Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage

Qiuping Wang, *The Chinese University of Hong Kong and Alibaba Group;*
Jinhong Li, and Patrick P. C. Lee, *The Chinese University of Hong Kong;*
Tao Ouyang, Chao Shi, and Lilong Huang, *Alibaba Group*

https://www.usenix.org/conference/fast22/presentation/wang

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage

Qiuping Wang[1,2], Jinhong Li[1], Patrick P. C. Lee[1], Tao Ouyang[2], Chao Shi[2], Lilong Huang[2]
[1]*The Chinese University of Hong Kong* [2]*Alibaba Group*

Abstract

Log-structured storage has been widely deployed in various domains of storage systems, yet its garbage collection incurs write amplification (WA) due to the rewrites of live data. We show that there exists an optimal data placement scheme that minimizes WA using the future knowledge of block invalidation time (BIT) of each written block, yet it is infeasible to realize in practice. We propose a novel data placement algorithm for reducing WA, SepBIT, that aims to infer the BITs of written blocks from storage workloads and separately place the blocks into groups with similar estimated BITs. We show via both mathematical and production trace analyses that SepBIT effectively infers the BITs by leveraging the write skewness property in practical storage workloads. Trace analysis and prototype experiments show that SepBIT reduces WA and improves I/O throughput, respectively, compared with state-of-the-art data placement schemes. SepBIT is currently deployed to support the log-structured block storage management at Alibaba Cloud.

1 Introduction

Modern storage systems adopt the *log-structured* design [30] for high performance. Examples include flash-based solid-state drives (SSDs) [5,10], file systems [15,21,27,30,32], key-value stores [25,28], table stores [9], storage management [6], in-memory storage [31], RAID arrays [18], and cloud block services [40]. Log-structured storage transforms random write requests into sequential disk writes in an append-only log, so as to reduce disk seek overhead and improve write performance. It also brings various advantages in addition to high write performance, such as improved flash endurance in SSDs [21], unified abstraction for building distributed applications [6,9], efficient memory management in in-memory storage [31], and load balancing in cloud block storage [40]. Recent advances in zoned storage [4,7] also advocate the adoption of log-structured storage based on append-only interfaces for scalable performance.

The log-structured design writes live data blocks to the append-only log without modifying existing data blocks in-place, so it regularly performs *garbage collection (GC)* to reclaim the free space of stale blocks. GC works by reading a segment of blocks, removing any stale blocks, and writing back the remaining live blocks. The repeated writes of live blocks lead to *write amplification (WA)*. They not only incur I/O interference to foreground workloads [18], but also lead to reduced flash lifespans and unnecessary power consumption in data centers.

Mitigating WA in log-structured storage has been a well-studied topic in the literature (§5). In particular, a large body of studies focuses on designing *data placement* strategies by properly placing blocks in separate groups. He *et al.* [16] point out that a data placement scheme should group blocks by the *block invalidation time (BIT)* (i.e., the time when a block is invalidated by a live block; a.k.a. the death time [16]) to achieve the minimum WA. However, without obtaining the future knowledge of the BIT pattern, how to design an optimal data placement scheme with the minimum WA remains an unexplored issue. Existing temperature-based data placement schemes that group blocks by block temperatures (e.g., write/update frequencies) [12,20,27,33,35,42,43] are arguably inaccurate to capture the BIT pattern and fail to effectively group the blocks with similar BITs [16].

We propose SepBIT, a novel data placement scheme that aims for the minimum WA in log-structured storage. It infers the BITs of written blocks from the underlying storage workloads and separately places the written blocks into different groups, each of which stores the blocks with similar *estimated* BITs. Specifically, it builds on the *skewed* write patterns observed in the real-world cloud block storage workloads (e.g., Alibaba Cloud [23] and Tencent Cloud [46]). It separates the written blocks into *user-written blocks* and *GC-rewritten blocks* (defined in §2.1). It further separates each set of user-written blocks and GC-rewritten blocks by inferring the BIT of each block, so as to perform fine-grained separation of blocks into groups with similar estimated BITs. We summarize our contributions below:

- We first design an ideal data placement strategy that has the minimum WA in log-structured storage, based on the (impractical) assumption of having the future knowledge of BITs of written blocks. Our analysis not only motivates how to design a practical data placement scheme that aims to group the written blocks with similar BITs, but also provides an oracular baseline for our comparisons.
- We design SepBIT, which performs fine-grained separation of written blocks by inferring their BITs from the underlying storage workloads. We show via both mathematical and trace analyses that our BIT inference is effective in skewed workloads. SepBIT also achieves low memory overhead in its indexing structure for tracking block statistics.

- We evaluate SepBIT using real-world cloud block storage workloads at Alibaba Cloud [23] and Tencent Cloud [46]. Trace analysis on both workloads shows that SepBIT has the lowest WA compared with eight state-of-the-art data placement schemes. For example, for the Alibaba Cloud traces, SepBIT reduces the overall WA by 8.6-15.9% and 9.1-20.2% when the Greedy [30] and Cost-Benefit [30,31] algorithms are used for segment selection in GC, respectively. It also reduces the per-volume WA by up to 44.1%, compared with merely separating user-written and GC-rewritten blocks in data placement.
- We prototype a log-structured storage system that supports different data placement schemes and runs on an emulated zoned storage backend based on ZenFS [3]. Our prototype experiments show that SepBIT improves I/O throughput over most volumes due to its efficient WA reduction; for example, its median throughput is 20% higher than the second best data placement scheme.

SepBIT is currently deployed at Alibaba Cloud Enhanced SSDs (ESSDs) [1], which provide cloud block storage services for end-users or applications. Each ESSD is a block-level volume (or virtual disk) backed by flash-based SSD storage, and aims to support low-latency (e.g., around 100μs) and high-throughput (e.g., up to 1 M IOPS) I/O access. ESSDs are deployed atop Pangu [29], a general distributed storage platform that provides an append-only write interface. To be compatible with the append-only write interface of Pangu, ESSDs adopt the log-structured design and are abstracted as log-structured storage in our paper.

Our trace analysis scripts and prototype are open-sourced at `http://adslab.cse.cuhk.edu.hk/software/sepbit`.

2 Background and Motivation

2.1 GC in Log-Structured Storage

We consider a log-structured storage system that comprises multiple *volumes*, each of which is assigned to a user. Each volume is configured with a capacity of tens to hundreds of GiB and manages data in an append-only manner. It is further divided into *segments* that are configured with a maximum size (e.g., tens to hundreds of MiB). Each segment contains fixed-size *blocks*, each of which is identified by a *logical block address (LBA)* and has a size (e.g., several KiB) that aligns with the underlying disk drives. Each block, either from a new write or from an update to an existing block, is appended to a segment (called an *open* segment) that has not yet reached its maximum size. If a segment reaches its maximum size, the segment (called a *sealed* segment) becomes immutable. Updating an existing block is done in an *out-of-place* manner, in which the latest version of the block is appended to an open segment and becomes a *valid* block, while the old version of the block is invalidated and becomes an *invalid* block.

Log-structured storage needs to regularly reclaim the space occupied by the invalid blocks via GC. A variety of GC policies can be realized, yet we can abstract a GC policy as a three-phase procedure:

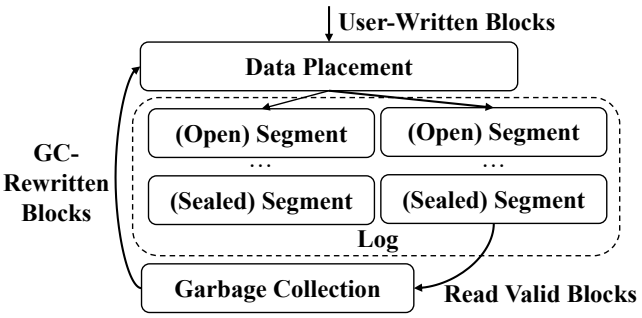


Figure 1: The workflow of a general data placement scheme.

- *Triggering*, which decides when a GC operation should be activated. In this work, we assume that a GC operation is triggered for a volume when its *garbage proportion (GP)* (i.e., the fraction of invalid blocks among all valid and invalid blocks) exceeds a pre-defined threshold (e.g., 15%).
- *Selection*, which selects one or multiple sealed segments for GC. In this work, we focus on two selection algorithms: (i) Greedy [30], which selects the sealed segments with the highest GPs, and (ii) Cost-Benefit [30,31], which selects the sealed segments that have the highest values $\frac{GP*age}{1-GP}$ (where *age* refers to the elapsed time of a sealed segment since it is sealed) for GC.
- *Rewriting*, which discards all invalid blocks from the selected sealed segments and writes back the remaining valid blocks into one or multiple open segments. The space of the selected sealed segments can then be reused.

A log-structured storage system sees two types of written blocks: each request that writes or updates an LBA in the workload generates one *user-written block* (i.e., a new block) and zero or more *GC-rewritten blocks* that are due to the rewrites of the block during GC. Thus, GC incurs *write amplification (WA)*, defined as the ratio of the total number of both user-written blocks and GC-rewritten blocks to the number of user-written blocks. In the deployment at Alibaba Cloud ESSDs (§1), we observe that the high WA from GC degrades both the effective I/O bandwidth and the SSD lifespans. It is thus critical to minimize WA.

In this work, we aim to design a general and lightweight data placement scheme that mitigates the WA due to GC in cloud-scale deployment. Figure 1 shows the workflow of a general data placement scheme, which separates all written blocks (i.e., user-written blocks and GC-rewritten blocks) into different groups and writes the blocks to the open segments of the respective groups. The data placement scheme is compatible with any GC policy (i.e., independent of the triggering, selection, and rewriting policies).

