FlashAlloc: Dedicating Flash Blocks By Objects

Jonghyeok Park*
Hankuk University of Foreign Studies
jonghyeok.park@hufs.ac.kr

Soyee Choi*
Samsung Electronics Co.
soyee.choi@samsung.com

Gihwan Oh
Sungkyunkwan University
wurikiji@skku.edu

Soojun Im
Samsung Electronics Co.
soojun.im@samsung.com

Moon-Wook Oh
Samsung Electronics Co.
mw.oh@samsung.com

Sang-Won Lee
Sungkyunkwan University
swlee@skku.edu

ABSTRACT

For a write request, today's flash storage cannot distinguish the logical object it comes from (*e.g.*, SSTables in RocksDB). In such object-oblivious flash devices, concurrent writes from different objects are simply packed in their arrival order to flash memory blocks; hence data pages from multiple objects with different lifetimes are multiplexed onto the same flash blocks. This multiplexing incurs write amplification, worsening the performance.

Tackling the multiplexing problem, we propose a novel interface for flash storage, *FlashAlloc*. It is used to pass the logical address ranges of objects to the underlying flash device and thus to enlighten the device to stream writes by objects. The object-aware flash storage can now de-multiplex concurrent writes from multiple objects with distinct deathtimes into per-object dedicated flash blocks. In essence, the interface enables the per-object fine-grained write streaming. Given that popular data stores tend to separate writes by logical objects, we can achieve, compared to the existing solutions, transparent streaming just by calling *FlashAlloc* upon object creation. Also, *FlashAlloc* is adaptive to workload changes, and liberates the stream conflicts in the multi-tenant environment.

Our experimental results using an open-source SSD prototype demonstrate that *FlashAlloc* can reduce the device-level write amplification factor (WAF) under RocksDB, F2FS, and MySQL by 1.5, 2.5, and 0.3, respectively and improve their throughput by 2.7x, 1.8x, and 1.2x, respectively. Also, *FlashAlloc* can mitigate the WAF interference among tenants: when running RocksDB and MySQL together on the same SSD, *FlashAlloc* reduced WAF from 2.5 to 1.6 and doubled their throughputs.

PVLDB Reference Format:
Jonghyeok Park, Soyee Choi, Gihwan Oh, Soojun Im, Moon-Wook Oh, Sang-Won Lee. FlashAlloc: Dedicating Flash Blocks By Objects. PVLDB, 16(11): 3266-3278, 2023.
doi:10.14778/3611479.3611524

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/JonghyeokPark/Flashalloc-Cosmos.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611524

* Most work done while in Sungkyunkwan University.

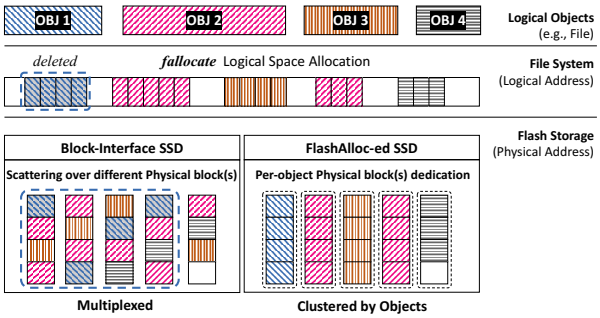

Figure 1: Data Placement: Multiplexing vs. Clustering

1 INTRODUCTION

Most data stores, including LSM (Log-Structured-Merge) tree-based KV (Key-Value) stores, relational DBMSs, and file systems manage data using logical objects: to name a few, SSTables in RocksDB, DWB (double write buffer) in MySQL, and segments in F2FS. And, upon each object creation, its logical space is secured in advance before writes are made to the object. For instance, RocksDB calls `fallocate()` right after creating an SSTable file so as to pre-allocate the logical address space for the file. The logical address range allocated from the call belongs to the file object. As such, host-side data stores can identify the corresponding object based on the LBA (Logical Block Address), which specifies the address of the host file system.

In addition, when an object is deleted, all its data pages tend to be invalidated together at once, having the same deathtime. Meanwhile, different objects are, though created and populated simultaneously, usually destructed at different points in time; they have different deathtimes. In summary, host software stacks manage data by objects; each object is the unit of logical space allocation and, in many cases, all its pages will have the same deathtime.

Though host software stacks can distinguish objects by their logical address ranges, the host-side semantic about objects' logical address ranges cannot cross the storage interface wall simply because no interface exists to pass it to the storage. As a result, today's flash storage has no knowledge about the belongs-to relationship between LBA address and object. Therefore, when concurrent writes from different objects interleave, the conventional *object-oblivious* flash storage cannot distinguish each write's object so that it has no choice but to simply append new data in their arrival order into flash blocks. As a consequence, writes from different objects

colocate in the same flash blocks. That is, each flash block will be multiplexed by data from multiple objects with different deathtimes, as illustrated in Figure 1. We call this situation as *multiplexing*.

As detailed in Section 2.2, the multiplexing is the main culprit of physical write amplification in flash storage, worsening the performance and lifespan of flash storage. Unfortunately, since no interface exists to offload the host semantic about objects' logical address range to the storage, the valuable semantic is discarded and the object-oblivious flash device can not stream writes by objects, incurring multiplexing and write amplification.

Addressing the multiplexing problem, we propose a novel interface, *FlashAllocate* (*FlashAlloc* in short), which is used to pass the host semantic about object's logical address range to the flash storage and thus to enlighten the storage to be *object-aware* in handling writes. To be concrete, after creating an object, a data store calls *FlashAlloc* with the object's local address range as a parameter to inform the flash device that the address range belongs to the same object. Then, upon receiving *FlashAlloc*, the flash device creates a corresponding *FlashAlloc* instance, which keeps the given address information, and thus is now aware of the address range of the object. At the same time, the flash device will secure physical flash block(s) where to place writes from the object and dedicate those blocks to the FA instance.

Once logical objects are *FlashAlloc*-ed, their writes will be, as illustrated in the right bottom of Figure 1, de-multiplexed into per-object dedicated flash blocks. In this regard, we say that flash storage, enlightened by *FlashAlloc*, can stream writes by objects. Given that popular data stores tend to separate writes by logical objects, the write streaming by objects is a natural way to achieve *grouping data by deathtime* [21] because logical objects have different deathtimes while all pages of each object become dead together at once upon the object destruction. The ultimate benefit of streaming writes by objects is to avoid GC-induced write amplification. For instance, as each SSTable file in Figure 1 is deleted, all its pages are invalidated, and thus, its dedicated block can be erased in its entirety, not causing any page relocation.

In essence, the *FlashAlloc* interface supports the *per-object fine-grained write streaming*, which is an alternative to existing solutions such as Multi-stream SSD and Zoned Name Space SSDs [6, 24] in *controlling the physical placement of writes* inside flash devices. The key contributions of this paper are summarized below.

- We make an observation that data stores with flash-friendly write patterns, contrary to the common belief, can experience severe write amplification on conventional SSDs and investigate the write multiplexing as the main culprit for the problem.
- We motivate that existing flash devices are object-oblivious simply because the host semantic about the object's logical address range cannot cross the storage interface wall.
- We propose a new interface, *FlashAlloc*, which allows to offload the host semantic about object's logical address range to the storage and thus to enlighten flash device to stream writes by objects, reducing device-level write amplification.
- We present the design principles and the architecture of *FlashAlloc*, and explain the rationales for its design decisions and how operations such as write and garbage collection work. And, we describe the implementation detail of the *FlashAlloc* prototype built using the Cosmos board [26] and also the changes made in file system and database storage engines.
- Our experimental results using the *FlashAlloc* prototype show that *FlashAlloc* can reduce WAFs in RocksDB, F2FS, and MySQL by 1.5, 2.5, and 0.3, respectively, and accordingly improve throughput by 2.7×, 1.8×, and 1.2×, respectively. And, when RocksDB and MySQL are run together, *FlashAlloc* can reduce WAF from 2.5 to 1.6, and double their throughputs.

2 BACKGROUND AND MOTIVATION

This section reviews several key concepts about flash storage and presents a few motivating examples about write amplification in flash-friendly data stores.

2.1 Flash Memory SSD

Here we review how the existing flash storage works and explain two key concepts of page deathtime and stream write by time.

FTL An FTL (Flash Translation Layer) is responsible for several key functionalities such as address mapping, GC and wear level management [31]. Because overwrites are not allowed in flash memory, a new page write should be handled in an out of place manner (*i.e.*, log-structured) - the old version of the page will be marked as invalid and new version will be stored in a new clean flash page. Thus, FTL has to manage the ever-changing address mapping between each page's logical address at the file system layer and its physical address in flash memory chips. Since the address mapping scheme is critical to the performance and lifespan of flash storages, most flash storage prefers the page-mapping FTL scheme among numerous address mapping schemes, mainly for performance reason at the cost of memory resource for managing the logical-to-physical mapping at the page granule [20, 25, 31].

