Preprint – The final publication is available at *https://ieeexplore.ieee.org/abstract/document/9605386*

Constant Time Garbage Collection in SSDs

Reza Salkhordeh*, Kevin Kremer*, Lars Nagel†, Dennis Maisenbacher*, Hans Holmberg‡, Matias Bjørling‡, André Brinkmann*

* *Johannes Gutenberg University Mainz, Germany*
† *Loughborough University, UK*
‡ *Western Digital, Inc*
{rsalkhor, k.kremer, nagell, brinkman}@uni-mainz.de, dmaisenb@students.uni-mainz.de,
{hans.holmberg, matias.bjorling}@wdc.com

Abstract

The Flash Translation Layer (FTL) plays a crucial role for the performance and lifetime of SSDs. It has been difficult to evaluate different FTL strategies in real SSDs in the past, as the FTL has been deeply embedded into the SSD hardware. Recent host-based FTL architectures like ZNS now enable researchers to implement and evaluate new FTL strategies. In this paper, we evaluate the overhead of various garbage collection strategies using a host-side FTL, and show their performance limitations when scaling the SSD size or the number of outstanding requests. To address these limitations, we propose *constant cost-benefit* policy, which removes the scalability limitations of previous policies and can be efficiently deployed on host-based architectures. The experimental results show that our proposed policy significantly reduces the CPU overhead while having a comparable write amplification compared to the best previous policies.

***Index Terms*—SSD, FTL, Garbage collection, ZNS**

I. Introduction

SSDs have replaced magnetic disks in many applications areas, as they provide high IOPS per cost at a low power consumption [1], [2]. Vendors have been able to scale the capacity of SSDs to tens of TBs of storage at a performance above a million read IOPS. Internally, an SSD stores its data inside a set of flash cells placed within a flash block. Each flash block should be written sequentially, has to be erased before being rewritten, and only has a limited endurance.

SSDs implement a *Flash Translation Layer (FTL)* that manages these requirements and exposes a conventional random access block-interface to the host. The FTL transforms incoming random writes into a *log-structured* write pattern. As part of the process, the FTL has to employ a *garbage collection* (GC) strategy to free up blocks for new writes. Each garbage collection cycle first selects a victim block, copies valid data from the victim block to a new write block, and then erases the block. The ratio of total writes (by GC and user) to the user writes is called the *Write Amplification Factor (WAF)* [3]. A perfect SSD with no GC overhead has a WAF equal to 1, whereas common workloads exhibit WAFs of 2–4 [4]. The required processing power and memory capacity within the FTL have to scale for all known GC strategies with the performance and the capacity of the SSD [5].

A new type of SSDs, called Zoned Namespace (ZNS) SSDs [6], [7], enables the partitioning of responsibilities between the host and the SSD through the ZNS storage interface [8]. The interface defines a zoned storage model for SSDs, in which a set of zones is exposed. Similar to the characteristics of the flash media, the zones must be written sequentially and reset to be rewritten. This change allows a ZNS SSD to directly map host writes to its flash media, which eliminates the overhead of the conventional GC process in the SSD and also the internal WAF overhead. The ZNS SSD still maintains the responsibility of managing media reliability and durability within the SSD.

This paper studies the impact of transitioning the responsibility of fine-grained data placement within flash blocks to the host and its impact on host-side FTLs [9], [10] with respect to the garbage collection policies. Previous GC policies have been implemented either in simulators [11], [12], [13] or custom-built SSDs [14], [15], [16], since commercial FTLs have been deeply embedded inside SSDs and cannot be changed by third parties. The actual overheads of GC policies in real-world enterprise SSDs have been rarely exposed. We, for the first time, evaluate the overhead of GC policies for ZNS SSDs. Our evaluation reveals that the existing GC policies suffer from high processing or memory overheads if the size or the performance of SSDs are scaled. Existing GC strategies therefore

cannot provide the required performance for future enterprise applications. ZNS SSDs provide the ability to utilize the existing GC algorithms within file-systems or applications [6], avoiding the SSDs GC overhead that amplifies host-side GC. However, when considering host-side FTLs, that expose a conventional random write interface on top of ZNS SSDs, the overheads of existing GC policies continue to exist due to having to perform victim selection and maintaining in-memory data structures. We show that selecting the victim block itself does not have to be the performance bottleneck of GC policies in large and high-performance SSDs. Instead, the maintenance cost of the data structures, which requires updating per I/O, is significantly higher than that of the victim selection.