2.2 Ideal Data Placement

We present an ideal data placement scheme that minimizes WA (i.e., WA=1). We also elaborate why it is infeasible to

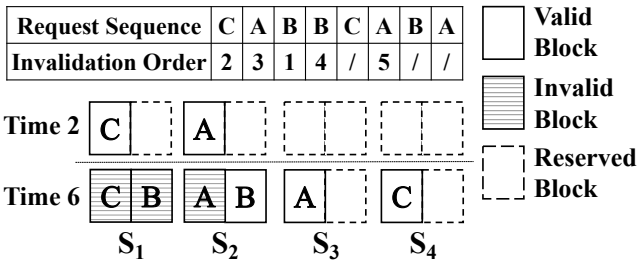


Figure 2: Example of the ideal data placement scheme.

realize in practice, so as to motivate the design of an effective practical data placement scheme.

System model. We first define the notations. Consider a write-only request sequence of blocks that are written to a log-structured storage system. Let m be the number of user-written blocks in the request sequence and s be the segment size (in units of blocks). Let $k = \lceil \frac{m}{s} \rceil$ be the number of sealed segments in the system, and let $S_1, S_2, \cdots, S_k$ denote the corresponding k sealed segments. Let o_i (where $o_i \geq 1$) be the *invalidation order* of the i-th block in the request sequence based on the BITs of all blocks (where $1 \leq i \leq m$), meaning that the i-th block is the o_i-th invalidated block among all invalid blocks.

Placement design. For the ideal placement scheme, we make the following assumptions. Suppose that the system has the future knowledge of the BITs of all blocks, and hence the invalidation order o_i of the i-th block in the request sequence (where $1 \leq i \leq m$). It also allocates k open segments for storing incoming blocks, and performs a GC operation whenever there are s invalid blocks in the system (i.e., one segment size of invalid blocks).

The system writes the i-th block to the $\lceil \frac{o_i}{s} \rceil$-th open segment. If the j-th (where $1 \leq j \leq k$) open segment is full, it is sealed into the sealed segment S_j. Thus, S_j stores the blocks with the invalidation orders in the range of $[(j-1) \cdot s+1, j \cdot s]$. The first GC operation is triggered when there exist s invalid blocks; according to the placement, all such blocks must be stored in S_1. Thus, the first GC operation will choose S_1 for GC, and there will be no rewrites as all blocks in S_1 must be invalid. In general, the j-th GC operation (where $1 \leq j \leq k$) will choose S_j for GC, and there will be no rewrites as S_j contains only invalid blocks.

Figure 2 depicts an example of the ideal data placement scheme. Consider a write-only request sequence with $m = 8$ blocks with three LBAs A, B, and C, and the i-th block is written at time i (where $1 \leq i \leq m$). We fix the segment size as $s = 2$. We show the status of the volume at time 2 and time 6 when the second block and the sixth block are written, respectively. At time 2, we have appended C to S_1 and A to S_2, as their invalidation orders are 2 and 3, respectively. Note that all blocks in S_1 become invalid when block C is updated at time 5, and at this time we can perform a GC operation to reclaim the free space occupied by S_1. Note that the GC operation does not incur any rewrite. Later, at time 6, the system appends A to S_3 since its invalidation order is 5.

Limitations and lessons learned. While the ideal data placement scheme achieves the minimum WA, there exist two practical limitations. First, the scheme needs to have future knowledge of the BIT of every block to assign the blocks to the corresponding open segments, but having such future knowledge is infeasible in practice. Second, the scheme needs to provision $k = \lceil m/s \rceil$ open segments to hold all m blocks in the request sequence in the worst case, as well as k corresponding sealed segments for keeping the blocks from the k open segments. Such provisioning incurring high memory and storage costs as m increases. Also, having too many open and sealed segments incurs substantial random writes that lead to performance slowdown.

A practical data placement scheme should address the above two limitations. Without the future knowledge of BITs, it should effectively *infer* the BIT of each written block. With only a limited number of available open segments, it should group written blocks by *similar BITs* instead of placing them in strict invalidation order. Our goal is to address the limitations driven by real-world cloud block storage workloads.

2.3 Trace Overview

We consider the *public* block-level I/O traces from two cloud block storage systems, Alibaba Cloud [23] and Tencent Cloud [46]. The Alibaba Cloud traces contain I/O requests (in multiples of 4 KiB blocks) from 1,000 virtual disks, referred to as *volumes*, over a one-month period in January 2020. The Tencent Cloud traces have 4,995 volumes over a nine-day period in October 2018. In this paper, we mainly focus on the Alibaba Cloud traces, while we verified that the Tencent Cloud traces show similar findings [39].

The Alibaba Cloud traces comprise a variety of workloads (e.g., virtual desktops, web services, key-value stores, and relational databases), and hence are representative to drive our analysis. We treat each volume in the traces as a standalone volume in the log-structured storage system (§2.1), such that each volume performs data placement and GC independently. Our goal is to mitigate the overall WA across all volumes.

We pre-process the traces for our analysis and evaluation as follows. We only consider write requests as they are the only contributors of WA. Since some volumes in the traces have limited write requests to trigger sufficient GC operations, we remove such volumes to avoid biasing our analysis. Specifically, we focus on the volumes with sufficient write requests: each volume has a write working set size (WSS) (i.e., the number of unique LBAs being written multiplied by the block size) above 10 GiB and a total write traffic size (i.e., the number of written bytes) above 2× its write WSS. To this end, we select 186 volumes from the Alibaba Cloud traces, which account for a total of over 90% of write traffic of all 1,000 volumes. The 186 volumes contain 10.9 billion write requests, 410.2 TiB of written data (with 390.2 TiB of

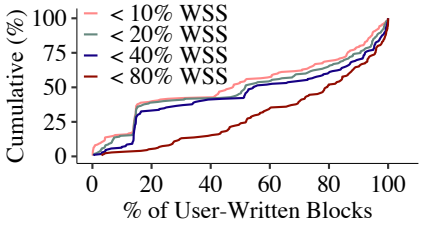


Figure 3: Percentages of user-written blocks with different short lifespans.

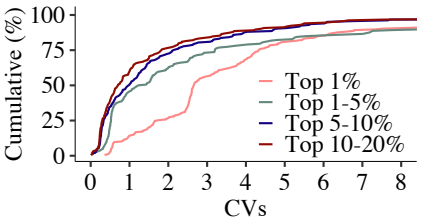


Figure 4: CVs of the lifespans of frequently updated blocks.

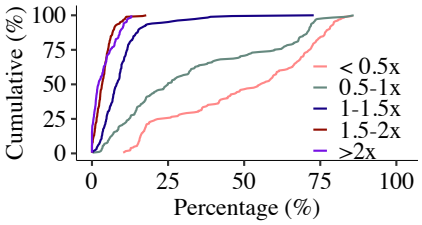


Figure 5: Percentages of rarely updated blocks with different lifespans.

updates), 20.3 TiB of write WSS (with 17.2 TiB of update WSS). Each of the 186 volumes has a write WSS ranging from 10 GiB to 1 TiB and a write traffic size ranging from 43 GiB to 36.2 TiB. Since the WSS varies across volumes, we configure the maximum storage space of each volume as $\frac{WSS}{1-GPT}$, where GPT denotes the GP threshold to trigger GC.

2.4 Motivation

We show via trace analysis that existing data placement schemes cannot accurately capture the BIT pattern and group the blocks with similar BITs for effective WA mitigation. We consider the 186 selected volumes from the Alibaba Cloud traces (§2.3). We define the lifespan of a block as the number of bytes written by the workload from when a block is written until it is invalidated (or until the end of the trace). A block is invalidated when the workload updates the same LBA. We make three key observations.

Observation 1: User-written blocks generally have short lifespans. We say that a block has a *short lifespan* if its lifespan is smaller than the write working set size (WSS) (i.e., the number of unique written LBAs multiplied by the 4 KiB block size). We examine the percentages of user-written blocks that fall into different lifespan range groups with short lifespans that are represented as the fractions of the write WSS for each volume. Figure 3 shows the cumulative distributions of the percentages of user-written blocks across all volumes in different lifespan groups. In a large fraction of volumes, their user-written blocks tend to have short lifespans. For example, half of the volumes have more than 79.5% of user-written blocks with lifespans smaller than 80% of their write WSSes, and have more than 47.6% of user-written blocks with lifespans smaller than only 10% write WSS. In contrast, GC-rewritten blocks generally have long lifespans. By definition, GC-rewritten blocks are rewritten as they remain valid in the GC-reclaimed segments. In both Greedy and Cost-Benefit selection algorithms, GC tends to select segments that either show a high GP or exist for a long time, implying that GC-rewritten blocks tend to have long lifespans.

Our findings suggest that user-written blocks and GC-rewritten blocks can have vastly different BIT patterns, in which user-written blocks tend to have short lifespans, while GC-rewritten blocks tend to have long lifespans. Existing data placement schemes either mix user writes and GC writes [12, 20, 27, 35], or focus on user writes [33, 42, 43], in the data placement decisions. Failing to distinguish between user-written blocks and GC-rewritten blocks can lead to inefficient WA mitigation. Instead, it is critical to separately identify the BIT patterns of user-written blocks and GC-rewritten blocks.