Garbage Collection When clean space for new writes runs out, FTL has to reclaim new clean space by the garbage collection (GC in short) procedure. Upon GC, a victim block V is chosen, then its valid pages are relocated to a clean block B (*i.e.*, valid pages are read out from V and written back to B), and then V is erased and returned to the free block pool. After GC, new writes from the host will be appended to the remaining space in B. Relocating valid pages during GC amplifies physical writes inside flash storage. Informally, *write amplification factor (WAF)* represents the ratio of physical writes to flash memory over logical writes from the host.

Page Deathtime When a flash block page copy is overwritten or discarded by trim, it is termed as *dead*, with its invalidation moment referred to as its *deathtime* [21]. The distribution of deathtimes of pages in flash blocks is critical to determining the write amplification. For instance, let us assume that a flash block `fb1` stores only data pages from the same SSTable in RocksDB. All the pages in `fb1` will be dead when the SSTable is deleted after compaction. Then, the GC procedure can secure a new clean block without relocating any page but simply by erasing `fb1`, incurring no write amplification. In contrast, when a flash block stores pages with quite distant deathtimes and later is chosen as a victim for GC, many valid pages should be relocated to another block. Therefore, grouping pages by deathtimes is paramount to reducing write amplification.

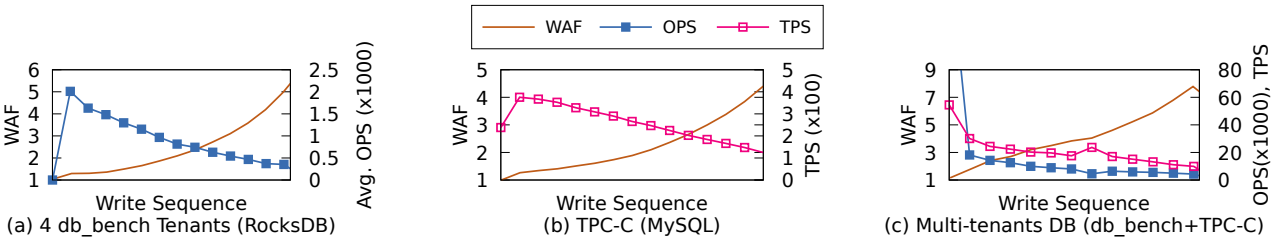

Figure 2: Three Database Workloads on a Commercial SSD: WAF and Throughput

Stream-Writes-by-Time Consider how today's flash storage handles writes when concurrent write requests from different objects interleave. For each write request with logical address (*i.e.*, start_lba), the existing flash storage cannot distinguish the object the data belongs to. Therefore, the conventional *object-oblivious* flash storage will simply append writes from different objects in their arrival order at the clean flash memory space [9]; we call this write policy as *stream-writes-by-time*.

2.2 Motivating Examples

It has long been believed that sequential writes are *flash-friendly*: less harmful than random writes in terms of write amplification [33]. In addition, mainly because of the parallelism in SSDs, issuing a single large data request in a write command yields higher bandwidth than making multiple small write requests [21]. With this expectation in mind, sequential writes have been opted for by many data stores such as LSM (log-structured merge) tree-based KV (key-value) stores [15, 19, 24, 34] and F2FS (Flash-Friendly File System) [27]. Contrary to the belief, however, log-structured sequential writes at such flash-friendly software stacks are not effective in reducing write amplification [13, 21, 44, 48].

In this section, we demonstrate that RocksDB with the so-called *flash-friendly* write pattern suffer from severe write amplification. We also motivate a write multiplexing problem in MySQL. In addition, we show that two tenants of RocksDB and MySQL interfere each other and thus exacerbate the write amplification further. While running each workload on top of a commercial SSD of 256GB, we measured its throughput and also the running WAF at the device-level using *smartmontools* [43] and present the results in Figure 2. For each workload, we describe the experimental setting and explain its IO architecture and dominant write patterns. In particular, we elaborate on why each workload experiences severe device-level WAF despite its *flash-friendly* write pattern.

RocksDB (Figure 2 (a)) RocksDB is a popular KV store used in many large-scale data services as well as databases [15, 16]. Since it uses Log-Structured-Merge (LSM) tree [34] as the primary data structure, the dominant write pattern from RocksDB is sequential in the unit of SSTable (Single Sorted Tables). Upon memtable flush or compaction, RocksDB creates new SSTable file(s), allocates a logical space of (by default) 64MB to each file via the `fallocate()` call, writes data, and then flushes the file. SSTables will be later deleted after compaction; all pages of an SSTable will be invalidated together upon the file deletion. Note that SSTable files which are created and populated simultaneously will be compacted and deleted at different points of time. Using the sequential batch write for each SSTable, RocksDB expects pages from the same SSTable and thus with the same deathtime to colocate in the same flash blocks and thus to barely cause write amplification.

To verify the WAF problem in RocksDB, we executed four concurrent RocksDB instances on Ext4 file system until the SSD was full, each of which runs the same db_bench's `fillrandom` workload against the initial database of 40GB [17]. Each RocksDB instance runs four user threads and four compaction threads concurrently to utilize storage better and mitigate the compaction overhead [16]. To further reduce the physical write amplification, the discard option in the Ext4 file system was also by default enabled [14]. While running the workload, we measured the average OPS (Operations Per Second) of four benchmarks and also the running WAF of the SSD in every five minutes, and plot the result in Figure 2 (a). Unexpectedly, the WAF has continued to increase over time, ending around five. This result about flash-unfriendly RocksDB has been reported consistently by other researchers [13, 44, 48].

Consider why RocksDB suffers from high WAF. Though each SSTable file is sequentially written, four compaction threads will flush their SSTables concurrently. In addition, each flush of 64MB SSTable file tends to split into smaller write requests due to file system fragmentation and kernel IO scheduling [11, 46]. Thus, write requests from multiple SSTables will interleave at the flash storage according to the stream-writes-by-time policy. Further, the striped architecture will divide each write request into smaller write chunks (*e.g.*, 4KB) and distribute them over multiple channels [23, 26, 38]. As a result, pages from multiple SSTables with distinct deathtime tend to be stored together in the same flash blocks.

MySQL (Figure 2 (b)) In order to guarantee the write atomicity in the presence of crashes, MySQL takes the redundant journaling approach using the special object, called *double write buffer* (DWB in short): before flushing dirty pages to their original locations, InnoDB engine first appends them sequentially to DWB. On system booting, contiguous logical address space of 2 MB is allocated to DWB. Though tiny in capacity, DWB will account for half of the writes in InnoDB to the storage. When full, the DWB space will be reused from the beginning. Therefore, the write pattern to DWB can be characterized as *sequentially appended and cyclically reused*. In addition, the pages written to DWB in the previous cycle will be overwritten in the next cycle. Namely, all the pages written to DWB during the same cycle will have the same deathtime.

Another characteristics in InnoDB engine's write pattern is that sequential writes to DWB and random writes to original database will interleave at the storage. Thus, according to the stream-writes-by-time policy at the flash storage, both types of data with different deathtimes will colocate in the same flash blocks. To evaluate the multiplexing effect, we measured TPS (Transactions Per Second) and running WAF while running the TPC-C benchmark [28] with 32 client threads and initial database of 150GB (*i.e.*, 1,500 warehouses) until the SSD was full, and present the result in Figure 2 (b). As the WAF at the device increases steadily over time, the transaction throughput (TPS) drops continuously. The multiplexing of DWB data and normal data pages accounts for the increasing WAF.

Multi-tenant Databases (Figure 2 (c)) With the ever-growing capacity of SSDs, it is not uncommon for multiple databases to share a single large SSD [29]. In such multi-tenant workloads, pages from *more* objects with *further distant* deathtimes are likely to be multiplexed onto the same flash blocks, exacerbating the the write amplification. This will in turn hinder the performance isolation among tenants [2]. To verify how the WAF behaves under multi-tenant databases, we ran db_bench and TPC-C together on the commercial SSD. While running both benchmarks with initial database of 80GB until the SSD became full, we measured their throughputs and the device WAF, and present the result in Figure 2 (c). When comparing WAF in the figure with those in Figure 2 (a) and (b), we confirm that the WAF in the multi-tenant case is almost double than that in either single tenant. Also, note that as the WAF spikes at the initial phase during the multi-tenant experiment, both OPS and TPS drop more rapidly than the throughput in either single tenant. In particular, we note that the multiplexing among multi-tenants worsens the performance interference.

3 THE MULTIPLEXING PROBLEM

This section presents an analysis of the object characteristics in data stores and delineates the multiplexing issue. It also motivates the absence of the storage interface for object-aware writing.

3.1 Object Characteristics in Data Stores

Understanding workloads is key to storage system design. However, little work has been conducted on characterizing objects in popular data stores from the perspective of the write multiplexing in flash storage. In this section, we make three observations about object characteristics in flash-friendly data stores: logical space allocation, write pattern, and deathtime. The design of *FlashAlloc* capitalizes on these characteristics.

Logical Space Allocation by Objects The host data stores manage data using logical objects such as SSTables in RocksDB and DWB in InnoDB. Each store will invoke the write system calls against such objects and those objects account for a dominant portion of total I/O. Prior to writing data to each object, data stores will allocate its logical address space at the file system layer in advance. For instance, RocksDB invokes the `fallocate()` call after creating a new SSTable but before writing data to it. Thus, the data stores will stream writes over the logical address space by objects.