Based on our observations, we propose a constant-time GC policy, called *constant cost-benefit* (CCB), on top of the *cost-benefit* (CB) strategy [15]. It provides identical WA compared to CB, while having constant maintenance overhead when scaling SSDs in both size and performance. CCB maintains several linked-lists and places blocks, based on their number of invalid pages, in different linked-lists. For each I/O, CCB removes an item from a linked-list and adds it to the tail of another linked-list, both in constant time. Selecting the victim is also performed in constant time by comparing the heads of all lists.

Our experimental results show that our proposed GC policy offers up to 74% less CPU overhead, compared to the best competing strategy.

The main contributions of this paper are as follows:

- For the first time, we have measured the performance and scalability of GC policies applicable to ZNS SSDs.
- We quantify the performance overhead of different parts of GC policies and show that per I/O overhead can be significantly higher than that of the victim selection.
- We propose CCB GC policy, which has constant time overhead concerning SSD size, similar to the greedy policy, while maintaining WA efficiency of state-of-the-art policies.

The paper is structured as follows: Section II presents the background and motivation of this paper. Section III presents our data structure as well as our modified selection policy. Section IV describes the evaluation environment and presents the results. Finally, Section V concludes the paper.

II. Background and motivation

This section first explains the background of flash memory and ZNS SSD internal architecture. Afterward, we present various garbage collection policies and their shortcomings, which motivate this paper.

A. Flash Memory and SSDs

Flash memory cells employed in SSDs have many unique characteristics. Flash cells, e.g., cannot be overwritten, unless being erased first. Flash memory is therefore partitioned into pages (set of flash cells) and blocks, where pages are the unit of reads and writes, while blocks are the unit of erase operations. Each block typically contains many hundred or even thousands of pages. In terms of performance, the write latency of pages is several times higher than their read latency, while the erase latency of a block is even higher than its write latency.

Providing a block interface for SSDs which behaves semantically similar to a magnetic disk requires a *flash translation layer* (FTL). SSDs employ a log-based approach to write data pages sequentially, while the FTL manages the mapping between logical and physical addresses. When the FTL runs short on available free blocks, it transparently selects a block for erasing, based on the GC policy. It then moves valid pages inside the victim block to another block and erases them. Moving pages between blocks means that the page is re-written, and hence, a user write will result in more than one physical write on the SSD.

Flash cells also have a limited endurance, which is determined by the number of times they can be erased. The GC overhead therefore shortens the lifetime of SSDs and GC policies try to minimize the number of moved pages, which is equivalent to minimizing the number of erase cycles. To measure the efficiency of GC policies, the *write amplification factor* metric is employed, which is the ratio of physical writes in the SSD to the actual user writes.

To expand the SSD's lifetime, the FTL performs wear-leveling of flash blocks, such that blocks are worn out with a similar pace by selecting blocks with the lower erase counts. GC and wear-leveling policies can be considered orthogonal to each other [17], [5].

B. Zoned Namespace SSDs

Contrary to current SSDs, ZNS SSDs transition the responsibility of fine-grained data placement within flash blocks to the host, but continue to manage the media durability and reliability, including wear-leveling, internally. The ZNS interface exposes zones, that must be written sequentially and reset if rewritten, aligning to the flash media characteristics. Each zone maintains a write pointer, which defines the next logical block to be written in the zone. Data can be only written to a zone at the address of the write pointer, which is incremented after each write (see Fig. 1).