Observation 2: Frequently updated blocks have highly varying lifespans. We investigate *frequently updated blocks*, referred to as the blocks whose *update frequencies* (i.e., the number of updates) rank in the top 20% in the write working set (i.e., the set of LBAs being written) of a volume. Specifically, for each volume, we divide the frequently updated blocks into four groups based on their ranks of update frequencies, namely top 1%, top 1-5%, top 5-10%, and top 10-20%, so that the blocks in each group have similar update frequencies. The medians of the minimum update frequency in the four groups across all volumes are 37.5, 8.5, 6.0, and 5.0, respectively. To avoid evaluation bias, we exclude the blocks that have not been invalidated before the end of the traces. For each group of a volume, we calculate the *coefficient of variation (CV)* (i.e., the standard deviation divided by the mean) of the lifespans of the blocks; a high CV (e.g., larger than one) implies a high variance in the lifespans.

Figure 4 shows the cumulative distributions of CVs across all volumes (note that 6, 6, 20, and 18 volumes in the four groups have CVs exceeding 8, respectively). We see that frequently updated blocks with similar update frequencies have high variance in their lifespans (and hence the BITs); for example, 25% of the volumes have their CVs exceeding 4.34, 3.20, 2.14, and 1.82 in the four groups top 1%, top 1-5%, top 5-10%, and top 10-20%, respectively. Our findings also suggest that existing temperature-based data placement schemes that group the blocks with similar write/update frequencies [12, 20, 27, 33, 35, 42, 43] cannot effectively group blocks with similar BITs, and hence the WA cannot be fully mitigated.

Observation 3: Rarely updated blocks dominate and have highly varying lifespans. We examine the write working set of each volume and define the *rarely updated blocks* as those that are updated no more than four times during the one-month trace period. We see that rarely updated blocks occupy a high percentage in the write working sets of a large fraction of volumes. In half of the volumes, more than 72.4% of their write working sets contain rarely updated blocks. We further examine the lifespans of those rarely updated blocks. For each volume, we divide the rarely updated blocks into

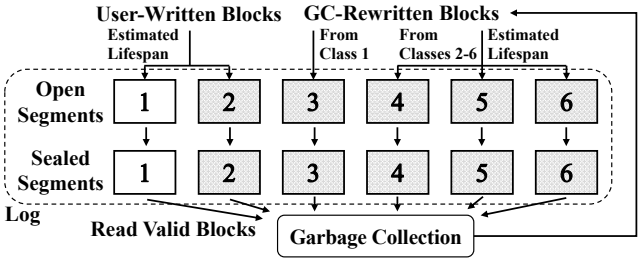


Figure 6: SepBIT workflow.

five groups that are partitioned by the lifespans of 0.5×, 1×, 1.5×, and 2× of their write WSSes. We then calculate the percentage of those blocks that fall into each group.

Figure 5 shows the cumulative distributions of the percentages of rarely updated blocks in different lifespan groups across all volumes. In 25% of the volumes, more than 71.5% of the rarely updated blocks have their lifespans smaller than 0.5× write WSS. For the remaining four groups, the medians of the percentages are 24.9%, 8.1%, 3.3%, and 2.2%, respectively. In other words, the lifespans of rarely updated blocks can span both short and long lifespan ranges, and hence show high deviations of BITs in a volume. As in Observation 2, our findings again suggest that existing temperature-based data placement schemes cannot effectively group the rarely updated blocks with similar BITs. Rarely updated blocks are often treated as cold blocks with low write frequencies, so they tend to be grouped together and separated from the hot blocks with high write frequencies. However, their vast differences in BIT patterns make temperature-based data placement schemes inefficient in mitigating WA.

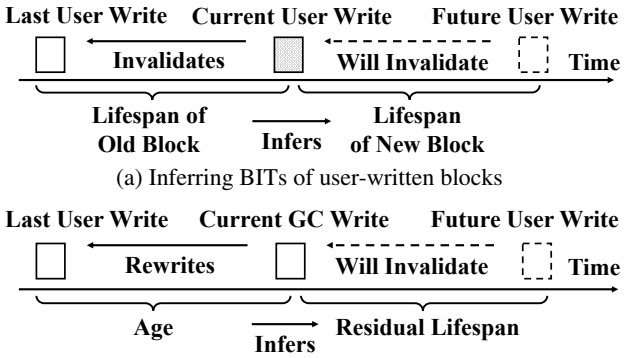
3 SepBIT Design

3.1 Design Overview

We design SepBIT based on our observations in §2.4. SepBIT first separates blocks into user-written blocks and GC-rewritten blocks due to their different BIT patterns (Observation 1). It further separates both user-written blocks and GC-rewritten blocks by inferring their BITs instead of using block temperatures as in existing temperature-based approaches (Observations 2 and 3).

Figure 6 depicts the workflow of SepBIT. Our current design of SepBIT defines *six* classes of segments, in which Classes 1-2 correspond to the segments of user-written blocks, while Classes 3-6 correspond to the segments of GC-rewritten blocks. Each class is now configured with one open segment and has multiple sealed segments. If an open segment reaches the maximum size, it is sealed and remains in the same class.

SepBIT infers the lifespans of blocks and in turn the corresponding BITs of blocks. For user-written blocks (i.e., Classes 1-2), SepBIT stores the *short-lived* blocks (with short lifespans) in Class 1 and the remaining *long-lived* blocks (with long lifespans) in Class 2. For GC-rewritten blocks (i.e., Classes 3-6), SepBIT appends the blocks from Class 1 that are rewritten by GC into Class 3, and groups the remaining



(a) Inferring BITs of user-written blocks

(b) Inferring BITs of GC-rewritten blocks

Figure 7: Ideas of inferring BITs in SepBIT.

GC-rewritten blocks into Classes 4-6 by similar BITs inferred.

The main idea of SepBIT is as follows. For each user-written block, SepBIT examines its *last user write time* to infer its lifespan. Specifically, for the write time, SepBIT uses a monotonic timer (instead of the real timestamp) that increments by one for each user-written block. If the user-written block is issued from a new write, SepBIT assumes that it has an infinite lifespan. Otherwise, if the user-written block updates an old block, SepBIT uses the lifespan of the old block (i.e., the number of user-written bytes in the whole workload since its last user write time until it is now invalidated) to estimate the lifespan of the user-written block, as shown in Figure 7(a). Our intuition is that *any user-written block that invalidates a short-lived block is also likely to be a short-lived block* (§3.2). Then if the short-lived blocks are written at about the same time, their corresponding BITs will be close, so SepBIT groups them into same class (i.e., Class 1). For the long-lived blocks (including the user-written blocks from new writes), SepBIT groups them into Class 2.

For each GC-rewritten block, SepBIT examines its *age*, defined as the number of user-written bytes in the whole workload since its last user write time until it is rewritten by GC, to infer its *residual lifespan*, defined as the number of user-written bytes since it is rewritten by GC until it is invalidated (or until the end of the traces), as shown in Figure 7(b). As a result, the lifespan of a GC-rewritten block is its age plus its residual lifespan. Our intuition is that *any GC-rewritten block with a smaller age has a higher probability to have a short residual lifespan* (§3.3), implying that GC-rewritten blocks with different ages are expected to have different residual lifespans. Thus, SepBIT can distinguish the blocks of different residual lifespans based on their ages and group the GC-rewritten blocks with similar ages into the same classes.

Our design builds on the assumption that the access pattern is *skewed* for inferring the BITs of blocks. We justify our assumption via the mathematical analysis for skewed distributions and the trace analysis for real-world workloads (§3.2 and §3.3). To adapt to changing workloads and GC policies, SepBIT monitors the workloads to separate user-written blocks and GC-rewritten blocks into different classes (§3.4).

3.2 Inferring BITs of User-Written Blocks

We show via both mathematical and trace analyses the effectiveness of SepBIT in estimating the BITs of user-written blocks based on the lifespans. Let n be the total number of unique LBAs in a working set; without loss of generality, each LBA is denoted by an integer from 1 to n. Let p_i (where $1 \le i \le n$) be the probability that LBA i is being written in each write request. Consider a write-only request sequence of blocks, each of which is associated with a sequence number b and the LBA A_b. Let b and b' (where $b' < b$) denote the sequence numbers of a new user-written block and the corresponding invalid old block, respectively (i.e., $A_b = A_{b'}$).

Recall from §3.1 that SepBIT estimates the lifespan (denoted by u) of the user-written block b using the lifespan (denoted by v) of the old block b', so the estimated BIT of block b is equal to the current user write time plus the estimated lifespan u; note that both u and v are expressed in units of blocks. We claim that if v is small, u is also likely to be small. To validate the claim, let u_0 and v_0 (both in units of blocks) be two thresholds. We then examine the conditional probability of $u \le u_0$ given the condition that $v \le v_0$ subject to a workload of different skewness. If the conditional probability is high for small u_0 and v_0, then our claim holds.

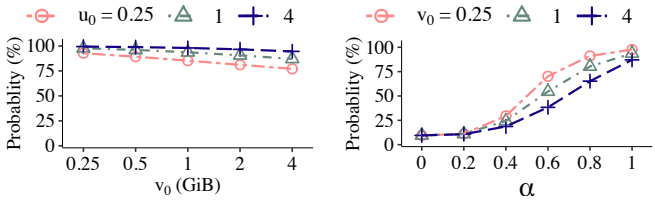
Mathematical analysis. We examine the following conditional probability (see derivation in our technical report [39]):

$$\Pr(u \le u_0 \mid v \le v_0) = \frac{\Pr(u \le u_0 \text{ and } v \le v_0)}{\Pr(v \le v_0)} = \frac{\sum_{i=1}^{n}(1-(1-p_i)^{u_0}) \cdot (1-(1-p_i)^{v_0}) \cdot p_i}{\sum_{i=1}^{n}(1-(1-p_i)^{v_0}) \cdot p_i}.$$

We analyze the conditional probability via the Zipf distribution, given by $p_i = (1/i^{\alpha})/\sum_{j=1}^{n}(1/j^{\alpha})$, where $1 \le i \le n$ for some skewness parameter $\alpha \ge 0$. A larger α implies a more skewed distribution. We fix $n = 10 \times 2^{18}$, which corresponds to a working set of 10 GiB with 4 KiB blocks. We then study how the conditional probability $\Pr(u \le u_0 \mid v \le v_0)$ varies across u_0, v_0, and α.