Write Pattern The write pattern to each logical object is usually sequential (in either batch or append). For instance, when flushing a memtable, RocksDB invokes a write system call against L0 SSTable with the memtable data as parameter, which is an example of the batch sequential write. Meanwhile, the writes to DWB in InnoDB exemplify the append sequential write. Though the write pattern to individual logical objects is sequential, however, data stores with multiple write threads will usually issue writes from multiple objects concurrently to the storage. For instance, RocksDB with four compaction threads can flush four SSTables concurrently. As such, writes from different objects and thus with different deathtimes will interleave each other to the underlying storage.

Deathtime When a logical object is deleted, its all data pages become dead together. Meanwhile, different objects with almost the same birthtimes tend to have different deathtimes. For instance, SSTables concurrently generated at different levels by two compaction threads will be compacted at quite distant points of time.

3.2 Multiplexing

The three characteristics of objects in data stores discussed above can offer the chance of realizing the stream-write-by-deathtime policy [21] at the flash storage, thus minimizing the write amplification. That is, once flash storage can store pages by objects with distinct deathtimes into different flash blocks, its effect is to stream writes by deathtimes. However, despite the eager logical space allocation by objects and log-structured sequential write to individual object in data stores, concurrent writes from different objects in single or multiple tenants will interleave to the flash storage. Thus, according to the stream-writes-by-time policy, flash devices will pack those writes in their arrival order into flash blocks. An undesirable consequence is that pages from different objects are packed onto the same flash blocks. We call this phenomenon as *multiplexing*.

The multiplexing is the main culprit of physical write amplification in flash storage [21]. As flash blocks are multiplexed with pages with different deathtimes, write amplification is inevitable. Since logical objects have different deathtimes each other in most cases (*e.g.*, four SSTables in Figure 1 will be deleted at different points of time), pages in a multiplexed flash block will be incrementally invalidated at different points of time. When the block is chosen as a victim for GC, the remaining valid pages have to be relocated to another block, amplifying physical writes.

Suppose the case where four SSTable files are multiplexed (left-bottom in Figure 1). When the file (denoted as `OBJ 1`) is deleted, each of the four flash blocks still keeps three valid pages (*i.e.*, see dashed edge in Figure 1). Thus, if a block becomes victim, three pages have to be relocated. In contrast, when four files are de-multiplexed into different files (right-bottom in Figure 1), a flash block with its all pages invalidated is available. Thus, a clean block can be obtained without relocating any page. In particular, as demonstrated in Section 2, WAF becomes greater than five under the RocksDB databases (Figure 2 (a)) and even becomes larger than eight in multi-tenancy (Figure 2 (c)). To sum up, due to the multiplexing problem, the so called *flash-friendly* sequential writes are *no less harmful* than random writes in terms of write amplification [33].

3.3 Object-Oblivious Flash Storage in Writing

The main reason why the existing flash devices are object-oblivious in handling writes is that, from the logical address given in a write

request, flash storage cannot distinguish the logical object the address belongs to. This is in turn because the conventional block interfaces do not provide any mechanism to pass the relationship between objects and their logical address ranges to the storage. While the host software stacks are aware of the relationship, the valuable semantic cannot cross the storage interface wall. As a result, the useful host semantic is simply discarded and the flash device cannot stream writes by objects (*i.e.*, object-oblivious), incurring multiplexing and write amplification.

As will be reviewed in Section 7, though several novel interfaces [6, 7, 24] have been recently suggested to allow the host programs to control the data placement within the flash storage and thus to mitigate write amplification, none of them allows to pass the host semantic about object's logical address to the underlying storage. Because those new interfaces are not based on objects, they require the applications to use new concept such as stream identifier or zone identifier [6, 24]. To be worse, in the case of Zoned Name Space, applications are required to follow the strict sequential writes [6], which can hinder its wide adoption [1].

4 DESIGN OF *FLASHALLOC*

In this section we propose new interface for flash storage, called *FlashAlloc*, which is used to inform the flash storage that an logical address range belongs to an object and thus enable flash storage to allocate and dedicate physical flash blocks by objects. We describe its design principles, semantics, and architecture. Also, we give its use cases and discuss its benefits and limitations.

4.1 Key Idea and Design Principles

As discussed in Section 2, grouping data by deathtime is effective in reducing write amplification [21]. Considering that pages of individual object have the same deathtime while different objects have different deathtimes, grouping data by objects will have the effect of grouping data by deathtime. However, as pointed out in Section 3.2, the existing flash devices can not group writes by objects simply because they are unaware of the relationship between objects and their logical address ranges. This is, in turn, because no interface exists to allow host to convey the semantic to the flash storage.

Recognizing the missing interface, we introduce a new interface, *FlashAlloc*, to hint flash devices about the host-side semantic that all pages in a logical address range belong to the same logical object. With the help of the simple hint, flash devices should be able to place writes into distinct flash blocks by objects. If the new abstraction requires excessive change along the software stacks, it increases the system complexity and hinders its applicability and future extension, thus being unlikely to be accepted in the market, regardless of its effect. With this in mind, we set three design objectives of *FlashAlloc*. First, it takes advantage of the existing concept at the host layer, per-object logical address range. This is in stark contrast with other approaches (*e.g.*, multi-stream SSD and ZNS [6, 24]) which introduce new concepts (*e.g.*, stream-id and zone-id) and thus force applications to adapt to their interfaces. Second, host-side data stores should be able to leverage *FlashAlloc* with minimal change. In particular, required changes, if any, must be limited to the use of abstraction provided by *FlashAlloc*. Third, *FlashAlloc* aims at passing the host semantic to the storage without being limited to any specific application domain. So the abstractions of *FlashAlloc* must introduce minimal changes to the standards such as NVMe and the changes must not disrupt existing applications. This approach is novel in that it turns the common knowledge at host (*i.e.*, a logical address range constitutes an object) into a strong point for flash storage (*i.e.*, to be able to stream writes by objects).

4.2 Interface

As a way to pass the information that a logical address range constitutes an object from the host to the flash storage, we propose new `FlashAlloc(logical_addr_range)` interface, as detailed below.

FlashAlloc ({LBA, LENGTH}*) *FlashAlloc* informs flash storage that the logical address range denoted by the parameter, `{LBA, LENGTH}*`, belongs to one object. As indicated by *, an address range can consist of one or multiple logical chunks. Each chunk is presented by a pair of `LBA` and `LENGTH` which represent its starting address and length, respectively.

Since the storage command is not always available to applications (*e.g.*, database engines) that access objects through a file system, we exploited the `ioctl` infrastructure so that the *FlashAlloc* command can pass through the file system to the storage device, instead of invoking the new command directly from applications.

More importantly, calling a *FlashAlloc* command, the host expresses its intention that it will perform operations on the given logical address range, *LS*, as an *integral unit for writes*: once a portion of the dataset is written, all the dataset is going to be written once, and later they will be invalidated together nearly at the same time. This is a useful hint for write optimization in flash storage. Given a *FlashAlloc* command, flash storage will dedicate the corresponding physical flash block(s) (*PS*), and then store all writes from *LS* in the arrival order into *PS*; In this way, flash storage will guarantee the physical clustering of all pages from the same logical object. In particular, note that even when the writes from an *FlashAlloc*-ed object are *spatially fragmented* (*e.g.*, due to file system aging [46]) or *temporally split* (*e.g.*, due to log-appending in F2FS), they are guaranteed to be eventually clustered into the same flash block(s). Thus, when properly *FlashAlloc*-ed, concurrent write streams from different logical objects will be de-multiplexed into each own dedicated flash block. The beauty of *FlashAlloc* lies in that it can achieve the transparent write streaming without the hassle of assigning stream-id or zone-id to each write request [6, 24].

Use Cases As illustrated in Figure 3, popular database engines and file systems have write-intensive objects (*e.g.*, SSTable, DWB, segment) whose IO patterns fit well with the purpose of *FlashAlloc*: each object is written sequentially just once and later becomes dead in its entirety at the same or similar time. In addition, objects with such "write-once and dead-at-once" pattern are ubiquitous in most data stores: numerous LSM-based KV stores, WAL log files in relational databases, two journaling modes (RBJ and WAL) in SQLite, and file system journaling. In addition, *FlashAlloc* will naturally stream writes from different tenants so that it is, as demonstrated in Section 6, quite beneficial in reducing write amplification and performance interference in multi-tenant database environments [2].

4.3 Architecture

Figure 3 illustrates the architecture of *FlashAlloc*. Using the figure, we explain the concept of FlashAlloc instance and its physical space management. Also, we explain how write and read operations work and describe how the logic of trim and garbage collection is extended to support *FlashAlloc*. Though the page-mapping FTL is assumed in this paper, *FlashAlloc* can be seamlessly supported by other FTL schemes such as block and hybrid mappings [31].