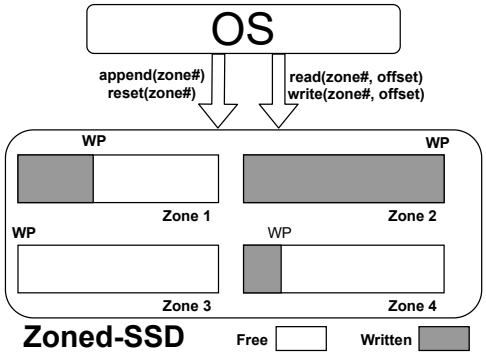


Fig. 1: ZNS SSD architecture [6]

If the host software requires a random write interface to be used, the host may implement a host-side translation layer. This layer, similarly to the FTL of conventional SSDs, has to implement a translation table, the GC process, associated memory, and computation. We use the new ZNS SSD interface to focus on the GC algorithms used to expose a random write block storage interface and evaluate its overheads.

C. Victim selection strategies

The **greedy** policy [18] is one of the oldest and simplest GC policies for SSDs. It selects the block with the lowest number of valid pages to reduce the WAF. Although it can achieve an optimal WAF for uniformly random accesses [19], it has high WAF when running real-world applications. Greedy can be implemented with fairly small processing and memory overhead. It has $O(S_B)$ complexity when selecting the victim data block, while having $O(1)$ complexity in maintaining its data structures. Table I denotes the notations used throughout the paper. **LINK-GC** [14] adds preemption to greedy to reduce the long IO latencies imposed by the GC process. **Stable greedy** [11] further optimizes baseline greedy to not select blocks containing hot pages. It employs the time since last block modification as the hotness metric.

Notation	Description
S_B	Pages per block/zone
B_t	Last modification time of block
N_B	Number of blocks
U_B	Block utilization
$Erase_B$	Block erase count
B_{inval}	Number of invalid pages in a block
B_{valid}	Number of valid pages in a block

TABLE I: Description of notations

One of the shortcomings of the greedy policy is its lack of providing wear-leveling. Blocks containing cold pages will have a low number of invalid pages, and hence, will not be selected for garbage collection. The **cost-benefit** (CB) policy tries to address this problem by adding a timing factor to the cost function. It considers the time since the last modification of the block (writing or invalidating a page) as an additional factor to the number of invalid pages. Moreover, it takes the cost of moving valid pages into account. Equation 1 shows how CB calculates the benefit of selecting a block for garbage collection.

$$benefit_{CB} = (t_{now} - B_t) \cdot \frac{B_{inval}}{2 \cdot B_{valid}} \tag{1}$$

To move valid pages, we need to read them from the victim block and write them to another block. Therefore, two times the number of valid pages is considered as the cost of erasing a block. The time since the last modification of cold blocks increases over time and eventually, a cold block will have a high enough benefit value to be selected for erasure. It allows CB to reuse invalid pages in blocks containing mostly cold valid pages. The benefit value depends on the current time. It increases with varied pace for different blocks. Therefore, during GC the benefit value for *all* blocks needs to be calculated. Having $O(N_B)$ complexity within a standard implementation for selecting the GC victim makes CB policy not scalable in terms of SSD size. CB does not maintain any separate data structure and only needs to update a few values in the block data structure. ParaFS [20] uses greedy and CB internally, based on if GC is triggered by the lack of free blocks or the file system being idle. Although ParaFS can reduce the GC overhead during burst I/O accesses, the WAF increases in such cases because of using greedy. It will result in more writes by the GC, which decreases both performance and lifetime of the SSD.

Fast CB [5] identified the huge overhead of CB victim selection. To address it, Fast CB separates blocks into two classes. It selects the victim block from the class containing few blocks with the highest cost-benefit values. Therefore, the victim selection time is limited to a small number of blocks. It, however, imposes CPU overhead for maintaining the classes and reduces the accuracy of CB. **Appr-CB** [5] is another approximation for CB, which selects many blocks for erasure in each meta-iteration. Therefore, it does not need to scan blocks until the selected blocks from the previous scan are erased. It reduces the victim selection overhead of CB by a constant factor. Note that this will also reduce the accuracy of CB. The accuracy loss depends on the number of selected blocks in the meta-iteration and the workload characteristics. Therefore, a pre-defined value for the number of selected blocks might not be optimal for all workloads.