Figure 8(a) first shows the conditional probability for varying u_0 and v_0, where we fix $\alpha = 1$. We focus on short lifespans by varying u_0 and v_0 of up to 4 GiB, which is less than the write WSS (§2.4). Overall, the conditional probability is high for different u_0 and v_0; the lowest one is 77.1% for $v_0 = 4$ GiB and $u_0 = 0.25$ GiB. This shows that a user-written block is highly likely to have a short lifespan if its invalidated block also has a short lifespan. In particular, the conditional probability is higher if v_0 is smaller (i.e., the invalidated blocks have shorter lifespans), implying a more accurate estimation of the lifespan of the user-written block.

Figure 8(b) next shows the conditional probability for varying v_0 and α, where we fix $u_0 = 1$ GiB. Note that for $\alpha = 0$, the Zipf distribution reduces to a uniform distribution. Overall, the conditional probability increases with α (i.e., more skewed). For example, for $\alpha = 1$, the conditional probability



(a) $\alpha = 1$; varying u_0, v_0 (b) $u_0 = 1$ GiB; varying v_0, α

Figure 8: Inferring BITs of user-written blocks: $\Pr(u \le u_0 \mid v \le v_0)$ versus v_0 and α.

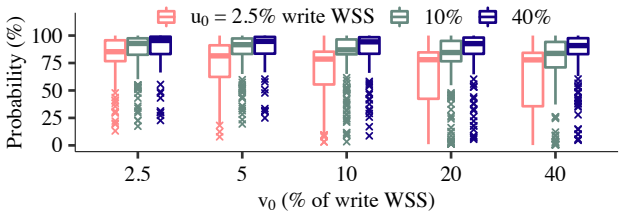


Figure 9: Inferring BITs of user-written blocks: Boxplots of $\Pr(u \le u_0 \mid v \le v_0)$ for different u_0 and v_0 in real-world workloads.

is at least 87.1%. However, for $\alpha = 0$, the conditional probability is only 9.5%. This indicates that the high accuracy of lifespan estimation only holds under skewed workloads.

Trace analysis. We use the block-level I/O traces from Alibaba Cloud (§2.3) to validate if the conditional probability remains high in real-world workloads. To compute the conditional probability, we first find the set of user-written blocks that invalidate old blocks with $v \le v_0$. Then the conditional probability is the fraction of blocks with $u \le u_0$ in the set. We vary both v_0 and u_0 as different percentages of the write WSS to examine different conditional probabilities. Figure 9 shows the boxplots of the conditional probabilities over all volumes for different u_0 and v_0. In general, the conditional probability remains high in most of the volumes. For example, for v_0 being 40% of write WSS, the medians of the conditional probabilities are in the range of 77.8-90.9%, and the 75th percentiles are in the range of 84.3-97.6%. Also, the conditional probability tends to be higher for a smaller v_0.

3.3 Inferring BITs of GC-Rewritten Blocks

We further show via both mathematical and trace analyses the effectiveness of SepBIT in estimating the BITs of GC-rewritten blocks based on the *residual* lifespans. Recall from §3.1 that SepBIT estimates the residual lifespan of a GC-rewritten block using its age, so the estimated BIT of the GC-rewritten block is equal to the current GC write time plus the estimated residual lifespan. However, characterizing directly GC-rewritten blocks is non-trivial, as it depends on the actual GC policy (e.g., when GC is triggered and which segments are selected for GC) (§2.1). Instead, we model GC-rewritten blocks based on user-written blocks. If a user-written block has a lifespan above a certain threshold, we assume that it is rewritten by GC and treat it as a GC-rewritten block with an age equal to the threshold. We can then apply a similar

analysis for user-written blocks as in §3.2.

We define the following notations. As each GC-rewritten block is a user-written block before being rewritten by GC, we identify each GC-rewritten block by its corresponding user-written block with sequence number b. Let u, g, and r be its lifespan, age, and residual lifespan, respectively, such that $u = g + r$; each of the variables is measured in units of blocks. We claim that r has a higher probability to be small with a smaller g. To validate the claim, let g_0 and r_0 (both in units of blocks) be the thresholds for the age and the residual lifespan, respectively. We examine the conditional probability of $u \leq g_0 + r_0$ given the condition that $u \geq g = g_0$ subject to a workload of different skewness. The conditional probability specifies the fraction of GC-rewritten blocks whose residual lifespans are shorter than r_0 among all GC-rewritten blocks with age g_0 (note that the GC-rewritten blocks are modeled as user-written blocks with lifespans above g_0). If the conditional probability is higher for a smaller g_0 subject to a fixed r_0, then our claim holds.

Mathematical analysis. We examine the following conditional probability (see derivation in our technical report [39]):

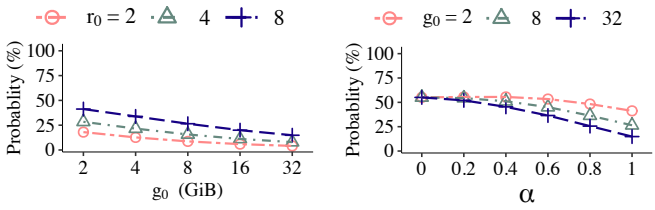
$$\Pr(u \leq g_0 + r_0 \mid u \geq g_0) = \frac{\Pr(g_0 \leq u \leq g_0 + r_0)}{\Pr(u \geq g_0)} = \frac{\sum_{i=1}^{n} p_i \cdot ((1-p_i)^{g_0} - (1-p_i)^{g_0+r_0})}{\sum_{i=1}^{n} p_i \cdot (1-p_i)^{g_0}}.$$

As in §3.2, we use the Zipf distribution and fix $n = 10 \times 2^{18}$ unique LBAs. We study how the conditional probability $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ varies across g_0, r_0, and α.

Figure 10(a) first shows the conditional probability for varying g_0 and r_0, where we fix $\alpha = 1$. We focus on a large value of g_0 of up to 32 GiB since we target long-lived blocks. We also vary r_0 up to 8 GiB. Overall, for a fixed r_0, the conditional probability decreases as g_0 increases. For example, given that $r_0 = 8$ GiB, the probability with $g_0 = 2$ GiB is 41.2%, while the probability for $g_0 = 32$ GiB drops to 14.9%. This validates our claim that GC-rewritten blocks with different ages are expected to have different residual lifespans. Thus, we can distinguish the GC-rewritten blocks of different residual lifespans based on their ages.

Figure 10(b) further shows the conditional probability for varying g_0 and α, where we fix $r_0 = 8$ GiB. For a small α, the conditional probability has a limited difference for varying v, while the difference becomes more significant as α increases. For example, for $\alpha = 0$ (i.e., the uniform distribution), there is no difference varying g_0; for $\alpha = 0.2$, the difference of the conditional probability between $g_0 = 2$ GiB and $g_0 = 32$ GiB is only 3.5%, while the difference for $\alpha = 1$ is 26.4%. This indicates that our claim holds under skewed workloads, and we can better distinguish the GC-rewritten blocks of different residual lifespans under more skewed workloads.

Trace analysis. We also use block-level I/O traces from Alibaba Cloud (§2.3) to examine the conditional probability in



(a) $\alpha = 1$; varying r_0, g_0 (b) $r_0 = 8$ GiB; varying g_0, α

Figure 10: Inferring BITs of GC-rewritten blocks: $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ versus g_0 and α.

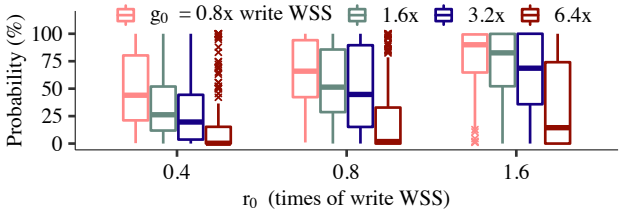


Figure 11: Inferring BITs of GC-rewritten blocks: Boxplots of $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ for different r_0 and g_0 in real-world workloads.

real-world workloads. We first identify the set of blocks with $u \geq g_0$ in the workload, and then compute the conditional probability as a fraction of blocks with $u \leq g_0 + r_0$ in the set. We vary both r_0 and g_0 as different percentages of the write WSS. Figure 11 depicts the boxplots of the conditional probabilities over all volumes for different g_0 and r_0. For a fixed r_0, the conditional probabilities have significant differences for varying g_0. For example, if we fix r_0 as 1.6× of write WSS and g_0 increases from 0.8× to 6.4× of write WSS, the median probabilities drop from 90.0% to 14.5%.

3.4 Implementation Details

Threshold selection. We assign blocks into different classes by their estimated BITs with multiple thresholds: for user-written blocks, we define a *lifespan threshold* for separating short-lived blocks and long-lived blocks; for GC-rewritten blocks, we need multiple *age thresholds* to separate them by ages (§3.1). We configure the thresholds via the *segment lifespan* of a segment, defined as the number of user-written bytes in the workload since the segment is created (i.e., the time when the first block is appended to the segment) until it is reclaimed by GC. Specifically, we monitor the average segment lifespan, denoted by ℓ, among a fixed number of recently reclaimed segments in Class 1. For each user-written block, if it invalidates an old block with a lifespan less than ℓ, we write it to Class 1; otherwise, we write it to Class 2. For GC-rewritten blocks, we set the age thresholds as multiples of ℓ (see below).