***FlashAlloc* Instance** Upon receiving a *FlashAlloc* command with a logical address range, flash storage will create its corresponding *FlashAlloc* instance (in short, FA instance). In addition, flash storage will secure the corresponding physical space (*i.e.*, one or more clean flash memory blocks whose total size amounts to the given logical address range's size), and dedicate the space to the instance[1]. The physical space to an FA instance is allocated in the unit of flash memory block (*e.g.*, usually 2MB in flash memory chips) so as to isolate writes from the instance into the dedicated blocks The dedicated blocks for FA instances are called as *FlashAlloc*-ed. Once multiple blocks are allocated to an instance, data for the instance will be striped across channels for better bandwidth [26]. As illustrated in Figure 3, an FA instance consists of three metadata: logical address range, physical flash blocks, and physical location for next write. Next write for an FA instance will be appended to the page pointed by `next_write_ptr`.

Active FA Instances Once created, every FA instance and its metadata will remain active and also be managed persistently until its destruction. An FA instance and its metadata will be destructed once its physical space is filled up. As depicted in Figure 3, multiple FA instances could be active at a point of time. Note that the logical address ranges of all instances should be disjoint from each other. Meanwhile, the number of active FA instances will remain rather small in practice, though unlimited in principle. For instance, considering that an SSTable file in RocksDB is, once created, quickly filled by memtable flushing or compaction, its FA instance will be destructed shortly after created. Thus, when running a RocksDB, the number of active FA instances will not be greater than that of concurrent compaction threads (*i.e.*, four by default). Similar argument can be made for F2FS segments.

Non-*FlashAlloc*-ed Objects While good for objects with *sequential or log-appending write patterns*, *FlashAlloc* is not intended for objects with random write pattern. For the latter type of objects, *FlashAlloc* is not recommended to be called. Thus, flash devices with *FlashAlloc* should be able to support such non-*FlashAlloc*-ed objects as well; writes from non-*FlashAlloc*-ed objects can be handled exactly the same as in the conventional SSD and stored in *normal* blocks. An object will be regarded as non-*FlashAlloc*-ed once the *FlashAlloc* call is not made when allocating logical space for the object. Accordingly, our *FlashAlloc* architecture has to support two types of blocks, *FlashAlloc*-ed and normal, as illustrated in the bottom of Figure 3. In this regard, our approach is in contrast with the ZNS interface [6] which strictly assumes all writes to be sequential and thus cannot support workloads with random writes.

Write For a write request with two parameters, `LBA_start` and `length`, indicating the starting LBA and length of the data respectively, flash storage first probes its matching FA instance using the `LBA_start`, as depicted in Figure 3. If a matching instance is found, the write data will be appended to a FlashAlloc-*ed block* pointed by the instance's *next_write_ptr*, and the corresponding entry in *L2P* table and `next_write_ptr` value will be accordingly adjusted. Otherwise, the request comes from non-*FlashAlloc*-ed objects. In this case, as in the conventional SSDs, the writes will be stored in *normal* block. In this way, our approach supports two write policies: stream-write-by-object for *FlashAlloc*-ed objects and stream-write-by-time for normal writes.

Read While *FlashAlloc* aims at reducing WAF by controlling the placement of writes into different blocks by objects, its read operation will proceed exactly the same as in the conventional page-mapping FTLs: after looking up the physical page number (*i.e.*, block-id + page-offset) from the mapping table with the given LBA, FTL reads the page from the corresponding flash block.

Trim The trim command was introduced to notify the flash storage that a set of logical pages is no longer valid [42]. Upon receiving a trim command, flash storage will invalidate the relevant pages, preventing them from being unnecessarily relocated. In our *FlashAlloc* architecture, the trim command will be handled same as in the conventional flash storage with one exception. Considering that pages in an *FlashAlloc*-ed object have the same deathtime and are also clustered in the same physical flash block(s), the trim command against the object can complete by erasing the block(s) instead of invalidating individual pages. In this sense, *FlashAlloc* enables to achieve nearly *zero-overhead trim*. In fact, the trim command is, though effective in lowering write amplification, known to induce non-trivial run-time overhead (*e.g.*, trim spikes [16]); thus data stores resort to the *delayed discard* policy to mitigate the effect of trim spikes on the write latency [27, 32]. The trim optimization enabled by *FlashAlloc* will remove the *trim-induced stalls* so that the developers are free from the burden of devising deliberate *rate limiting* to file deletion [32] or *delayed discard* [27].

Garbage Collection (GC) When no space is available for new writes, FTL has to conduct the GC operation: copying valid pages from a victim block to a clean block, *fb* and then returning *fb*. Unfortunately, the conventional GC algorithm returning a partially clean block *fb* is inappropriate for FA instances. Recalling that *FlashAlloc* intends to disallow pages from different objects to mix in the same flash block(s), each FA instance needs total-clean block(s). Thus, FTL has to handle two cases of GC differently: The first case is when *FlashAlloc* is called, and the second one is when free space is unavailable for normal write. Recall that when a *FlashAlloc*-ed object is trimmed, all its blocks will be returned as free and thus the free block pool is likely to have some free blocks. In the case when GC is triggered for normal write, FTL will first check the free block pool for a free block and, if found, use it. Otherwise, it will carry out the conventional GC: it chooses a victim block with

[1]For the simplicity of discussion, we assume synchronous physical space allocation throughout this paper. Note, however, that the *FlashAlloc* interface does not mandate synchronous allocation. Instead, as long as the physical space is secured prior to the first write to the instance, either asynchronous or even on-demand physical space allocation (or securing) is acceptable, which may bring some optimization opportunities. We leave such an interesting optimization topic as future work.

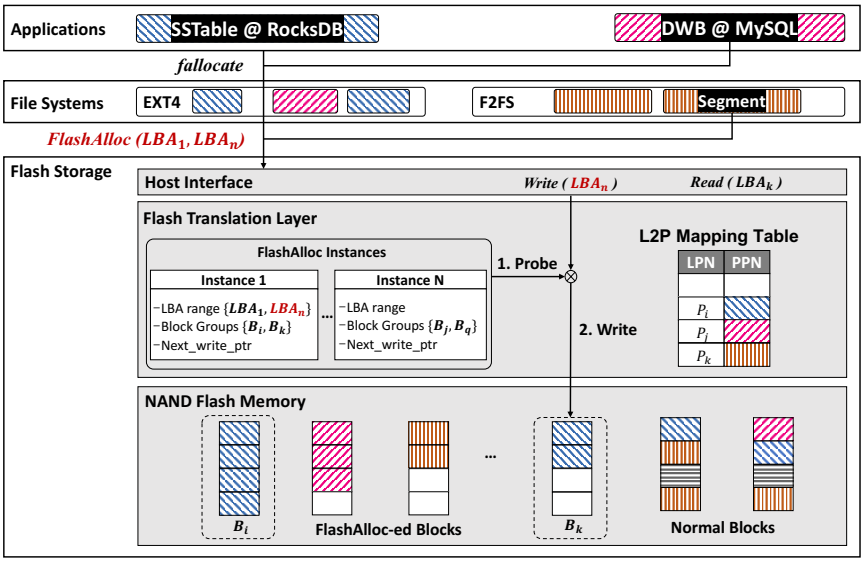

Figure 3: *FlashAlloc* Architecture

the least valid pages and merges the victim. In the second case, when a *FlashAlloc* call is made, FTL checks whether clean blocks are available in the free block pool and, when unavailable, has to merge multiple blocks to secure total-clean block(s) [25].

In addition, while selecting victim blocks for each of two GC cases, block types have to be taken into account so that pages from normal blocks are not mixed with those from *FlashAlloc*-ed blocks. In the case of GC for normal writes, only normal blocks can be victims. Otherwise, once an *FlashAlloc*-ed block is chosen as victim, non-*FlashAlloc*-ed normal writes should co-locate with the relocated pages from the old *FlashAlloc*-ed victim block. In the case of GC for *FlashAlloc*, as discussed above, one or more blocks with the same time have to be merged to secure total-clean block(s) for *FlashAlloc*. In this sense, we call our algorithm as GC-By-Block-Types. Interestingly, the GC-By-Block-Type algorithm will *adaptively allocate* the space of *FlashAlloc*-ed and normal blocks: depending on victim block type, one region grows while the other shrinks. For instance, if a new *FlashAlloc* block is in need and no free block is available, multiple normal blocks have to be merged so as to secure a total-clean block, enlarging *FlashAlloc*-ed region. In this way, the space allocation to both regions will adapt to the changing workloads, requiring no static allocation or tuning.

4.4 Advantages and Limitations

Advantages *FlashAlloc* is the first work which allows host applications to hint the storage that a logical address range belongs to the same object. It provides three advantages. First, enlightened by *FlashAlloc*, flash storage can now cluster data from the same object into the same flash blocks, minimizing write amplification. In particular, data from objects which are even *logically fragmented* will be *physically de-fragmented* into the same flash block(s). In this regard, *FlashAlloc* could be a clean solution to the logical and physical fragmentation problem in flash storage [41, 44]. Second, since the abstraction taken in *FlashAlloc* complies to the abstractions used for logical space allocation in data stores (*e.g.*, the `fallocate()` call in Ext4 and segment in F2FS), the existing software stacks can achieve write streaming with minimal changes, as detailed in Section 5. Third, since the *FlashAlloc* semantic does not require involving intermediate layers such as the kernel block layer, the command can be easily incorporated into the existing storage interfaces such as SAS and NVMe (*e.g.*, using the vendor specific command).

