# PCStream: Automatic Stream Allocation Using Program Contexts

## Abstract

We propose a fully automatic stream management technique, called PCStream, for multi-streamed SSDs. PCStream is based on our observation that data lifetimes can be reliably predicted using write program contexts. By extracting program contexts during runtime, PCStream automates the data-to-stream mapping. When data mapped to the same stream show large differences in their lifetimes, PCStream moves the long-lived data of the current stream to its substream during garbage collection. Our experimental results show that PCStream can reduce the garbage collection overhead as much as a highly-optimized manual stream management technique while no code modification is necessary.

## 1 Introduction

Multi-streamed SSDs provide a special mechanism, called streams, for a host system to prevent data with different lifetimes from being mixed into the same block [1, 2]. When the host system maps two data $D_1$ and $D_2$ to different streams $S_1$ and $S_2$, a multi-streamed SSD guarantees that $D_1$ and $D_2$ are placed in different blocks. Since streams, when properly managed, can be very effective in minimizing the copy cost of garbage collection (GC), they can significantly improve both the performance and lifetime of flash-based SSDs [2, 3, 4, 5].

In order to achieve high performance on multi-streamed SSDs, data with similar *future* update times [6] should be allocated to the same stream, so that the copy cost of GC can be minimized. However, since it is difficult to know the future update times *a priori* when they are written, stream allocation decisions are often *manually* made by programmers based on their expertise on the application [2, 3] or the file system [4]. Furthermore, these manual techniques assume that the number of streams in an SSD is not changing, thus requiring manual modifications whenever the number of streams in the SSD changes. In this paper, our goal is to develop a *fully automatic* technique for managing streams which is applicable for *any* multi-streamed SSD.

To the best of our knowledge, AutoStream [5] is the only automatic stream management technique without additional manual work. However, since AutoStream predicts data lifetimes using the update frequency of the logical block address (LBA), it does not work well with modern append-only workloads such as RocksDB [7] or Cassandra [8]. Unlike conventional update workloads where data written to the same LBAs often show strong update locality, append-only workloads make it impossible to predict data lifetimes from LBA characteristics (such as access frequency or access patterns). For example, as shown in Fig. 1(b), data written to a fixed LBA range over time in RocksDB show widely varying data lifetimes, thus making it difficult to allocate streams based on LBA characteristics.

In this paper, we propose a fully automatic stream management technique, called PCStream, for multi-streamed SSDs based on program contexts (PCs). Since the key motivation behind PCStream was that data lifetimes should be estimated at a higher abstraction level than LBAs, PCStream employed a write program context[1] as a stream management unit. A program context [9, 10], which represents a particular execution phase of a program, is known to be an effective hint in separating data with different lifetimes [6]. PCStream automatically maps an identified program context to a stream. Since program contexts can be computed during runtime, PCStream does not need any manual work. In order to handle append-only workloads, PCStream extended the definition of the data lifetime so that the effect of the TRIM command [13] can be accounted for.

Although most program contexts show that their data lifetimes are distributed with small variances, we observed a few outliers whose data lifetimes have rather large variances. In PCStream, when such a PC *pID* is observed (which was mapped to a stream *sID*), the long-lived data of *pID* are moved to the substream of *sID* during GC. The substream

[1] Since we are interested in write-related system call such as write() in the Linux kernel, we call the related program context as *write program contexts* or simply *program contexts* in this paper.

prevents the long-lived data of the stream *sID* from being mixed with future short-lived data of the stream *sID*.

In order to evaluate the effectiveness of PCStream, we have implemented PCStream in the Linux kernel (ver. 4.5) and measured write amplification factor (WAF) values using RocksDB on a Samsung PM963 SSD and an SSD emulator. Our experimental results show that PCStream can reduce the GC overhead as much as a highly-optimized manual stream management technique while requiring no code modification. Furthermore, PCStream outperformed AutoStream by reducing the average WAF by 35%.

The rest of this paper is organized as follows. We explain the key motivations behind PCStream in Section 2. Section 3 describes the design of PCStream. The experimental results are shown in Section 4. Finally, we conclude in Section 5 with a summary and future work.

## 2 Basic Idea

### 2.1 I/O activity-based allocation

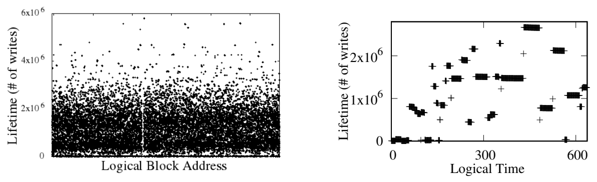
#### 2.1.1 Fallacy: LBA-based lifetime prediction

+ in-place update workload will be added

Many existing data separation techniques such as [5, 11] estimate the data lifetime based on the update frequency of LBAs. For example, AutoStream [5] assumes that, if some LBAs are frequently rewritten by applications, those LBAs hold hot data. This LBA-based lifetime prediction approach, however, does not work well with recent data-intensive applications where a majority of new data are written in an append-only manner.