Algorithmic details. Algorithm 1 shows the pseudo-code of SepBIT, which consists of three functions: `GarbageCollect`, `UserWrite`, and `GCWrite`. Each class always corresponds to one open segment. If an open segment is full, it becomes a sealed segment, and SepBIT creates a new open segment

Algorithm 1 SepBIT

```
1: t = 0; ℓ = +∞; ℓ_tot = 0; n_c = 0, where t is the global timestamp
2: function GarbageCollect( )
3:     Select a segment S by selection algorithm
4:     if S is from Class 1 then
5:         n_c = n_c + 1, ℓ_tot = ℓ_tot + (t − S.creation_time)
6:     end if
7:     if n_c = 16 then
8:         ℓ = ℓ_tot/n_c; n_c = 0; ℓ_tot = 0
9:     end if
10:    for each valid block b in S do
11:        GCWrite(b)
12:    end for
13: end function
14: function UserWrite(b)
15:    Find lifespan v of the invalidated block b′ due to b
16:    if v < ℓ then
17:        Append b to open segment of Class 1
18:    else
19:        Append b to open segment of Class 2
20:    end if
21:    t = t + 1
22: end function
23: function GCWrite(b)
24:    if b is from Class 1 then
25:        Append b to open segment of Class 3
26:    else
27:        g = t − b.last_user_write_time
28:        If g ∈ [0, 4ℓ), append b to open segment of Class 4
29:        If g ∈ [4ℓ, 16ℓ), append b to open segment of Class 5
30:        If g ∈ [16ℓ, +∞), append b to open segment of Class 6
31:    end if
32: end function
```

within the same class. SepBIT initializes the average segment lifespan $\ell = +\infty$, which is updated on-the-fly. It also tracks a global timestamp t, which records the sequence number of the current user-written block.

`GarbageCollect` is triggered by a GC operation according to the GC policy (§2.1). It performs GC and monitors the runtime information of the reclaimed segments. It selects a segment S for GC based on the selection algorithm (e.g., Greedy or Cost-Benefit (§2.1)). It sums up the lifespans of collected segments from Class 1 as ℓ_{tot}, and computes the average lifespan $\ell = \ell_{tot}/n_c$ for every fixed number n_c (e.g., $n_c = 16$ in our current implementation) of reclaimed segments.

`UserWrite` processes each user-written block b. It first computes the lifespan v of the invalidated old block b'. If v is less than ℓ, `UserWrite` appends b (which is treated as a short-lived block) to the open segment of Class 1; otherwise, it appends b (which is treated as a long-lived block) to the open segment of Class 2.

`GCWrite` processes each GC-rewritten block that corresponds to some user-written block b. If b is originally stored in Class 1, `GCWrite` appends b to the open segment of Class 3; otherwise, `GCWrite` appends b to one of the open segments of Classes 4-6 based on the age of b. Currently, we configure the age thresholds as three ranges, $[0, 4\ell)$, $[4\ell, 16\ell)$, and $[16\ell, +\infty)$, for Classes 4-6, respectively, based on our evaluation findings. Nevertheless, we have also experimented with different numbers of classes and thresholds [39], and we observe only marginal differences in WA.

Memory usage. SepBIT only stores the last user write time of each block as the metadata alongside the block *on disk*, without maintaining a mapping from every LBA to its last user write time in memory. Putting metadata alongside a block is feasible, as SSDs typically associate a small spare region (e.g., of size 64 bytes) with each flash page for storing metadata. Specifically, for user-written blocks, SepBIT only needs to know whether the lifespan of an invalidated block is shorter than a threshold. It thus suffices for SepBIT to track only the recently written LBAs. In our current implementation (written in C++), SepBIT maintains a first-in-first-out (FIFO) queue to record recently written LBAs. It dynamically adjusts the queue length according to the value ℓ. If the FIFO queue is full, each insert of an element will dequeue one element from the queue. If ℓ increases, the FIFO queue allows more inserts without dequeueing any element; if ℓ decreases, the FIFO queue dequeues two elements for each insert until the number of elements drops below ℓ. If the LBA exists in the FIFO queue and its user write time is within the recent ℓ user writes, SepBIT writes it into Class 1. To efficiently query the FIFO queue, SepBIT creates a `std::map` structure in the C++ standard template library to record each unique LBA in the FIFO queue and its latest queue position. When we enqueue the LBA of a newly written block into the FIFO queue, we insert or update the LBA with its current queue position in the `std::map` structure; when we dequeue an LBA from the FIFO queue, we remove the LBA from the `std::map` structure if its recorded queue position is equal to the dequeued one.

For GC-rewritten blocks, SepBIT retrieves them during GC and examines the user write time directly from the metadata, so as to assign the GC-rewritten block to the corresponding class without any memory overhead incurred.

Prototype. We prototype a log-structured block storage system that realizes SepBIT and existing data placement schemes. We choose to deploy our prototype on zoned storage [4], whose append-only interfaces favor log-structured storage deployment. Specifically, our prototype runs on an emulated zoned storage backend based on ZenFS [3] (due to the lack of a real zoned storage device, we currently emulate the zoned storage backend using Intel Optane Persistent Memory [2]). Each segment in the prototype is a one-to-one mapping to a *ZoneFile*, the basic unit in the zoned storage backend in ZenFS. Then ZenFS stores ZoneFiles in different zones without incurring device-level GC. For the metadata and the FIFO queue in SepBIT, the prototype stores them in separate files and accesses them using `mmap` for memory efficiency; for other existing data placement schemes, the prototype stores

all metadata in memory. When the prototype triggers GC (at the system level), it reads only valid blocks from storage and rewrites the blocks into different segments.

The reasons of choosing emulated zoned storage based on ZenFS in our prototype are three-fold. First, zoned storage has a similar storage abstraction to Pangu (§1), as both of them support append-only writes and large-size append-only units (e.g., up to hundreds of MiB). Second, emulated zoned storage provides minimal external interference, making the performance evaluation reproducible; in contrast, the performance of traditional SSDs can be easily disturbed by device-level GC. Finally, ZenFS is a lightweight user-space zone-aware file system that readily supports zoned storage.

4 Evaluation

4.1 Data Placement Schemes

We compare SepBIT with eight existing temperature-based data placement schemes, namely Dynamic dAta Clustering (DAC) [12], SFS [27], MultiLog (ML) [35], extent-based identification (ETI) [33], MultiQueue (MQ) [42], Sequentiality, Frequency, and Recency (SFR) [42], Fading Average Data Classifier (FADaC) [20], and WARCIP [43]. Note that these existing schemes are mainly designed for mitigating the flash-level WA in SSDs, yet they are also applicable for general log-structured storage. Take DAC [12] as an example. DAC associates each LBA with a temperature-based counter (quantified based on the write count) and writes blocks to the segments of different temperature levels. Each user write promotes the LBA to a hotter segment while each GC write demotes the LBA to a colder segment. Other temperature-based data placement schemes follow the similar idea of DAC. Specifically, the above designs adopt different metrics to measure block temperatures, such as access frequencies (in ML [35], MQ [42], and ETI [33]), recency (in FADaC [20]), hotness (in SFS [27]), access counts (in DAC [12]), sequentiality (in SFR [42]), and update intervals (in WARCIP [43]).

We also consider three baseline strategies.

- **NoSep** appends any written blocks (either user-written blocks or GC-rewritten blocks) to the same open segment.
- **SepGC [37]** separates written blocks by user-written blocks and GC-rewritten blocks, and writes them into two different open segments.
- **Future knowledge (FK)** assumes that the BIT of each written block is known in advance. For a written block (either a user-written block or a GC-rewritten block), if its invalidation will occur within t bytes since the written time, we write the block to the $\lceil \frac{t}{s} \rceil$-th open segment, where s is the segment size (in bytes). Given the limited number of open segments, FK uses the last open segment to store all user-written blocks and GC-rewritten blocks if their BITs do not belong to the prior open segments. We annotate the lifespan of each block in the traces in advance, so that we can compute the BITs during evaluation.

Note that FK represents an *oracular* baseline that leverages future knowledge for placement decisions. It is identical to the ideal scheme (§2.2) if there are unlimited memory and storage budgets. Otherwise, with limited memory and storage budgets, it applies future knowledge to group a subset of blocks in a limited number of segments, and applies trivial data placement for the remaining blocks. Thus, FK represents both the ideal data placement scheme that has no memory and storage constraints and the trivial data placement scheme with the memory and storage constraints; the latter serves the baseline in our experiments.

By default, we configure six classes (each containing one open segment) for data placement for all schemes, except for NoSep, SepGC, and ETI. For NoSep, we configure one class for all written blocks; for SepGC, we configure two classes, one for user-written blocks and one for GC-rewritten blocks; for ETI, we configure two classes for user-written blocks and one class for GC-rewritten blocks. For MQ, SFR, and WARCIP, as they focus on separating user-written blocks only, we configure five classes for user-written blocks and the remaining class for GC-rewritten blocks. For DAC, SFS, ML, FADaC, and FK, since they do not differentiate user-written blocks and GC-rewritten blocks, we let them use all six classes for all written blocks. We adopt the default settings as described in the original papers of the existing schemes.