The **cost-age-times** (CAT) [16] policy adds the number of erases to the benefit function to give higher

priority to blocks with low erase count. It also employs a pseudo-log function of time to limit its effect on the benefit function. Menon *et al.* [13] suggested a different approach that prioritizes older blocks over mostly invalid blocks. It selects the block with the highest number of invalid pages from blocks having an age older than a predefined threshold.

To further increase the accuracy of the age detection mechanism, the **FeGC** [12] policy suggests per page aging calculation. The age of a page is determined by the time since its invalidation. FeGC employs the sum of the age of all invalid pages in a block as the scoring function to select a victim block. Similar to CB, the score function of FeGC also depends on the garbage collection time. Therefore, the scores need to be calculated at the time of the garbage collection. To reduce this overhead, the authors suggested employing several heaps for managing the blocks. They placed blocks based on their number of invalid pages into different heaps. The pace of increasing the score for blocks with the same number of invalid pages is the same, and hence, the ordering between them does not change over time. When a page is invalidated, it is removed from its heap and placed into the next heap.

To select the victim block, only the heads of heaps need to be compared. Since the number of heaps is equal to the number of pages in a block, the victim selection time of FeGC has the same order of complexity as greedy ($O(S_B)$). However, the cost of maintaining the data structures is higher in FeGC, since greedy needs a constant time for moving a block to the next list, while FeGC needs $O(log(N_B))$. FeGC relies on storing per-page metadata on the SSD and needs to retrieve it for new writes to pages. While the overhead of this method might not be significant in traditional SSDs, it imposes a significant overhead in ZNS SSDs. Each page write requires sending a request from the OS to the SSD to read the metadata, which cannot be done concurrently with the actual page write. Therefore, each page write will have an additional read request overhead. **FaGC+** [21] employs incremental write numbers instead of the time as the age, since the GC policy is not sensitive to the real time. However, the old write number still needs to be retrieved, by a read request, before writing a page.

D. Motivation

In this section, we discuss the limitations of existing GC policies for upcoming ZNS SSDs in terms of CPU and memory overheads while scaling the size and performance of SSDs and considering a host-side translation layer that exposes a conventional random write block interface.

1) Memory overhead: Implementing a random write block interface on the host inhibits similar overheads to existing SSD FTLs. Due to the fine-grained data placement, that requires data to be written sequentially within a zone, the host must maintain a mapping table where its memory overheads are imposed on the system memory. A server usually manages tens of SSDs at the same time. Hence, having a high memory overhead is not acceptable for host FTLs. GC policies like greedy and CB only require a fixed number of variables per block. FeGC requires two sets of data structures in addition to the same number of added variables to block data structure as CB. First, it requires several variables *per-page* to identify hot/cold pages. Second, it needs several heaps to maintain the sorted lists of blocks. Maintaining per-page data structures is not feasible in large SSDs. For instance, keeping only a 4 byte variable per page will result in 1 GB memory overhead per 1 TB of SSD. FeGC and other GC policies like FaGC+ try to overcome this problem by storing per-page data structures within the flash page's metadata section.

The block interface does provide the ability to pass this metadata through per-LBA metadata, but is not commonly available. When not available, the metadata exhibits extra write/read requests in addition to the actual data. Therefore, the hot/cold identification of such GC policies cannot be employed generally and we consider them without this functionality. Even when the interface is available, the metadata stored with the previous version of the page needs to be retrieved, updated, and then written with the new version of the page. Therefore, still, a read I/O needs to be performed. FaGC+ relies on per-page information for victim selection, and hence, it cannot be practically used in ZNS SSDs.

