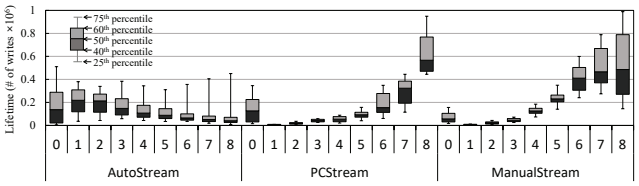



Fig. 11: A Comparison of per-stream lifetime distributions.

Fig. 11, 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 particular, 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. 11 over PCStream and ManualStream. As shown in Fig. 11, 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.

## 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 stream management techniques. Fig. 12 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 is 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.

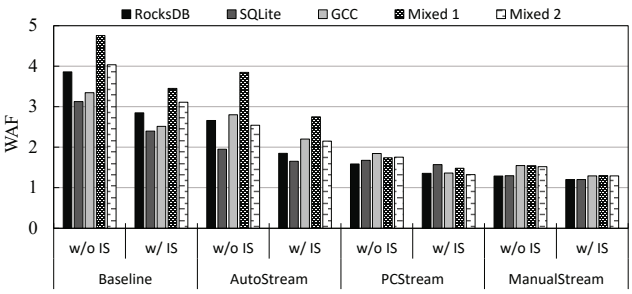



Fig. 12: The effect of internal streams on WAF.

## 6.6 Impact of the PC Attribute Table

As explained in Section 5, 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. 13 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. 13, 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 re-

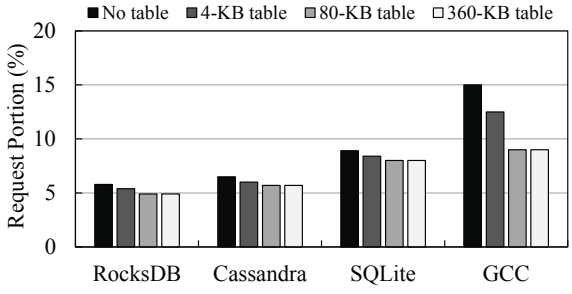



Fig. 13: The effect of the PC attribute table.

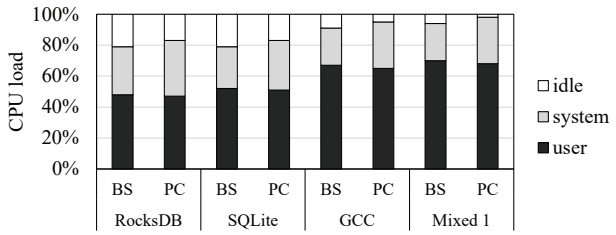



Fig. 14: A comparison of cpu load.

duced to 12%. With the 360-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.

### 6.7 CPU Overhead Evaluation

As described in Sections 3 and 5, PCStream requires additional CPU usage to compute and clustering PCs. We used the sar command in linux to evaluate the additional CPU load on PCStream. Fig. 14 shows the CPU utilization of the baseline (BS) and PCStream (PC) technique. The percentage of CPU utilization that occurred while executing at the user level and kernel level was represented by *user* and *system*, respectively. *idle* indicates the percentage of time that the CPU was idle. For all cases, the increased CPU load due to PCStream was less than 5%.

## 7 Related Work

There have been many studies for multi-streamed SSDs [10, 11, 12, 13, 14, 35]. Kang *et al.* first proposed a multi-streamed SSD that supported manual stream allocation for separating different types of data [10]. Yang *et al.* showed that a multi-streamed SSD was effective for separating data of append-only applications like RocksDB [11]. 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.

Yang *et al.* presented an automatic stream management technique at the block device layer [14]. 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 [19]. Kim *et al.* extended it for multi-streamed SSDs [35]. Unlike these works, 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 run frequently.

## 8 Conclusions

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%.

The current version of PCStream can be extended in several directions. First, PCStream does not support applications that rely on a write buffer (*e.g.,* MySQL). To address this, we plan to extend PCStream interfaces so that developers can easily incorporate PCStream into their write buffering modules with minimal efforts. Second, we have only considered write-related systems calls to collect PCs, but many applications (*e.g.,* MonetDB [36]) heavily access files with mmap-related functions (*e.g.*, `mmap()` [37] and `msync()`). We plan to extend PCStream to work with mmap-intensive applications.

## Acknowledgments

We thank Youjip Won, our shepherd, and the anonymous reviewers for their valuable feedback and comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R1A2B6006878). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2018R1A5A1060031, NRF-2017R1E1A1A01077410) (Corresponding Author: Jihong Kim)

## References


[1] 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.

[2] 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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

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

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

[29] MySQL, https://www.mysql.com.

[30] PostgreSQL, https://www.postgresql.org.

[31] OpenJDK, http://openjdk.java.net/.

[32] NVM-Express user space tooling for Linux, https://github.com/linux-nvme/nvme-cli.

[33] 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.

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

[35] 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.

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

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