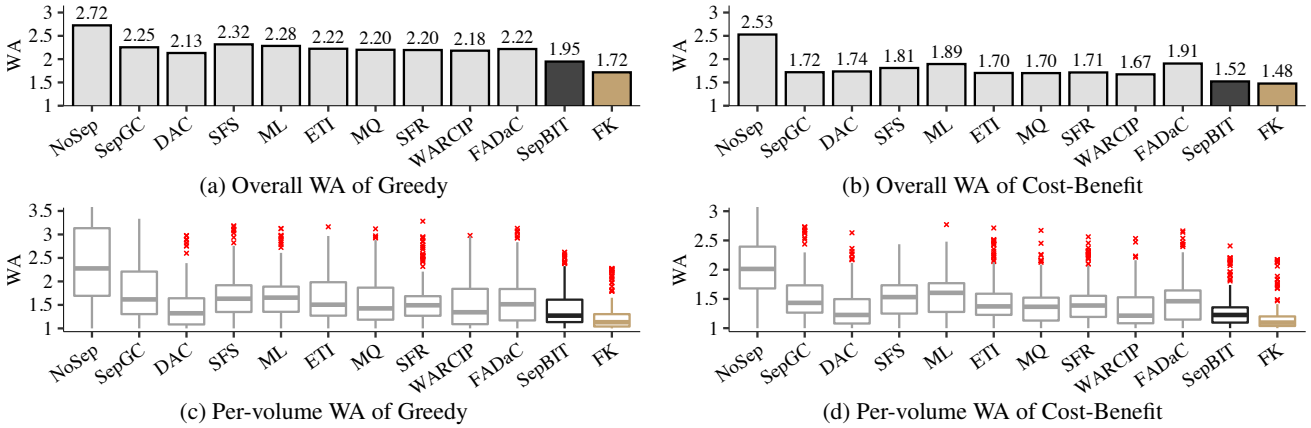
4.2 Results

Summary of findings. Our major findings include:

- SepBIT achieves the lowest WA among all data placement schemes (except FK) for different segment selection algorithms (Exp#1), different segment sizes (Exp#2), and different GP thresholds (Exp#3).
- We show that SepBIT provides accurate BIT inference (Exp#4).
- We provide a breakdown analysis on SepBIT, and show that it achieves a low WA by separating each set of user-written blocks and GC-rewritten blocks independently (Exp#5).
- SepBIT achieves the lowest WA in the Tencent Cloud traces (Exp#6).
- SepBIT shows high WA reduction for highly skewed workloads (Exp#7).
- We provide a memory overhead analysis and show that SepBIT achieves low memory overhead for a majority of the volumes (Exp#8).
- Our prototype evaluation shows that SepBIT achieves the highest throughput in a majority of the volumes (Exp#9).

Default configuration. Our default GC policy uses Cost-Benefit [30, 31] for segment selection and fixes the segment size and the GP threshold for triggering GC as 512 MiB and 15%, respectively; in Exp#1-Exp#3, we vary each of the configurations for evaluation. For real-world workloads, we use the Alibaba Cloud traces except for Exp#5.

Exp#1 (Impact of segment selection). We compare SepBIT with existing data placement schemes using Greedy [30] and



(a) Overall WA of Greedy

(b) Overall WA of Cost-Benefit

(c) Per-volume WA of Greedy

(d) Per-volume WA of Cost-Benefit

Figure 12: Exp#1 (Impact of segment selection).

Cost-Benefit [30, 31] for segment selection in GC (§2.1).

Figures 12(a) and 12(b) depict the overall WA across all 186 volumes under Greedy and Cost-Benefit, respectively. With separation in data placement, SepBIT reduces the overall WA of NoSep by 28.5% and 39.8% under Greedy and Cost-Benefit, respectively. More importantly, SepBIT achieves the lowest WA compared with all existing data placement schemes (except FK). It reduces the overall WA of SepGC and the eight state-of-the-art data placement schemes (i.e., excluding NoSep and FK) by 8.6-15.9% and 9.1-20.2% under Greedy and Cost-Benefit, respectively. Compared with FK, the overall WA of SepBIT is 13.5% and 3.1% higher under Greedy and Cost-Benefit, respectively. In short, SepBIT is highly efficient in WA mitigation under real-world workloads. Note that some data placement schemes even show a higher WA than SepGC, which performs simple separation of user-written blocks and GC-rewritten blocks, mainly because they fail to effectively group blocks with similar BITs (§2.4).

Figures 12(c) and 12(d) show the boxplots of per-volume WAs over all 186 volumes under Greedy and Cost-Benefit, respectively (we omit outliers of NoSep with very high WAs). SepBIT has the lowest 75th percentiles (1.61 and 1.36) among all existing data placement schemes (except FK) under Greedy and Cost-Benefit, while the second lowest one is DAC (1.64 and 1.50), respectively. This shows that SepBIT effectively reduces WAs in individual volumes with diverse workloads. In particular, Cost-Benefit is more effective in the WA reduction of SepBIT than Greedy, as the gap of the 75th percentiles between SepBIT and the second lowest one increases from 1.8% in Greedy to 9.4% in Cost-Benefit. Compared with FK, for 75th percentiles, SepBIT has 23.6% and 12.9% higher WA under Greedy and Cost-Benefit, respectively.

Exp#2 (Impact of segment sizes). We vary the segment size from 64 MiB to 512 MiB. For fair comparisons, we fix the amount of data (both valid and invalid data) to be retrieved in each GC operation as 512 MiB, meaning that a GC operation collects eight, four, two, and one segment(s) for the segment sizes of 64 MiB, 128 MiB, 256 MiB, and 512 MiB, respectively. We focus on comparing NoSep, SepGC, WARCIP, SepBIT, and FK, as they show the lowest WAs among existing data placement for various segment sizes. We present the complete results in our technical report [39].

Figures 13 depicts the overall WA versus the segment size. Overall, using a smaller segment size yields a lower WA, as a GC operation can perform more fine-grained selection of segments for more efficient space reclamation. Again, SepBIT achieves the lowest WA compared with all existing data placement schemes; for example, its WAs are 5.5%, 8.2%, and 10.0% lower than WARCIP for the segment sizes of 64 MiB, 128 MiB, and 256 MiB, respectively. Interestingly, SepBIT even has a lower WA (by 3.9-5.7%) than FK when the segment size is in the range of 64 MiB to 256 MiB. The reason is that FK currently groups blocks of close BITs in five open segments, while the last open segment stores all blocks (we now configure six classes in total) (§4.1). If the segment size is smaller, FK can only group fewer blocks in the limited number of open segments, so it becomes less effective of grouping blocks of close BITs.

Exp#3 (Impact of GP thresholds). We vary the GP thresholds from 10% to 25%. We again focus on comparing the overall WAs of NoSep, SepGC, WARCIP, SepBIT, and FK as in Exp#2. Figure 14 shows the overall WA versus the GP threshold. A larger GP threshold has a lower WA in general, as it is easier for a GC operation to select segments with high GPs. SepBIT still shows the lowest WA. It has 5.0-13.8% lower WAs than WARCIP for different GP thresholds. Compared with FK, SepBIT has comparable WAs with differences smaller than 1.8%, for different GP thresholds.

Exp#4 (BIT inference analysis). We study the effectiveness of the BIT inference in SepBIT. Note that SepBIT does not explicitly compute the estimated BIT of a block, but instead assigns blocks into classes corresponding to different ranges of estimated BITs (§3.4). To examine the effectiveness of BIT inference, our intuition is that each valid block that is rewritten during GC indicates that we incorrectly infer its BIT and places it into an incorrect segment. Thus, we can examine the GP of each collected segment to estimate the inference

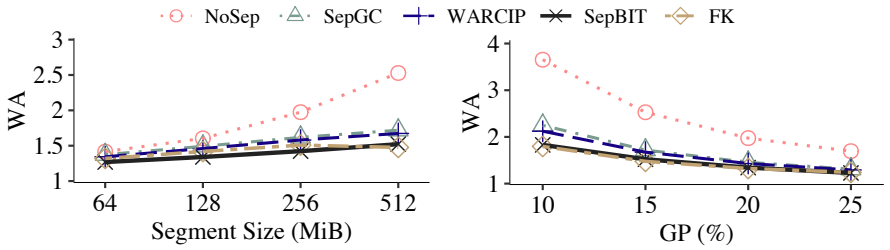


Figure 13: Exp#2 (Impact of segment sizes).

Figure 14: Exp#3 (Impact of GP thresholds).

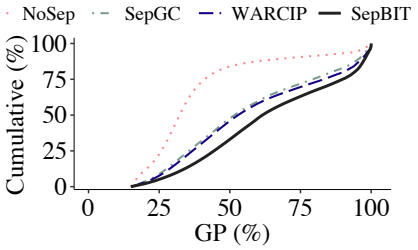
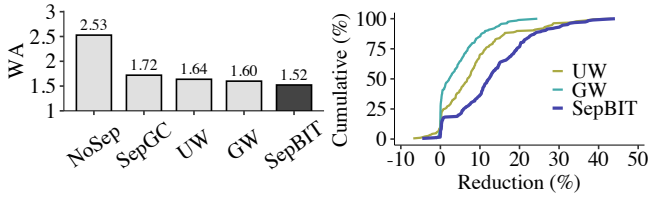


Figure 15: Exp#4 (BIT inference analysis).



(a) Overall WA (b) Per-volume WA reduction

Figure 16: Exp#5 (Breakdown analysis).

accuracy, such that a higher GP implies more accurate inference. We use the Cost-Benefit selection algorithm and fix the segment size and GP for triggering GC as 512 MiB and 15%, respectively. We study NoSep, SepGC, WARCIP, and SepBIT (WARCIP has the second lowest WA). We aggregate the collected segments during GC for all 186 volumes.

Figure 15 depicts the cumulative distributions of collected segments across different GPs for different schemes. The median GPs of the collected segments for NoSep, SepGC, WARCIP, and SepBIT are 32.3%, 51.6%, 52.9%, and 61.5%, respectively. SepBIT has the highest GPs, implying that it also has the highest accuracy in inferring BITs. WARCIP only shows a slightly higher GP of the collected segments than SepGC, so its WA reduction over SepGC is marginal.