The second memory overhead of FeGC comes from maintaining several heaps. The number of heaps is equal to the number of pages in a block. In ZNS SSDs with large zones, FeGC needs thousands of heaps. The maximum size of heaps is equal to the number of zones. Hence, we cannot pre-allocate all heaps with their maximum size. Dynamically increasing/decreasing the size of heaps in the runtime still imposes overheads, since one cannot perform too many heap expand/shrink without significant performance overhead.

In summary, Greedy and CB have ~0.0007% memory overhead, compared to the storage capacity for 1MByte zones. Appr-CB imposes ~0.0014% memory overhead. FeGC and FaGC+ have ~0.101% and ~0.1007% overhead, respectively. FeGC without hot/cold identification, however, imposes ~0.001% memory overhead.

2) CPU overhead: A host-side FTL acts as a middle-layer between applications and SSD, and impacts

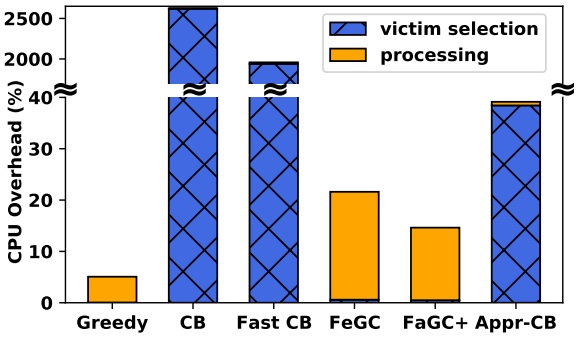


Fig. 2: CPU overhead

the host CPU utilization when processing I/Os. As mentioned before, many SSDs can be connected to a server, which all need their own FTL. If the FTL consumes too much processing power, the CPU will become a performance bottleneck when servicing I/O requests. GC policies have two types of overhead: processing overheads for maintaining data structures and for selecting the victim block. Previous studies focused on minimizing the overhead of selecting victim blocks by imposing higher maintenance overhead.

Fig. 2 shows the CPU overhead of various GC policies managing a 2TB SSD performing 200K IOPS (see Section IV-A for experimental setup). CB requires examining *all* blocks, and hence, it is not possible to employ it in this setup. Fast CB reduces the CPU overhead of CB by 27%. However, it still needs to perform complex operations on its data structures, and therefore, still has a significant CPU overhead. Greedy has the lowest CPU overhead for both processing and victim selection because of its simple architecture. Both FeGC and FaGC+ impose a high CPU overhead, which is mostly because of their processing overhead. Appr-CB, on the other hand, has a high victim selection overhead, since it only reduces the CB victim selection time by a constant factor.

We can conclude that current GC policies suffer either from 1) high memory overheads like FeGC with wear-leveling and FaGC+ or 2) high CPU processing like CB and FeGC without wear-leveling. Therefore, there is a need for a GC policy which provides a suitable WAF, while having a processing and victim selection overhead similar to that of greedy.

III. Optimizing cost-benefit

CB has shown to be able to reduce the WAF for many real-world traces compared to the greedy strategy, while there have been only a few attempts to reduce its significantly high victim selection overhead [5]. They, however, do not entirely eliminate it and can also affect the WAF. This section presents our new strategy *constant cost-benefit* (CCB), which uses ordered lists for organizing blocks. It accelerates the victim selection process by limiting it to a small fraction of relevant blocks. As a preliminary step, we first emphasize two properties of the cost-benefit strategy, which must be preserved by ordered lists to maintain cost-benefit's selection quality. We then show that cost-benefit can select a victim block by simply comparing the heads of all lists, and that ordered lists inflict only constant run time maintenance cost.

A. Preliminary characteristics

(1) Strict monotonicity of age: Let b_0 and b_1 be any two blocks that have the same amount of valid pages at some time t. Then the cost-benefit values of these two blocks are equal if and only if they have the same age:

$$f^{cb}_{b_0}(t) = f^{cb}_{b_1}(t) \Leftrightarrow age_{b_0}(t) = age_{b_1}(t),$$

and the cost-benefit value of b_0 is greater than the one of b_1 if and only if b_0 is older

$$f^{cb}_{b_0}(t) > f^{cb}_{b_1}(t) \Leftrightarrow age_{b_0}(t) > age_{b_1}(t).$$

If no pages of these blocks are invalidated, then this ordering is preserved as time progresses.