In order to illustrate a mismatch between an LBA-based predictor and append-only workloads, we analyzed the write pattern of RocksDB [7], which is a popular key-value store based on the LSM-tree algorithm [12]. Fig. 1(a) shows how LBAs may be related to data lifetimes[2] in RocksDB [7]. As shown in Fig. 1(a), there exists no strong correlation between the LBAs and lifetimes in RocksDB. This scatter plot is in sharp contrast with one for update workloads where a few distinct LBA regions have short lifetimes while others have very long lifetimes.

We also analyzed if the lifetimes of LBAs change under some predictable patterns over time although the overall lifetime distribution shows large variances. Fig. 1(b) shows a scatter plot of data lifetimes over the logical time for a specific 1-MB chunk with 256 pages. As shown in Fig. 1(b), for the given chuck, data lifetimes vary in a random fashion (although some temporal locality is observed). Our illustration using RocksDB strongly suggests that under append-



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

Fig. 1: Lifetime distributions over addresses and times.

only workloads, LBAs are not useful in deciding data lifetimes.

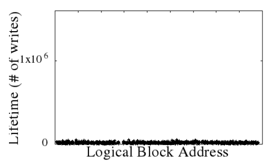
#### 2.1.2 Program context as a lifetime predictor

In developing PCStream, 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 lifetime patterns. In order to distinguish data by their lifetimes, therefore, it is important to effectively distinguish dominant I/O activities from each other. For example, in update workloads, LBAs alone were effective in separating dominant I/O activities.

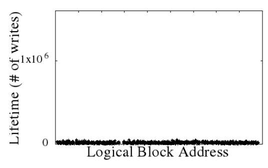
In this paper, we argue that a program context is an efficient general-purpose indicator for separating dominant I/O activities regardless of the type of I/O workloads. Since a PC represents an execution path of an application which invokes write-related system functions such as `write()` and `writev()` in the Linux kernel, we represent the PC by summing program counter values of all the functions along the execution path which leads to a write system call. In RocksDB, for example, dominant I/O activities include logging, flushing and compaction. Since they are invoked through different function-call paths, we can easily identify dominant I/O activities of RocksDB using PCs. For example, Fig. 5(a) shows an execution path for flushing in RocksDB. The sum of program counter values of `Run()`, `WriteLevel0Table()`, and `BuildTable()` is used to represent the PC for the flushing activity. Note that using the program context to distinguish data lifetimes is not new. For example, Ha *et al.* proposed a data separation technique based on the program context [6]. However, their work was neither designed for append-only workloads nor for modern multi-streamed SSDs.

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

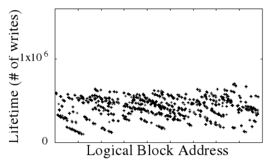
[2]The lifetime of data is defined by the logical time which is the number of writes to the device between when the data is first written and when the data is invalidated by an overwrite or a TRIM command.



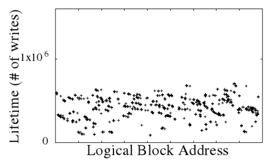
(a) Logging (PC)



(b) Logging (manual)

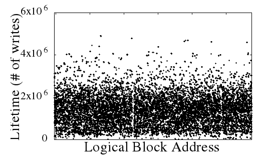


(c) Flushing (PC)

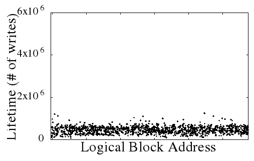


(d) Flushing (manual)

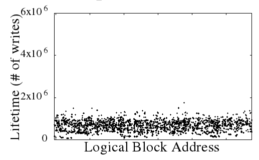
Fig. 2: Data lifetime distributions of different PCs.



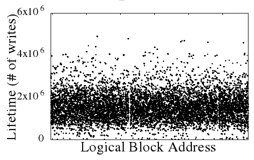
(a) compaction: all levels



(b) compaction: L2



(c) compaction: L3



(d) compaction: L4

Fig. 3: Lifetime distributions of the compaction activity at different levels.