Exp#5 (Breakdown analysis). We analyze how different components of SepBIT contribute to WA reduction. Recall that SepBIT separates written blocks into the user-written blocks and GC-rewritten blocks, and further separates each set of user-written blocks and GC-rewritten blocks independently. In our analysis, we consider NoSep (i.e., without separation), SepGC (i.e., separating written blocks into the user-written blocks and GC-rewritten blocks), and two variants:

- *UW*: It further separates user-written blocks based on SepGC, but without separating GC-rewritten blocks. It maintains three classes: Classes 1 and 2 store short-lived blocks and long-lived blocks as in SepBIT, respectively, while Class 3 stores all GC-rewritten blocks.
- *GW*: It further separates GC-rewritten blocks based on SepGC, but without separating user-written blocks. It maintains four classes: Class 1 stores all user-written blocks, and Classes 2-4 store GC-rewritten blocks as in Classes 4-6 of SepBIT.

Figure 16(a) shows the overall WAs of different data placement schemes. UW and GW reduce WA by 35.2% and 36.7% compared with NoSep, respectively; they also reduce WA by 4.8% and 7.0% compared with SepGC, respectively. The findings show that more fine-grained separation of each set of user-written blocks and GC-rewritten blocks brings further WA reduction. Also, SepBIT reduces WA by 7.0% and 4.9% compared with UW and GW, respectively, meaning that SepBIT can combine the benefits of UW and GW.

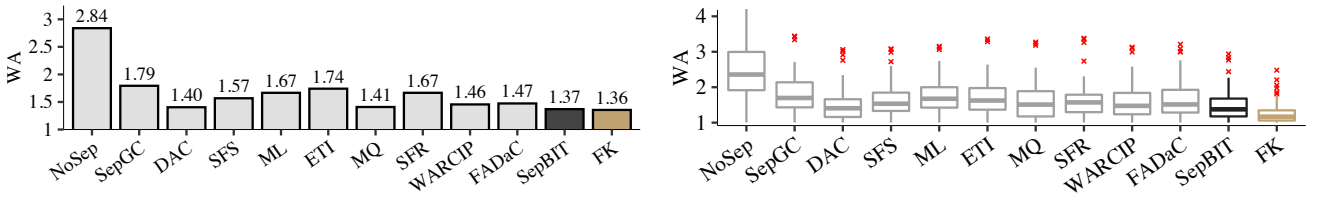
Figure 16(b) further shows the cumulative distributions of the WA reductions of UW, GW, and SepBIT compared with SepGC across all volumes. UW, GW, and SepBIT can reduce the WA of most of the volumes. The 75th percentiles of reductions of UW and GW are 11.4% and 6.9%, respectively, and their highest WA reductions are 43.3% and 24.5%, respectively. By combining UW and GW, the 75th percentile of the WA reductions of SepBIT compared with SepGC improves to 19.3% with the highest WA reduction as 44.1%.

Exp#6 (Results on the Tencent Cloud traces). We validate the effectiveness of SepBIT on the Tencent Cloud traces [46]. We pre-process the traces the same as for the Alibaba Cloud traces (§2.3) and select 271 out of 4,995 volumes. We run all the schemes as in Exp#1, using Cost-Benefit for segment selection and fixing the segment size and the GP threshold as 512 MiB and 15%, respectively.

Figure 17 depicts the overall WA and the per-volume WA across all 271 volumes. Among all existing data placement schemes, SepBIT achieves the lowest overall WA. Its overall WA is 2.5-21.3% lower than those of the eight existing schemes and 1.1% higher than that of FK. Compared with the second lowest scheme DAC, SepBIT has similar 50th and 75th percentiles of per-volume WA, and reduces the 90th percentile of per-volume WA from 2.09 to 1.97.

Exp#7 (Impact of workload skewness). We study how SepBIT works in workloads of different skewness. We set the selection algorithm as Greedy instead of Cost-Benefit, since Cost-Benefit also leverages the workload skewness to reduce WA and we want to exclude its impact from our analysis.

We inspect the skewness of each volume in the Alibaba Cloud traces, and analyze the correlation between the per-volume skewness and the WA reduction percentage of SepBIT over NoSep. We also present the results for synthetic workloads in our technical report [39]. Since not all real-world workloads have good fitness to a Zipf distribution [45], we describe the per-volume skewness according to how write traffic aggregates in the most frequently updated blocks. Specifically,



(a) Overall WA

(b) Per-volume WA

Figure 17: Exp#6 (Results on the Tencent Cloud traces).

Skewness α	0	0.2	0.4	0.6	0.8	1
Pct. (%)	20	27.6	38.1	52.4	71.1	89.5

Table 1: The percentage of write traffic over top-20% blocks in Zipf workloads of different skewness.

we compute the percentage of aggregated write traffic over the top 20% frequently written blocks. To show the relationship between the percentage of aggregated writes and the skewness factor of the Zipf distribution, Table 1 shows the percentage of write traffic over the top 20% frequently written blocks and the corresponding skewness factor α; note that the numbers are generated using 10 GiB of write WSS.

Figure 18 shows the results. Each point represents one volume. The x-axis is the percentage of aggregated write traffic over top 20% frequently written blocks and the y-axis is the WA reduction of SepBIT over NoSep. We see a positive correlation between the percentage of aggregated write traffic and the WA reduction (we also find that the p-value is smaller than 0.01 for the Pearson correlation coefficient 0.75, meaning that the positive correlation is statistically significant). For the volumes with percentages of aggregated write traffic larger than 80%, SepBIT reduces the WA by at least 38.0% with a maximum reduction of 76.7%.

Exp#8 (Memory overhead analysis). We analyze the memory overhead of SepBIT using the Alibaba Cloud traces. Recall that SepBIT tracks only the unique LBAs inside the FIFO queue (§3.4), instead of maintaining the mappings for all LBAs in the write working set. We report the memory overhead reduction of SepBIT as one minus the ratio of the number of unique LBAs in the FIFO queue to the number of unique LBAs in the write working set. To quantify the reduction, for each volume, we collect all values of the number of unique LBAs in the FIFO queue at runtime when ℓ (§3.4) is updated. To avoid bias due to the cold start of trace replay, for each volume, we exclude the beginning 10% of the values. We also collect the number of unique LBAs at the end of the traces. We consider two cases, namely (i) the worst case and (ii) the snapshot case. In the worst case, we use the maximum number of unique LBAs in the FIFO queue for all volumes; it assumes that each volume has its peak number of unique LBAs in the FIFO queue and incurs the most memory. In the snapshot case, we use the number of unique LBAs at the end of the traces, representing a snapshot of the system status.

From our analysis, we find that in the worst case, SepBIT reduces the overall memory overhead by 44.8%, while in the snapshot case, SepBIT reduces the overall memory overhead by 71.8%. To calculate the actual memory overhead, suppose that the mapping for each LBA has 8 bytes, in which both the LBA and the FIFO position are of size 4 bytes each (a 4-byte LBA can represent an address space of $2^{32} \times 2^{12} =$ 16 TiB for 4-KiB blocks). Since the aggregated write WSS of the 186 volumes is 20.3 TiB (§2.3), SepBIT reduces the overall memory overhead from $20.3 \cdot \frac{2^{40}}{2^{12}} \cdot 8 = 41.6$ GiB to $41.6 \cdot (1 - 71.8\%) = 11.7$ GiB.

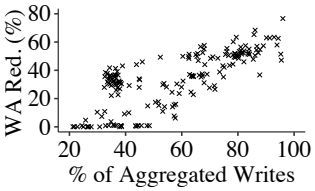


Figure 18: Exp#7 (Impact of workload skewness).

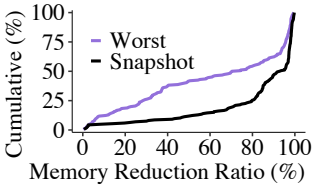
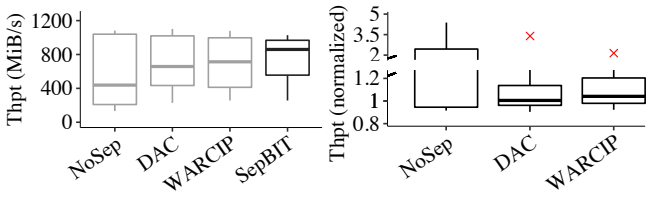


Figure 19: Exp#8 (Memory overhead).

Figure 19 further depicts the cumulative distributions of the memory overhead reductions across volumes under both the worst case and the snapshot case. In the worst case, SepBIT reduces the memory overhead by more than 72.3% in half of the volumes and the highest memory overhead reduction is 99.5%; in the snapshot case, the median reduction is 93.1% with the highest reduction as 99.7%. In the snapshot case, the 25th, 50th, and 75th percentiles of the number of unique LBAs across volumes are 99 K, 1,063 K, and 6,190 K, respectively, while the 25th, 50th, 75th percentiles of the number of total LBAs in the FIFO queue across volumes are 398 K, 2,242 K, and 8,857 K, respectively. The reason of the differences among volumes is their different degrees of skewness. The volumes with higher skewness see more aggregated traffic patterns, and hence the number of recently updated LBAs is much smaller compared with the write WSS.

Exp#9 (Prototype evaluation). We deploy our log-structured block storage system prototype (§3.4) on a machine equipped with an Intel Xeon Silver 4215 CPU, 96 GiB DDR4 RAM, and 4×128 GiB Intel Optane Persistent Memory modules. The machine runs Ubuntu 20.04.2 LTS with kernel 5.4.0.