(2) Obliviousness: On every invalidation of a block's page the block's last invalidation time is set to the current time so that its age becomes zero. The age does not reflect the time it took to invalidate the previous pages, and we can therefore say that the block is oblivious about its age. As a result, the block's cost-benefit value becomes zero whenever a page is invalidated.

B. CCB design

First, we leverage cost-benefit's monotonicity for reducing the selection time. For our new data structure we create S_B double linked lists where S_B denotes the number of pages per block. Each list i stores all blocks that have exactly i valid pages and sorts them by their age in ascending order such that the head is the oldest and the tail is the youngest block (see Fig. 3). Due to the monotonicity, this design allows cost-benefit to only compare the heads of the lists as the head must have the highest cost-benefit value in its list. Then the selection time is reduced from $O(N_B)$ to $O(S_B)$.

Reducing the selection time, however, is only beneficial if maintenance of the data structure does not impose additional costs. Previous works like FeGC also employ a similar technique to reduce the victim selection time. However, maintaining their data structure (i.e., heaps) adds significantly high

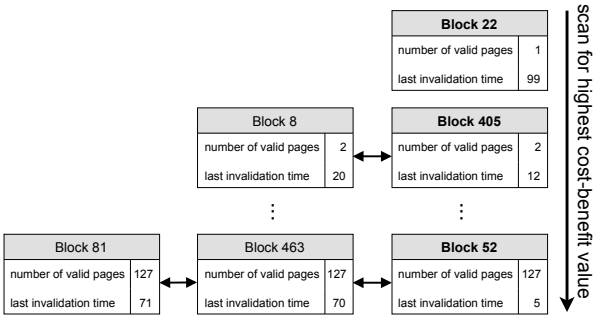


Fig. 3: Overview of ordered lists: Each list stores all blocks sharing the same number of valid pages in ascending order by age.

overhead. We therefore explain next how the list functions `append()`, `remove()`, and `find()` are implemented, and show that they not only suffice to operate the lists, but that they also run in constant time.

Using a double linked list, appending a block as a list's new tail requires constant time by setting the left and right pointers. Likewise, removing a block has the same overhead. We extend the zone data structure by a left and a right pointer to create ordered lists. Additionally, we keep one pointer to the head and one pointer to the tail for each list. Note that the FTL already knows the corresponding zone data structure for accessed addresses through its mapping table, and hence, accessing the corresponding zone in a list has a complexity of $O(1)$.

Finally, we must show that the list is kept in ascending order, which is immediately implied by the cost-benefit function's obliviousness. Whenever some block b_0 is invalidated, its number of valid pages decreases from i to $i-1$, and thus it must be removed from list i and inserted into list $i-1$. Note that ordered lists stores full blocks only, and thus their number of valid pages can only decrease and not increase. Since the block at list $i-1$'s tail b_1 has at least the same age as b_0, its cost-benefit value is at least as large as the one of b_0, as $f^{cb}_{b_1}(t) \geq f^{cb}_{b_0}(t)$. Hence, appending the block preserves the ascending order without additional overhead.

In terms of memory overhead, we added two pointers each accounting for four bytes, one pointer to each list's head, and one pointer to each list's tail. This totals up to additional eight bytes per zone and eight bytes per list. The additional memory overhead imposed by CCB is the same as the overhead of implementing lists for greedy. We do not need to add pointers from FTL to the lists, since linked-list pointers are integrated into the zone data structure and FTL already includes pointers to the zone data structure containing the physical page for any logical page.