Limitations While effectively de-multiplexing objects with sequential write-once pattern, *FlashAlloc* is not a panacea for the write amplification problem and also has a few limitations, some of which are the trade-offs of being object-aware.

First, *FlashAlloc* does not suit for all objects and thus needs to be judiciously used depending on each object's write pattern. For instance, *FlashAlloc* is not intended for objects with small and random overwrites (*e.g.*, relational database's tablespace under OLTP workload). Also, *FlashAlloc* is not suitable either for tiny objects whose sizes are less than the size of a flash block or for objects with append-only writes of unknown size. In particular, *FlashAlloc*-ing tiny objects makes flash blocks under-utilized[2]. Such objects are recommended to be non-*FlashAlloc*-ed, stored in a non-FA instance, and managed by the conventional FTL. This approach would be, instead of forcing one type of write pattern (*e.g.*, ZNS [6]), a flexible and practical approach for supporting real workloads with various types of objects with different write patterns.

Second, the effect of *FlashAlloc* will diminish as writes are skewed to non-*FlashAlloc*-ed objects which can cause the write amplification. However, it is always beneficial to isolate objects suitable for *FlashAlloc* into dedicated flash blocks, as illustrated in Section 6.

[2]For the simplicity of discussion, the flash block is assumed as the unit of physical space allocation for each *FlashAlloc*-ed object in this paper. However, we do not exclude the case where multiple tiny objects share one flash block. It would be always better to cluster pages from tiny objects, as long as they have the same deathtime, into the same flash block. We leave this design and implementation as future work.

Third, when the *FlashAlloc* call is made at a coarser granule that encompasses multiple objects with varying lifetimes, the fragmentation problem can still persist. As such, it is crucial for the user to call *FlashAlloc* at the granule of a single object whose pages have the same deathtime. Fortunately, in most cases, the appropriate unit for *FlashAlloc* is clear to users (*e.g.*, SSTable and DWB).

Fourth, *FlashAlloc* incurs the run-time overhead of memory and computation, though marginal. To ensure instant access to FA instances, the FTL firmware has to manage the metadata for each active instances on the Cosmos board's DRAM. However, considering each active instance needs a small memory footprint (*i.e.*, tens of bytes) and the number of active instances is limited in practice, *FlashAlloc* strikes a reasonable trade-off between performance and memory usage. In addition, for every write request, the FTL firmware first has to determine the matching active FA instance while comparing the given LBA against the LBA ranges of all active instances. Though, we can remove the computational overhead by accelerating the matching step with special hardware and also pipelining the step with flash memory access [26].

Lastly, *FlashAlloc* trades off the I/O latency for small objects which are dedicated by a few flash block(s). For instance, the size of each segment in the F2FS file system is 2MB by default and is thus, when *FlashAlloc*-ed, dedicated with single flash block. Consequently, when a large write request is made to the segment, the write will be directed to the single flash block, taking long to complete. In contrast, in the case of the conventional SSDs, a large write request is striped in parallel over multiple flash blocks across different channels and ways, completing faster [9]. A similar argument can be made for the read latency. However, for large objects which are *FlashAlloc*-ed by multiple flash blocks (*e.g.*, 64MB-sized SSTables in RocksDB), this will not hold for. In addition, *FlashAlloc* will not sacrifice the throughput when multiple I/O requests are concurrently made. Most importantly, the reduced WAF by *FlashAlloc* will can compensate the prolonged I/O latency for small objects.

5 IMPLEMENTATION

This section presents the implementation details of the *FlashAlloc* architecture. The dominant portion of the *FlashAlloc* implementation is made into the FTL firmware of the Cosmos board written in C. The *FlashAlloc* command was prototyped via vendor unique command. A user-level library that implements a protocol for the new commands via the *ioctl* system call supports applications and SSDs. This approach not only allows to quickly prototype the concept but also to make the prototype portable to most file systems. In addition, this section describes the changes we made in file systems and database engines to enable them to run on the *FlashAlloc* interface. Note that the changes are, as summarized in Table 1, marginal and moreover local to a few modules.

5.1 Changes made in Applications

RocksDB As explained in Section 2, RocksDB manages key-value documents using SSTables. In particular, after creating each SSTable file, RocksDB secures its logical address space (whose size is by default 64MB) in advance by calling `fallocate()` for the file. Once allocating the logical space for the given `fallocate()` call, RocksDB engine calls *FlashAlloc*. For this reason, every SSTable's data will be streamed to its dedicated flash block(s). Flash blocks dedicated to each SSTable remain full of valid pages until the SSTable is later compacted and deleted. When an SSTable is deleted, all its pages will be trimmed and thus invalidated altogether at once and accordingly all flash blocks dedicated for the SSTable can be simply erased. Hence, as shown in Section 6, RocksDB can achieve near ideal WAF (*i.e.*, 1) transparently with minimal code change.

Table 1: Lines modified across applications to use *FlashAlloc*

Applications		Lines Added	Lines Removed
RocksDB	(v6.10)	72	-
F2FS	(v5.4.20)	26	-
MySQL/InnoDB	(v5.7)	74	16
FTL (cosmos+)	(v3.0.0)	1683	193

MySQL/InnoDB In order to isolate DWB pages from normal ones into different flash blocks and thus to reduce write amplification [12], we modified the InnoDB engine to call *FlashAlloc* with the logical address range of DWB as parameter before writing to the journal area for the first time. To obtain the address range was used the `FS_IOC_FIEMAP` ioctl call. Also, whenever DWB is cyclically reused, the trim call is made for the journal area so as to invalidate all the old pages and and thus to make the old *FlashAlloc*-ed block to be erased in its entirety. As shown in Table 1, the changes made in InnoDB engine were minimal - less than 100 lines of code change were made at two modules of `double-write-buffer` and `file`.

5.2 Changes made in FTL

We have prototyped *FlashAlloc* on the OpenSSD Cosmos board [26] by extending its firmware. The Cosmos board adopts a page mapping scheme for flash memory management, as in most contemporary SSD products. The board is connected to a host system through the NVMe interface. Main technical issues encountered while embodying *FlashAlloc* on the board are summarized below. Note that the existing FTL can support *FlashAlloc* with moderate changes in its codebase, as shown in Table 1.

FA Instances For each *FlashAlloc* command, a corresponding FA instance is created in Cosmos board's DRAM, which contains its logical address range, the list of flash blocks dedicated to the instance, and the current write pointer. The memory requirement per a logical FA instance is, though varying slightly depending on address range and the number of flash blocks, just several tens bytes. Thus, considering that the number of active FA instances is in practice limited (*e.g.*, less than 100), small amount of DRAM (*i.e.*, several tens KB) will suffice to maintain active FA instances.

GC and Block Type The existing GC firmware in the Cosmos board was extended to support the *GC-By-Block-Type* policy. Also, to distinguish two types of blocks, normal and *FlashAlloc*-ed, one bit flag, `FA-BLK`, was added to the `block_header` struct in the Cosmos firmware. The `FA-BLK` flag will be set on dedicating a block to an FA instance and reset when the block is erased and returned as free.

Probing the matching FA instance For a write request, FTL should be able to quickly probe the matching FA instance using

the given LBA address. If the probing fails (*i.e.*, no matching instance exists), the request is not for active FA instance thus will be written to non-FA instance. To determine whether the given write is for active FA instance or not, a flag bit was added to every entry in page-mapping table. The flag bit of every relevant logical page is set when an *FlashAlloc* command is invoked and later reset when the page is overwritten or discarded. The next issue is, when the flag is turned on, how to probe the matching instance. While there should be alternative implementations such as hardware-acceleration and pipelining, rather a simple approach was taken for fast prototyping. That is, while scanning each of all active FA instances, we check whether its logical address range contains the start_LBA in the given write request. Once a matching instance is found, the write will be appended at the physical space pointed by the `next_write_ptr` of the instance.

6 PERFORMANCE EVALUATION

In this section, we present performance evaluation carried out to analyze the impact of *FlashAlloc* on key-value store, log-structured file system, relational database, and multi-tenancy.

6.1 Experimental Setup

All experiments were conducted on a Linux platform with 5.4.20 Kernel running on an Intel Core i7-6700 CPU 3.40GHz processor with two sockets of four cores and 50GB DRAM. The host machine has two storage devices, 16GB Cosmos OpenSSD and 256GB Samsung 850 Pro SSD. The Cosmos OpenSSD employs a controller based on Dual Core ARM Cortex-A9 on top of Xilinx Zynq-7000 board with 256KB SRAM, 1GB DDR3DRAM, and 16GB MLC Nand flash memory [37]. The Cosmos OpenSSD was used as the main storage device for the experimental data and connected to host using PCIe interface. The over-provisioning area in the board was set to 10% (*i.e.*, 1.6GB) for all experiments. In Cosmos OpenSSD, the flash block size is 2MB and the page size is 16KB. The Samsung 850 Pro SSD was used as the log device when MySQL/InnoDB was run.