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

## 2.2 Two-phase stream management

### 2.2.1 Limited number of streams

+ resource requirement explanation (memory, power)

### 2.2.2 I/O activity with large lifetime variance

For most PCs, their lifetime distributions tend to have small variances. However, we observed that a few outlier PCs which have large lifetime variations. For example, when multiple I/O contexts are covered by the same write system function, the corresponding PC may represent several I/O contexts whose data lifetimes are quite different. Such a case occurs, for example, in the compaction module of RocksDB. 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 [12], 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. 3(a) shows the lifetime distribution collected from the compaction-activity PC. Since this distribution includes lifetimes of data written from all the levels, its variance is large. When we manually separate the single compaction step into several per-level compaction steps, as shown in Figs. 5(b) and 5(c), the lifetime distributions of per-level compaction steps show smaller variances. In particular, L2 and L3 show distinct lifetime distributions from that of L4. Data from L2 and L3 are likely to have shorter lifetimes, while L4 has generally long-lived data as shown in Fig. 5(d).

### 2.2.3 Separating long-lived data during GC

Since it is difficult to separate data with different lifetimes within the same PC (as in the compaction-activity PC), we devised a two-phase method that decides SSD streams in two levels: the main stream ID in a host level and the substream ID in an SSD level. Conceptually, long-lived data in the main stream are moved to its substream to separate from (future) short-lived data of the main stream. Although moving data to the substream may increase WAF, the overhead can be hidden if we restrict the substream move during GC only. Since long-lived data (i.e., valid pages) in a victim block are moved to a free block during GC, they can be moved to the substream by changing the target block. For instance, PCStream assigns the compaction-activity PC *pID* to a main stream *sID* for the first phase. To separate the long-lived data of *pID* (e.g., L4 data) from future short-lived data of *pID* (e.g., L1 data), valid pages of the *sID* are assigned to its substream for the second phase during GC.

## 3 Design of PCStream

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

Fig. 4 shows an overall organization of PCStream. *The PC extractor module*, which is implemented in the Linux kernel as part of a system call handler, computes a PC signature, which is used as a unique ID for each program context. We use the signature program counter [9] as a PC signature by summing program counter values along the execution path to a write-related system function (e.g., `write()`). With the PC signature, we can monitor the data lifetime of each write at the program context level. A PC signature value is stored in an inode data structure of a file system (modified for PCStream) and is delivered to *the lifetime analyzer module* which estimates expected lifetimes of data belonging to a given PC in the block device level. In order to efficiently detect the end of data lifetime in append-only workloads, the lifetime analyzer also intercepts TRIM [13] requests from a file system. Based on the lifetime information, *the PC-to-stream mapper module* clusters PCs with similar lifetimes and maps them together to the same stream ID. This mapping is required because the number of streams in an SSD is generally less than the number of PCs in host applications.

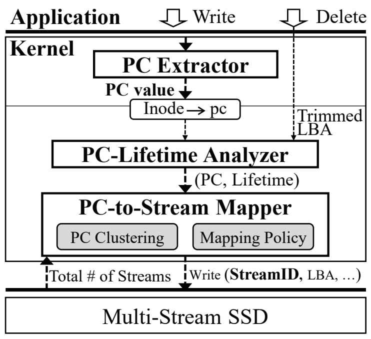


Fig. 4: An overall architecture of PCStream.

### 3.1 External Stream Management

#### 3.1.1 Automatic PC computation

As mentioned earlier, a PC is represented by a PC signature which is defined as the sum of program counter values along the execution path of a function call that finally reaches a write-related system function. A function call involves pushing the next program counter, which is used as a return address, to the stack followed by pushing a frame pointer value. In general, by using frame pointer values, we are able to back-track the stack frames of the process and selectively get return addresses for generating a PC signature. For example, Fig. 5(a) shows the abstracted execution path for flushing data in RocksDB and Fig. 5(b) illustrates how a PC signature is obtained by back-tracking the stack. Since a frame pointer value in the stack holds the address of the previous frame pointer, the PC extractor can easily obtain return addresses and accumulate them to compute a PC signature. (The return addresses are pushed before calling the write(), BuildTable() and WriteLevel0Table() functions.)

The frame pointer-based approach for computing PC signatures, however, is not always possible because modern C/C++ compilers often do not use the frame pointer for improving the efficiency of register allocation. One example is a `-fomit-frame-pointer` option of GCC [14]. Although this option allows the frame pointer to be used as a general-purpose register for high performance, it makes very difficult for us to back-track return addresses along the call chains.