Due to the limited storage capacity in our testbed machine, we focus on 20 volumes whose write traffic ranks the top 31-50 among the 186 volumes in the Alibaba Cloud traces. Their write traffic ranges from 0.82 TiB to 2.82 TiB, and their WAs under NoSep range from 1.00 to 4.96. Specifically, 9 volumes have their WAs less than 1.1, while 7 volumes have



(a) Absolute throughput (b) Normalized throughput

Figure 20: Exp#9 (Prototype evaluation).

their WAs greater than 3.0.

Also, our evaluation rate-limits user writes while GC is running due to the capacity constraint. The reason is that a GC operation removes the invalid blocks only after rewriting all valid blocks. If we issue user writes at full speed while GC is running, the storage space may run out. Thus, we limit the rate of user writes as 40 MiB/s while GC is running; otherwise, we issue user writes at full speed. We measure the *write throughput* (i.e., the number of user-written bytes divided by the total time for replaying each volume).

We compare SepBIT with NoSep, DAC, and WARCIP, based on our previous experiments that DAC and WARCIP perform the best among existing schemes and NoSep serves as the baseline. We configure the segment selection algorithm, the segment size, and the GP threshold as Cost-Benefit, 512 MiB, and 15%, respectively.

Figures 20(a) and 20(b) show the boxplots of the absolute write throughput and the normalized write throughput of SepBIT (w.r.t. NoSep, DAC, and WARCIP) in individual volumes for different schemes, respectively. SepBIT achieves the highest throughput for the 25th and 50th percentiles, at 556.1 MiB/s and 859.4 MiB/s, which are 28.3% and 20.4% higher than the second best, respectively.

For the 75th percentile, the absolute throughput of SepBIT is 6.9%, 5.2%, and 3.0% lower than those of NoSep, DAC, and WARCIP, respectively (Figure 20(a)). The reason is that such volumes (with top-25% throughput) have low WAs (less than 1.1) and hence are less affected by GC. Compared with other schemes, SepBIT spends extra time to access the FIFO queue (§3.4) and has slightly degraded throughput.

5 Related Work

GC in SSDs. We evaluated several existing data placement schemes (§4.1) for mitigating the WA of flash-level GC in SSDs. Other data placement schemes build on the use of program contexts [19] or the prediction of block temperature based on neural networks [44]. Some empirical studies evaluate the data placement algorithms on an SSD platform [22], or characterize how real-world I/O workloads affect GC performance [41]. In particular, Yadgar *et al.* [41] also investigate the impact of the number of separated classes in data placement based on the temperature-based data scheme MultiLog [35]. In contrast, SepBIT builds on the BIT for data placement, backed by the empirical studies from real-world I/O traces. ML-DT [8] uses neural networks to predict the block death time. Compared with ML-DT, SepBIT infers BITs only with the last user write time in a simpler manner.

Besides data placement, existing studies propose segment selection algorithms to reduce the WA of flash-level GC. In addition to Greedy and Cost-Benefit (§2.1), Cost-Age-Times [11] considers the cleaning cost, data age, and flash erasure counts in segment selection. Windowed Greedy [17], Random-Greedy [24], and d-choices [36] are variants of Greedy in segment selection. Desnoyers [14] models the WA of different segment selection algorithms and hot-cold data separation. SepBIT can work in conjunction with those algorithms.

GC in file systems. Several studies examine the GC performance for log-structured file systems. Matthew *et al.* [26] improve the GC performance by adapting GC to the system and workload behaviors. SFS [27] separates blocks by hotness (i.e., write frequency divided by age). Some studies reduce WA using file system semantics in data placement; for example, WOLF [38] groups blocks by files or directories, while hFS [47] and F2FS [21] separate data and metadata. Extending SepBIT with file system awareness is a future work.

GC for RAID and distributed storage. Some studies address the GC performance issues in RAID and distributed storage, such as reducing the WA of Log-RAID systems [13] and mitigating the interference between GC and user writes via GC scheduling in RAID arrays [19, 34]. RAMCloud [31] targets persistent distributed in-memory storage. It proposes two-level cleaning to maximize memory utilization by coordinating GC operations in memory and disk backends. It also corrects the original Cost-Benefit algorithm [30] for accurate segment selection. Our work focuses on data placement for WA mitigation and is orthogonal to those studies.

6 Conclusion

We propose SepBIT, a novel data placement scheme that mitigates WA caused by GC in log-structured storage by grouping blocks with similar estimated BITs. Inspired from the ideal data placement that minimizes WA (i.e., WA=1) using future knowledge of BITs, SepBIT leverages the skewed write patterns of real-world workloads to infer BITs. It separates written blocks into user-written blocks and GC-rewritten blocks and performs fine-grained separation in each set of user-written blocks and GC-rewritten blocks. To group blocks with similar BITs, it infers the BITs of user-written blocks and GC-rewritten blocks by estimating their lifespans and residual lifespans, respectively. Evaluation on production traces shows that SepBIT achieves the lowest WA compared with eight state-of-the-art data placement schemes.

Acknowledgements. We thank our shepherd, Keith Smith, and the anonymous reviewers for their comments. This work was supported in part by Alibaba Group via the Alibaba Innovation Research (AIR) program. The corresponding author is Patrick P. C. Lee.

References

[1] Alibaba Cloud ESSDs. https://www.alibabacloud.com/help/doc-detail/122389.htm.

[2] Intel Optane Persistent Memory 128GB Module. https://ark.intel.com/content/www/us/en/ark/products/190348/intel-optane-persistent-memory-128gb-module.html.

[3] ZenFS. https://github.com/westerndigitalcorporation/zenfs.

[4] Zoned storage. https://zonedstorage.io/.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX FAST*, 2008.

[6] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Trans. on Computer Systems*, 31(4):10, 2013.

[7] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *Proc. of USENIX ATC*, 2021.

[8] C. Chakraborttii and H. Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proc. of ACM SYSTOR*, 2021.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI*, 2006.

[10] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of ACM SIGMETRICS*, Jun 2009.

[11] M. Chiang and R. Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Softwares*, 48(3):213–231, 1999.

[12] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[13] T.-c. Chiueh, W. Tsao, H.-C. Sun, T.-F. Chien, A.-N. Chang, and C.-D. Chen. Software orchestrated flash array. In *Proc. of ACM SYSTOR*, 2014.

[14] P. Desnoyers. Analytic models of SSD write performance. *ACM Trans. on Storage*, 10(2):1–25, 2014.

[15] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proc. of ACM SOSP*, 1993.

[16] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proc. of ACM EuroSys*, 2017.

[17] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proc. of ACM SYSTOR*, 2009.

[18] J. Kim, K. Lim, Y. Jung, S. Lee, C. Min, and S. H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *Proc. of USENIX ATC*, 2019.

[19] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proc. of USENIX FAST*, 2019.

[20] K. Kremer and A. Brinkmann. FADaC: A self-adapting data classifier for flash memory. In *Proc. of ACM SYSTOR*, 2019.

[21] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proc. of USENIX FAST*, 2015.

[22] J. Lee and J.-S. Kim. An empirical study of hot/cold data separation policies in solid state drives (SSDs). In *Proc. of ACM SYSTOR*, 2013.

[23] J. Li, Q. Wang, P. P. C. Lee, and C. Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *Proc. of IEEE IISWC*, 2020.

[24] Y. Li, P. P. C. Lee, and J. C. S. Lui. Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization. In *Proc. of ACM SIGMETRICS*, 2013.

[25] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, 2016.

[26] J. N. Matthews, D. S. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, 1997.

[27] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proc. of USENIX FAST*, 2012.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[29] Z. Pang, Q. Lu, S. Chen, R. Wang, Y. Xu, and J. Wu. ArkDB: A key-value engine for scalable cloud storage services. In *Proc. of ACM SIGMOD*, 2021.

[30] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, 1992.

[31] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *Proc. of USENIX FAST*, 2014.

[32] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX ATC*, 1993.

[33] M. Shafaei, P. Desnoyers, and J. Fitzpatrick. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *Proc. of USENIX HotStorage*, 2016.

[34] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *Proc. of USENIX FAST*, 2013.

[35] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. of the VLDB Endowment*, 6(9):733–744, 2013.

[36] B. Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[37] B. Van Houdt. On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs. *Performance Evaluation*, 82:1–14, 2014.

[38] J. Wang and Y. Hu. WOLF - A novel reordering write buffer to boost the performance of log-structured file systems. In *Proc. of USENIX FAST*, 2002.

[39] Q. Wang, J. Li, P. P. C. Lee, T. Ouyang, C. Shi, and L. Huang. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. Technical report, The Chinese University of Hong Kong, 2022. `https://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_sepbit.pdf`.

[40] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu. Lessons and actions: What we learned from 10k SSD-related storage system failures. In *Proc. of USENIX ATC*, 2019.

[41] G. Yadgar, M. Gabel, S. Jaffer, and B. Schroeder. SSD-based workload characteristics and their performance implications. In *ACM Trans. on Storage*, 2021.

[42] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proc. of ACM SYSTOR*, 2017.

[43] J. Yang, S. Pei, and Q. Yang. WARCIP: Write amplification reduction by clustering I/O pages. In *Proc. of ACM SYSTOR*, 2019.

[44] P. Yang, N. Xue, Y. Zhang, Y. Zhou, L. Sun, W. Chen, Z. Chen, W. Xia, J. Li, and K. Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *Proc. of USENIX HotStorage*, 2019.

[45] Y. Yang and J. Zhu. Write skew and Zipf distribution: Evidence and implications. *ACM Trans. on Storage*, 12(4):1–19, 2016.

[46] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *Proc. of USENIX ATC*, 2020.

[47] Z. Zhang and K. Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys*, 2007.