6.2 Workloads

To demonstrate the benefit of *FlashAlloc*, we used a synthetic workload and two realistic workloads, db_bench and TPC-C. The `fio` tool was used to generate a synthetic workload. And, to evaluate the effect of *FlashAlloc* on key-value stores, we ran the db_bench benchmark using RocksDB on ext4 file system. The same db_bench workload was run also using F2FS [27] to test the impact of *FlashAlloc* on log-structured file system. In addition, the TPC-C benchmark was used to measure the effect of separating DWB in MySQL/InnoDB into dedicated flash blocks. Finally, to highlight the benefit of *FlashAlloc* in multi-tenancy, we ran db_bench using RocksDB and TPC-C using MySQL concurrently on Ext4 file system. In all experiments, the direct I/O option (`O_DIRECT`) was enabled to minimize the interference from file system's page caching and the `TRIM` option was turned on for both file systems. To compare the impact of *FlashAlloc*, we ran those workloads using the vanilla databases and file systems on the Cosmos board running the original FTL and also ran them using the modified versions with the board supporting *FlashAlloc*. The three workloads are summarized below.

FIO The Flexible I/O (FIO) benchmark is commonly used to test the performance of file and storage systems [18]. It spawns a number of threads or processes doing a particular type of I/O operations as specified by the user parameters.

db_bench RocksDB provides db_bench as the default benchmark program [17]. The `fillrandom` workload which write key-value pairs in random key order was run against empty database till the Cosmos board became full.

TPC-C The `tpcc-mysql` tool [39] was used for TPC-C benchmarking [28]. The benchmark was run using 32 clients against initial database of 8GB until the storage device became full.

6.3 Performance Analysis

Let us briefly review the overall performance benefit of *FlashAlloc* using Figure 4. While running four experiments using vanilla and *FlashAlloc*-ed configurations, we measured the throughput of each benchmark program and the running WAF at the Cosmos device every minute till no space is left in the Cosmos device, and present the results in Figure 4. In the figure, the X-axis represents the time and the left and right Y-axis does the running WAF and the throughput of each benchmark, respectively. As shown in Figure 4, *FlashAlloc*-ed version outperforms the vanilla one considerably in terms of throughput as well as WAF consistently across all four experiments. The running WAF gaps between two versions are ever-growing over time in all experiments. That is, as the Cosmos board is filled with data, the effect of de-multiplexing different objects into different blocks in *FlashAlloc* becomes outstanding. In particular, the running WAF in *FlashAlloc*-ed version remains close to 1 even at the ends of RocksDB and F2FS experiments.

Synthetic FIO Workload Before explaining the effect of *FlashAlloc* on realistic workloads in Figure 4, let us show the benefit of *FlashAlloc* using a synthetic write workload. For this, using the `fio` tool, we created eight 2GB files on Linux and Cosmos board and ran eight threads, each of which performs random overwrites in the unit of 2MB against its dedicated 2GB file. The same experiments were conducted in two modes, vanilla and *FlashAlloc*-ed. In the *FlashAlloc*-ed mode, before invoking each 2MB overwrite, *FlashAlloc* was called a prior so as to secure a dedicated flash block to store new data. While running each experiment during one hour, we measured the device WAF and the write bandwidth and plotted the result in Figure 5. *FlashAlloc* has reduced the device WAF from 3.1 to 1 and has doubled the write bandwidth from 75MB to 150MB.

In addition, to further highlight the effect of *FlashAlloc* when the multiplexing degree is increased, we carried out another experiment by increasing the number of concurrent write threads to 32 in `fio` tool and thus decreasing the per-thread file size to 512MB, and present the result in Figure 5(b). Note that, under more concurrent write threads, a flash block in the Cosmos board will be multiplexed by more files with more deviating lifetimes. As shown in Figure 5, both WAF and write bandwidth of the vanilla version under 32 threads become worse than those under 8 threads. Even in the case of 32 threads, *FlashAlloc* has reduced the device WAF from 4 to 1 and thus has tripled the write bandwidth (*i.e.*, roughly from 60MB to 180MB). The considerable gain of the *FlashAlloc* version is direct reflection of reductions in the garbage collection overhead.

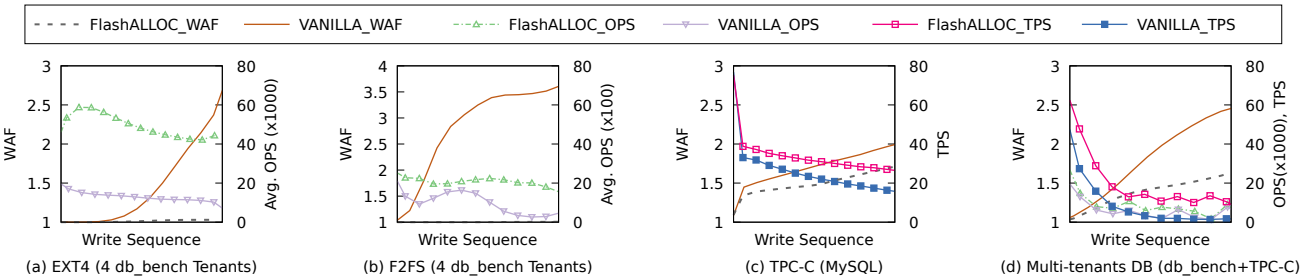

Figure 4: Effect of *FlashAlloc*: Write Amplifications and Throughput in Database Workloads

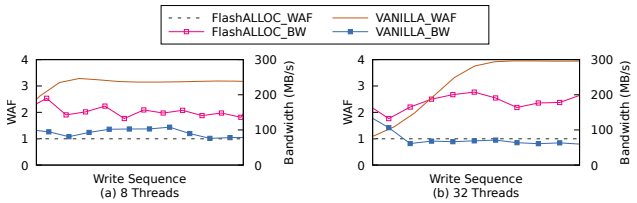

Figure 5: Effect of *FlashAlloc* on FIO Benchmark

RocksDB on EXT4 To analyze the effect of *FlashAlloc* on RocksDB engine, we ran 4 RocksDB instances concurrently on top of two configurations of ext4 file system and the Cosmos board, vanilla and *FlashAlloc*-ed. Each RocksDB instance was run with the `fillrandom` workload in the db_bench benchmark. In order to minimize the interference from flushing the WAL log, the log file was stored in a separate storage device. While running both experiments, we measured device-level WAF and average OPS of four RocksDB instances over time and present the result in Figure 4 (a). In the case of vanilla mode, as SSTables at different levels are simultaneously created and populated by multiple compaction threads in RocksDB, they are multiplexed into the same flash blocks. Recall that SSTables at different levels will be compacted and thus deleted at different points of time. As such, in the vanilla version, device-level WAF increased steadily while RocksDB's OPS decreased, which is consistent with the result in Figure 2 (a). In contrast, in the case of *FlashAlloc*-ed mode, WAF at the Cosmos board, as expected, remained nearly one even till the end of the experiment and accordingly, compared to the vanilla version, the average RocksDB's OPS improved by 2.7×. This result clearly illustrates that *FlashAlloc* can drastically reduce write amplification in RocksDB by de-multiplexing SSTables into different flash blocks.

Nonetheless, write amplification was not completely removed in *FlashAlloc*-ed mode. The running WAF was 1.1 at the end of the experiment. The residual physical write amplification is attributable to metadata files. RocksDB maintains several metadata files (*e.g.*, MANIFEST and CURRENT) to keep track of database state changes whose write patterns are not log-structured instead random writes. Though the sizes of those files are relatively small, they contribute non-marginal fraction of total writes from RocksDB. Those random writes go to the non-FA instance which is managed by the conventional Greedy FTL and hence incur write amplifications.

RocksDBs on F2FS The F2FS file system [27], a flash-friendly variation of log-structured file system [40], has been gaining the popularity on flash devices successfully. F2FS is argued to be flash-friendly since it takes the out-of-place-update (OPU) policy and also the sequential write pattern [27].

To evaluate the effect of *FlashAlloc* on F2FS, we ran four RocksDB instances each with the `fillrandom` workload on top of two configurations of F2FS and Cosmos boards, vanilla and *FlashAlloc*-ed. For *FlashAlloc*-ed F2FS version, we modified the F2FS file system to call the *FlashAlloc* whenever allocating new segment. While concurrent log writes from active segments are multiplexed into the same flash block in the vanilla configuration, writes from each segment is perfectly isolated into its dedicated flash block in the *FlashAlloc*-ed version. Thus, the *FlashAlloc*-ed F2FS can nearly remove write amplification due to the write multiplexing in the vanilla version; the WAF at the final phase was reduced from 3.5 to 1.1, as shown in Figure 4 (b). The residual WAF of 0.1 in *FlashAlloc*-ed version is presumably contributed by random writes for hot metadata in F2FS [27]. Accordingly, *FlashAlloc*-ed version outperforms the vanilla version about by 3× in terms of the db_bench's OPS at the end of experiment. The result indicates that *FlashAlloc* can be a fundamental solution to the *log-on-log* problem [44, 48] by allowing to perfectly align logical segments in F2FS with physical flash blocks. Namely, *FlashAlloc* enables F2FS to achieve an ideal WAF of 1 and thus to realize its full potential of log-structured write.