In PCStream, we employ a simple but effective workaround for backtracking the call stack when the frame pointer is not used. When a write system call is made, we scan every word in the stack and check if it belongs to the process's code segment. If the scanned stack word holds a value within the address range of the code segment, we assume that it is a return address. Since scanning the full stack takes too long, we stop the stack scanning procedure when a sufficient number of return address candidates are found. In the current version, we stop when 5 return address candidates are found. Although quite ad-hoc, a restricted scan is effective in distinguishing different PCs because two different PCs cannot follow the same execution path to write system functions. (If they do, they are the same PC.) In our evaluation with a 3.4 GHz CPU machine, the performance overhead of the restricted scan was almost negligible, taking only 300-400 *n*sec per write system call.

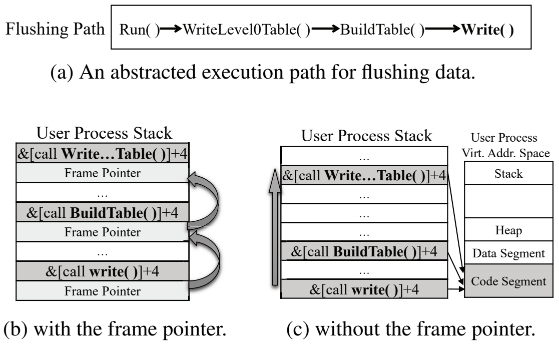


Fig. 5: An example execution path and its PC extraction.

#### 3.1.2 Computing PC for Indirect Writes

+ JVM (cassandra), mmap (mongo DB), internal buffer (mysql) - future work

#### 3.1.3 PC lifetime management

+ maintaining PC information for repeated executing application (hashing)

The prediction of PC lifetimes is rather complicated. The data lifetime of the append-only workload is defined from when a write request is issued until the TRIM command [13] is issued to the corresponding address. In order to measure the lifetime of data, the lifetime analyzer records the write time and PC value for each write request using its LBA. Upon receiving the TRIM command, the lifetime analyzer can compute the lifetime of the corresponding data using the recorded information. Note that, the same PC may generate multiple data streams with different lifetimes. We take the average lifetime as the PC's lifetime.

#### 3.1.4 Mapping PCs to SSD streams

The last step in PCStream is to map a group of PCs with similar lifetimes to an SSD stream. This is because each SSD supports a limited number of stream IDs. For example, SSDs used in FStream [4] and AutoStream [5] support only 9 and 16 streams, respectively. To properly group multiple PCs, the PC-to-stream mapper employs a simple 1-D clustering algorithm. In order to cluster PCs with similar lifetimes, the mapper calculates the lifetime difference between PCs. Then, PCs with the smallest lifetime difference are clustered into the same PC group. The mapper repeats this clustering step until all the PCs are assigned to their PC groups. For adapting to changing workloads, reclustering operations should be regularly performed. Since the number of PCs created by applications is not limited, the clustering algorithm must be efficient enough to quickly handle many PCs. Our goal in this work is to confirm the feasibility of using PCs, so we leave those issues as our future work.

### 3.2 Internal Stream Management

#### 3.2.1 Low resource requirement of IOS

+ explain different memory and power resource requirement

#### 3.2.2 IOS Management

+ explain block utilization trade-off, deciding number of IOS and mapping

## 4 Experimental Results

For our experiments, we have implemented PCStream in the Linux kernel 4.5. For an objective evaluation, we compared PCStream with three existing schemes: Baseline, Manual [2], and AutoStream [5]. Baseline stands for a legacy SSD that does not support a multi-stream feature. Manual is a RocksDB implementation which is manually optimized for multi-streamed SSDs. AutoStream is an LBA-based data separation technique which is implemented at the device driver layer. To understand the impact of the two-phase assignment, in addition, we compared PCStream with PCStream* which excluded the two-phase assignment feature.

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

### 4.1 Experimental Setting

### 4.2 WAF Comparison

We carried out a set of experiments using an SSD emulator which is based on the open flash development platform [15]. The SSD emulator emulates the behaviors of an SSD using host DRAM in the kernel level. Thus, it not only allows us to easily add new features, but enables to analyze detailed internal activities of an SSD. For our evaluations, we extended the SSD emulator to support a multi-streamed feature. We assume that the SSD emulator supports up to 8 streams (as with Samsung PM963 which was used in our experiments). We enhanced the original emulator so that it supported a multi-streamed feature as well as the two-phase stream assignment. The number of streams supported by the emulator was 8. The SSD emulator provided 12 GB capacity with 4 channels and 4 ways, and there were 8192 flash blocks, each of which was composed of 384 4-KB pages.