Though effective in reducing WAF, *FlashAlloc* can negatively affect the read and write latency for small *FlashAlloc*-ed objects such as segments in F2FS. In the experiment for Figure 4 (b), the segment size is set by default to 2MB and the size of the flash block used in the Cosmos board is also 2MB. Consequently, each *FlashAlloc*-ed segment is mapped to single flash block and the write request to the segment (usually, several tens of KBs) will go for the flash block and the write latency will be limited . In contrast, in the vanilla mode, the write request will be striped across multiple flash blocks in different channels and ways [9]. A same argument can be made for the segment read operations during F2FS segment cleaning. This seems to be one of the main reasons why the relative OPS improvement by *FlashAlloc* (*i.e.*, about 2×) is rather smaller than the WAF improvement by *FlashAlloc* (*i.e.*, roughly 3.5×).

DWB in MySQL/InnoDB Under workloads with both *FlashAlloc*-ed and non-*FlashAlloc*-ed objects, the effect of *FlashAlloc* will depend on the write amount to each object type - the more writes are made for *FlashAlloc*-ed objects, the less WAF and wear-out and

Table 2: Effect of *FlashAlloc* on Latency (DB-Bench)

(unit: us)	DB-Bench Operations			Block I/Os Latency
	Avg.	99th	99.9th	Avg. Read
Vanilla	140.2	20.9	5694.2	34.89
FlashAlloc	94.4	18.3	3401.2	18.95

the higher performance are expected. Conversely, as writes are skewed towards non-*FlashAlloc*-ed objects, the effect of *FlashAlloc* diminishes. Though, regardless of the write skewness to either object type, it is always beneficial to apply *FlashAlloc* to appropriate objects and thus isolate them to dedicated flash blocks. To evaluate the effect of separating the DWB object with cyclic and sequential writes from the main database with random writes, we measured the throughput and the device-level WAF while running the TPC-C benchmark using the vanilla and *FlashAlloc*-ed MySQL/InnoDB engines, and present the result in Figure 4 (c). Recall that, in the case of *FlashAlloc*-ed version, half of writes goes to the FA instance for DWB while the other half does to the non-FA instance for original database. Hence, the write amplification induced by the random writes in non-FA instance is inevitable. Nevertheless, *FlashAlloc* can reduce the increased write amplification by one third (*i.e.*, 1.2 to 0.8) and thus improve the throughput by 50%. The benefit of *FlashAlloc* on DWB should also apply to other ubiquitous journal objects, such as WAL files in RocksDB, SQLite, and relational databases.

Multi-Tenancy As discussed in Section 2, when run together on the same SSD, multi-tenants can interfere each other in terms of write amplification. To demonstrate the effect of *FlashAlloc* on multi-tenant databases, we ran two databases concurrently on Ext4 file system and the Cosmos board, one RocksDB instance (used in Figure 4 (a)) and one MySQL instance (used in Figure 4 (b)), in vanilla and *FlashAlloc*-ed modes, respectively, and present the results in Figure 4 (d). In the case of vanilla version, the WAF in multi-tenancy is worse than that in either single tenant (*i.e.*, Figure 4 (a) and (c)), which is consistent with the result in Figure 2 (c) obtained from commercial SSDs. In the case of *FlashAlloc*-ed version, the WAF in multi-tenancy remains lower than that in either single tenant. As a result, both benchmarks' throughputs in *FlashAlloc*-ed mode are higher than those in vanilla mode. The result in Figure 4 (d) indicates that *FlashAlloc* is not only beneficial to the calling tenant itself but also *altruistic* to neighbor tenants.

Though beneficial for all tenants, the effects of *FlashAlloc* are not uniform across tenants. To be specific, the TPC-C's TPS improve about by 1.9× while the DB Bench's OPS does only by 1.3×. This is because the WAF interference do more harms for the TPC-C benchmark in the vanilla mode.

Operation Latency High tail latency can negatively impact performance stability, making it challenging to meet Service-Level-Agreements of service providers. Although numerous studies [3, 5, 30] have presented solutions to reduce latency, garbage collection in SSDs can still hinder traffic and revenue. The root cause of latency spikes during garbage collection in the existing SSDs is that victim blocks have pages with different lifetimes, resulting in excessive copyback overhead. To evaluate the effect of *FlashAlloc*

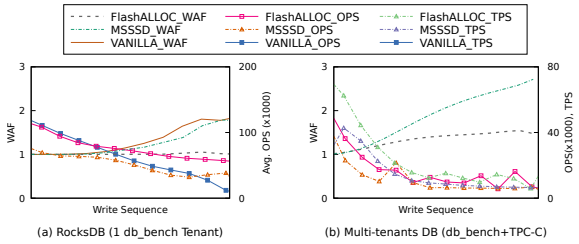

Figure 6: Comparison with MS-SSD

on the latency at the application layer, we measured the operation latency of db_bench while running the benchmark with the same configuration in Figure 4 (a) and presented in Table 2. *FlashAlloc* substantially reduces the overall operation latency and narrows the latency distribution. To be concrete, as shown in Table 2, the average, 99th, and 99.9th percentile latency of *FlashAlloc* are 32%, 12%, and 40% lower than those of Vanilla, respectively. In addition, to evaluate the effect of *FlashAlloc* on the read latency at the block I/O layer, we measured the average latency for all read requests to the storage by RocksDB using the `blktrace` utility and presented in the last column in Table 2. *FlashAlloc* can almost halve the average read latency at the block layer, compared to the Vanilla version. This result corroborates that the reduced WAF by *FlashAlloc* helps to complete the read and write operation faster than the Vanilla.

6.4 Quantitative Comparison with MS-SSD

A novel interface for flash storage, Multi-Stream SSD (MS-SSD in short), was proposed and standardized [24, 47], which allows applications to place pages with different lifetimes to different streams (*i.e.*, flash blocks). More specifically, when a write system call is invoked, applications can assign a proper *stream-id* to the data. Upon receiving this write command, MS-SSD will allocate the data to the corresponding physical stream. This interface performs effectively when correctly hinted by applications [24].

In that MS-SSD aims to reduce write amplification by streaming objects into different flash blocks [24], it is meaningful to quantitatively compare *FlashAlloc* with MS-SSD and discuss their differences. For this, we have prototyped the MS-SSD interface using the Cosmos+ board [3] and have measured throughput and WAF while running three workloads in *FlashAlloc* and MS-SSD modes: the TPC-C benchmark using MySQL, the db_bench using RocksDB (single tenant), and concurrent execution of the former two workloads (multi-tenancy), and present the results of the latter two experiments in Figure 6. For the db_bench experiment, the static stream-id assignment which manually maps stream-ids and physical stream of Cosmos+ board based on file type or level of SSTables is taken to lower the write amplification [10, 49]. Figure 6 omits TPC-C benchmark results, as MS-SSD's performance is almost identical to FlashAlloc, as shown in Figure 4(c). This is mainly because both schemes isolate the single DWB object from other data.

Meanwhile, in the case of single tenant db_bench(Figure 6 (a)), MS-SSD suffers from write amplification while *FlashAlloc* does not.

[3]The source codes of Cosmos+ Board firmware for MS-SSD are available at https://github.com/JonghyeokPark/Flashalloc-Cosmos/tree/master/Multistream

With the MS-SSD prototype supporting eight streams, as in commercial MS-SSDs [24], each stream has to be shared by multiple SSTables with differing lifetimes due to the higher number of SSTables in the benchmark compared to physical streams. SSTables, even when created simultaneously at the same level but with varying lifetimes, are often multiplexed onto the same flash blocks in MS-SSD due to their differing deletion times. To be Worse, MS-SSD, while using stream-id for writing, doesn't perform stream-aware GC, leading to pages from different streams mixing in the same flash block.

For these two reasons, MS-SSD suffers from device-level write amplification. In contrast, with *FlashAlloc*, each SSTable streams into distinct flash blocks, and its GC-By-Block-Type scheme prevents data pages from different FA instances mixing within the same block, eliminating write amplification in db_bench experiments. Figure 6 clearly shows the advantage of *FlashAlloc*'s fine-grained per-object streaming over MS-SSD when the object count exceeds the number of physical streams in MS-SSD. As expected due to MS-SSD's write amplification under db_bench, *FlashAlloc* surpasses MS-SSD in the multi-tenancy experiment as shown in Figure 6 (b). Specifically, *FlashAlloc* boosts TPC-C's TPS and db_bench's OPS by 1.7× and 1.4× , respectively, compared to MS-SSD.

7 RELATED WORK

In that *FlashAlloc* aims at reducing physical WAF by passing the host semantic to flash devices, three interfaces are closely related to it: Trim [42], Multi-Stream SSD [24] and ZNS-SSD [6].