We compared WAF of the existing techniques with PCStream for the three scenarios, and the result is shown in Fig. 6. PCStream was quite effective in reducing WAF, thus achieving an equivalent level of the WAF reduction as in Manual. For example, both PCStream and Manual reduced WAF by 38% over Baseline for the `UR` case. Compared with AutoStream, PCStream was more effective, reducing WAF more by 35% on average. PCStream outperformed AutoStream by reducing WAF by 35% on average.

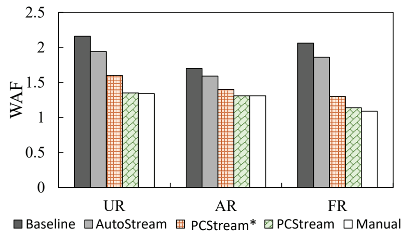


Fig. 6: A comparison of WAF on the SSD emulator.

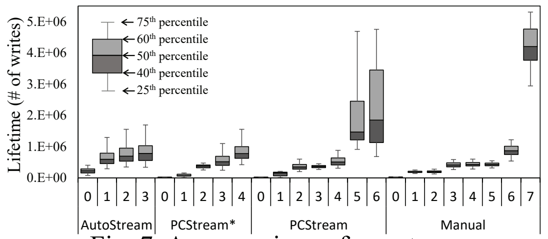


Fig. 7: A comparison of per-stream lifetime distributions.

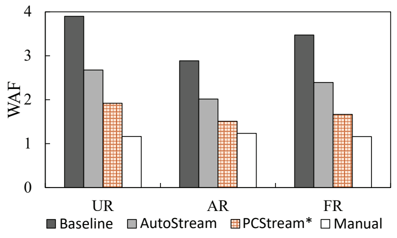


Fig. 8: A comparison of WAF on Samsung PM963 SSD.

Fig. 6 also indicates that the two-phase stream assignment technique is effective. PCStream outperformed PCStream* by 12% on average in the WAF reduction. As shown in Fig. 6, PCStream* reduced WAF by up to 30% over AutoStream. The result shows that separating short-lived data (e.g., log and flush) from long-lived one (e.g., compaction) using PC was quite effective in reducing WAF. Moreover, PCStream even showed similar WAF to Manual, reducing it by up to 38% over AutoStream. This additional gain of PCStream over PCStream* came from isolating long- and short-lived data in separate blocks by moving the long-lived data of the compaction-activity PC to substreams during GC at the SSD.

## 4.3 Per-stream Lifetime Distribution

In order to better understand how PCStream achieved a high reduction in WAF, we measured per-stream lifetime distributions under each technique for the UR scenario. Fig. 7 shows a box plot of data lifetimes from the 25th percentile to the 75th percentile. As shown in Fig. 7, streams in PCStream are divided into two groups, $G1 = \{0, 1, 2, 3, 4\}$ and $G2 = \{5, 6\}$, where $G1$ includes streams with short lifetimes and small variances (i.e., streams 0, 1, 2, 3, and 4) and $G2$ includes streams with large lifetimes and large variances (i.e., streams 5 and 6). Since the GC copy cost is affected by how data in $G1$ and $G2$ are mixed into the same block, PCStream can significantly reduce the GC overhead by avoiding such data mixtures in the same block by separating $G1$ and $G2$ into different streams. On the other hand, in AutoStream, three streams (i.e., streams 1, 2, and 3) show similar lifetime distributions with large variances without a distinct data separation pattern. In Fig. 7, we can also observe the effect of substreams. Streams 3 and 4 of PCStream*, which have large variances in lifetimes, are split into two substreams 5 and 6 in PCStream. This split reduces variances of streams 3 and 4, thus reducing the GC copy cost.

## 4.4 Effect of IOS Management

## 4.5 TBD - additional experiment

## 5 Conclusions

We have presented a new stream management technique, PCStream, for multi-streamed SSDs. Unlike existing stream management techniques, PCStream fully automates the process of mapping data to a stream based on PCs, which work well for append-only workloads as well as update workloads. By exploiting an observation that most PCs are distinguishable from each other in their lifetime characteristics, PCStream allocates each PC to a different stream. When a PC has a large variance in their lifetimes, PCStream refines its stream allocation during garbage collection and moves the long-lived data of the current stream to its substream. Our experimental results show that PCStream can reduce the average WAF by 35% over the existing automatic technique.

The current version of PCStream can be extended in several directions. For example, we plan to optimize the PC clustering method so that multiple PCs can be better clustered when the number of PCs significantly outnumbers the number of streams.

## References

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

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

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

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

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

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

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

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

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

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

[11] 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.

[12] 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.

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

[14] 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`.

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