Trim Even when a file is deleted, the old storage interface (*e.g.*, SATA) provides no mechanism to pass the host semantic about the file deletion and thus flash devices regard pages from the deleted file as still valid and unnecessarily relocate them during GC, causing write amplification. To address this, the `trim` command was proposed to inform flash devices that page(s) specified by a logical address range are no longer valid (*i.e.*, dead) at the host [42]; the trim-hinted device will no longer relocate those pages upon GC [4].

FlashAlloc and `trim` are common in that both explicitly provide flash storages with the host semantic to lower write amplification. In addition, they are *synergetic* to each other: when a file is *FlashAlloc*-ed, the `trim` command can complete simply by erasing all the *FlashAlloc*-ed blocks, instead of invalidating all pages individually [22]. Meanwhile, they differ in their invoke time: *FlashAlloc* is called at object creation while trim is at object deletion.

Multi-Stream SSD While both commonly aim at streaming writes to reduce write amplification, MS-SSD and *FlashAlloc* quite differ in their abstractions for write streaming. The MS-SSD interface has introduced the additional concept of stream-id and mandates applications to statically bind stream-id to each write call. The static binding of stream-id, combined with the limited number of physical streams available in commercial MS-SSDs (*e.g.*, 4 [24]), will raise several practical issues. First, it is a *non-transparent* abstraction in that every write call has to come with static stream-id. Next, it would be a non-trivial task for developers to estimate the number of physical streams for their applications and to correctly group numerous objects with different lifetimes into the limited streams. Third, the static stream-id assignment is *non-adaptive*. As the lifetimes of objects can change over time, programmers need scrutinize those statistics and periodically re-assign stream-ids to objects. Fourth, the effect of write streaming could diminish due to the *stream-id conflict* in the multi-tenant environment [29]. Lastly and most importantly, as shown in Section 6.4, MS-SSD still suffers from write amplification when the number of logical objects to be streamed exceeds the number of physical streams available in the SSD. This situation is mostly the case in real applications.

On the other hand, *FlashAlloc*'s *per-object write streaming* abstraction offers benefits. It enables *transparent write streaming* by only needing the logical address range of each object during creation, resulting in minimal or no application changes. Additionally, its fine-grained per-object streaming removes the need for developers to manage stream-ids or worry about stream-id conflicts in multi-tenant environments.

ZNS-SSD As another way to address the write amplification problem in the conventional SSDs, a new interface, ZNS (Zoned Name Space) is recently proposed [6], which exposes *zones* (a set of logical blocks) to the host as the unit of data management. ZNS is an advanced interface of Open-Channel SSD [7] and LightNVM [8] and is thus reviving the existing zoned echo system using open-source storage management software [35, 36, 45].

Though novel and effective in several use cases, however, the interface imposes the strict write-ordering rule: all writes to zones should be to be sequential and also in their LBA order. Data-center vendors criticize that the strict write rule of ZNS can hinder the innovation [1]. Such strict write rule has two drawbacks. First, software stacks should adapt to the sequential write ordering, which is unlikely to be accepted in industry [1]. Second, while exempting from the block interface tax, ZNS can instead introduce yet-more-expensive tax of log-structured writes (*e.g.*, compaction in RocksDB). Such operations can induce logical WAF of more than ten [13, 48]. In contrast, *FlashAlloc* requires neither any write rule nor the log structured write, enabling transparent write streaming.

8 CONCLUSION

Existing flash devices are *object-oblivious* in handling writes and thus allow to colocate data from different objects in the same flash block. To remedy such write multiplexing problem, we proposed a novel interface, *FlashAlloc*, which enables flash devices to stream writes by logical objects into different physical flash blocks, thus *object-aware* in handling writes and minimizing write amplification.

We prototyped *FlashAlloc* on a real SSD board by extending its FTL firmware and also modified a set of software stacks so as to use *FlashAlloc*. Experimental results confirmed that *FlashAlloc* can enable popular data stores to reduce write amplification and can also mitigate the WAF interference among multiple tenants.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No.2022R1A2C2008225, No.RS-2023-00251665), Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1802-07, Hankuk University of Foreign Studies Research Fund of 2023, and Samsung Electronics. We thank the anonymous PVLDB reviewers for their valuable comments.

REFERENCES

[1] 2022 OCP Global Summit 2022. PANEL: Flexible Data Placement using NVM Express - Implementation Perspective. https://www.opencompute.org/events/past-events/2022-ocp-global-summit.

[2] Daniel Abadi et al. 2020. The Seattle Report on Database Research. *SIGMOD Record* 48, 4 (Feb. 2020).

[3] Mijin An, In-Yeong Song, Yong-Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1404–1417. https://doi.org/10.1145/3514221.3526126

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, USA.

[5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores.. In *USENIX Annual Technical Conference*. 753–766.

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703.

[7] Matias Bjørling, Javier González, and Philippe Bonnet. [n.d.]. LightNVM: The Linux Open-Channel SSD Subsystem.

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 359–374. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[9] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage* 12, 3, Article Article 13 (May 2016).

[10] Changho Choi. 2016. Increasing SSD Performance and Lifetime with Multi-stream Technology. SNIA Data Storage Innovation, San Mateo, CA.

[11] Soyee Choi, Dong Hyun Kang, Sang-Won Lee, and Young Ik Eom. 2020. Concurrent Sequential Writes Also Considered Harmful in Solid State Drives (Poster paper). In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. USENIX Association.

[12] Soyee Choi, Hyunwoo Park, and Sang Won Lee. 2018. Don't write all data pages in one stream. In *Proceedings of the 21th International Conference on Extending Database Technology (EDBT 2018)*. 654–657.

[13] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of VLDB Endowment* 14, 3 (Nov. 2020), 364–377.

[14] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of VLDB Endowment* 14, 3 (Nov. 2020), 364–377.

[15] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research (CIDR) 2017)*.

[16] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies, FAST 2021*. USENIX Association, 33–49.

[17] FaceBook 2021. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[18] Fio: Flexible I/O tester [n.d.]. Fio: Flexible I/O tester. https://github.com/axboe/fio.

[19] Google. 2020. LevelDB. https://github.com/google/leveldb.

[20] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 229–240.

[21] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 127–144.

[22] Choulseung Hyun, Jongmoo Choi, Donghee Lee†, and Sam H. No. 2011. To TRIM or Not to TRIM: Judicious TRIMing for Solid State Drive. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*.

[23] Myoungsoo Jung and Mahmut Kandemir. 2012. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*.

[24] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.

[25] Minji Kang, Soyee Choi, Gihwan Oh, and Sang Won Lee. 2020. 2R: Efficiently Isolating Cold Pages in Flash Storages. *Proceedings of VLDB Endowment* 13, 11 (2020), 2004–2017.

[26] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Transactions on Storage* 16, 3, Article 15 (July 2020), 35 pages.

[27] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 273–286.

[28] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/170035.170042

[29] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *PVLDB* 12, 12 (2019), 1942–1945.

[30] Chen Luo and Michael J. Carey. 2020. On Performance Stability in LSM-Based Storage Systems. *Proc. VLDB Endow.* 13, 4 (jan 2020), 449–462. https://doi.org/10.14778/3372716.3372719

[31] Ma, Dongzhe and Feng, Jianhua and Li, Guoliang. 2014. A Survey of Address Translation Technologies for Flash Memories. *ACM Computing Survey* 46, 3, Article 36 (Jan. 2014), 36:1–36:39 pages.

[32] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of VLDB Endowment* 13, 12 (Aug. 2020).

[33] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Information* 33, 4 (June 1996), 351–385.

[35] OpenMPDK 2022. FlexAlloc. https://github.com/OpenMPDK/FlexAlloc.

[36] OpenMPDK 2022. xZTL: Zone Translation Layer User-space Library. https://github.com/OpenMPDK/xZTL.

[37] OPENSSD TEAM 2019. The OpenSSD Project. http://www.openssd.io.

[38] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*. 109–116.

[39] Percona-Lab 2008. tpcc-mysql benchmark. https://github.com/Percona-Lab/tpcc-mysql.

[40] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 26–52.

[41] Sangwook Shane Hahn and Sungjin Lee and Cheng Ji and Li-Pin Chang and Inhyuk Yee and Liang Shi and Chun Jason Xue and Jihong Kim. 2017. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 759–771.

[42] Frank Shu. 2007. Notification of Deleted Data Proposal for ATA-ACS2. http://t13.org.

[43] smartmontools. 2021. Smartmontools Documentation. https://www.smartmontools.org/wiki/TocDoc#Tutorials.

[44] Sungjin Lee and Ming Liu and Sangwoo Jun and Shuotao Xu and Jihong Kim and Arvind. 2016. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 339–353.

[45] westerndigitalcorporation 2022. xZTL: Zone Translation Layer User-space Library. https://github.com/westerndigitalcorporation/zenfs.

[46] Wikepedia. 2019. File System Fragmentation. https://en.wikipedia.org/wiki/File_system_fragmentation.

[47] William Martin(T10 Technical Editor). 2015. SCSI Block Commands - 4 (SBC-4) (Working Draft Revision 9): 4.34 Stream Control. http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc4r09.pdf. , 110–112 pages.

[48] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. USENIX Association.

[49] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual Stream Management for Multi-streamed SSDs. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/yong