IV. Experimental results

In this section, we first detail the experimental setup. To have a fair comparison, we selected GC policies that can be employed for large SSDs and have reasonable memory overheads: greedy, Appr-CB, FeGC, and our proposed CCB. Greedy has very low overheads and is considered as the lower bound of GC policy overheads. Appr-CB exemplifies GC policies with high victim selection time. We selected Appr-CB over Fast CB and original CB since it imposes less processing overhead. FeGC, on the other hand, represents the GC policies maintaining a costly data structure. FaGC+ is also excluded because it relies on per-page metadata, which is significantly costly to access in ZNS SSDs as discussed in Section II-C. We show the CPU overhead of various GC policies when scaling the SSD size and the number of concurrent writes. To show the effect of employing large zones in ZNS SSDs, we compare the WAF of GC policies on many zone sizes.

A. Evaluation setup

In the experiments, we employed an enterprise server with a single 10 core Xeon Gold processor running at 2.50 GHz. The server has 192 GByte of memory and has been running CentOS 8.2 and Linux kernel 5.1.0. We implemented all GC policies within dm-zap [9], which is a kernel device mapper, that implements a host-side FTL. It exposes a conventional random write block interface using ZNS SSD as its storage media. To ensure reproducibility, we obtained traces from several applications and replayed them in our experiments. For the **TPCC** workload, we used HammerDB[1] on top of a MySQL server. The **YCSB A** [22] workload also used MySQL as the backend database. It has employed a Zipf distribution where 95% of the requests are write operations. In addition, we have used **src1_0** and **spc1** from the MSR traces [23] as additional trace files. Table II reports the characteristics of all workloads. We have measured the WAF when GC policies reach their steady-state. The over-provisioning factor in all experiments is set to 10%. We used 1MByte zones in the evaluation, unless stated otherwise.

B. CPU overhead

As discussed in Section II-D, high CPU overhead is one of the shortcomings of previous GC policies. Our proposed GC policy addresses this issue. Fig. 4 shows the CPU overhead of various GC policies for different SSD sizes. Note that the CPU overhead mostly depends

[1]https://www.hammerdb.com/

Workload	# of accesses	Working set
TPCC	527M	98GB
YCSB A	505M	23GB
src1_0	16M	117GB
spc1	500M	110GB

TABLE II: Workload characteristics

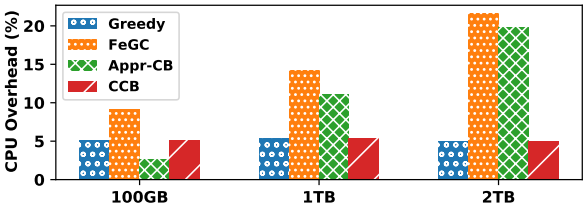


Fig. 4: CPU overhead of GC policies for various SSD sizes

on the SSD size, rather than the workload. The overhead of previous GC policies increases with the SSD size, since the maintenance cost of their data structure is not constant. For instance, by increasing the SSD size, the number of items in FeGC's heaps increases, which affects the heap insert/delete latency.

When the size of the SSD is small, i.e., 100GB, Appr-CB reduces the CPU overhead even compared to greedy and CCB. However, it needs to scan all blocks occasionally, and hence, its overhead is affected more by increasing the SSD size. CCB, on the other hand, manages its data structure in constant time and imposes less CPU overhead. For 2 TB SSDs, CCB has 74% less CPU overhead compared to other GC policies. Since the required computations for both greedy and CCB are almost the same, their CPU overhead is also very similar.

In addition to the SSD size, the number of concurrent accesses also affects the CPU overhead of GC policies. They either employ spinlocks or mutexes to provide concurrency. Since previous GC policies perform time-consuming operations in their critical section, a higher amount of concurrent writes will significantly increase their latency. Concurrent writes can originate from multiple threads/processes or asynchronous writes by a single thread.

Fig. 5 shows the CPU overhead of GC policies for various thread counts on a one TB SSD. Most of the locks in our implementation are based on spinlocks, since using mutexes further increases the latency. The CPU overhead of FeGC almost doubles when changing the number of concurrent writes from 1 to 64. This shows that complex data structures not only limit the scalability in terms of the SSD size, they also impose performance degradation in highly concurrent workloads. Contrary to scaling the SSD

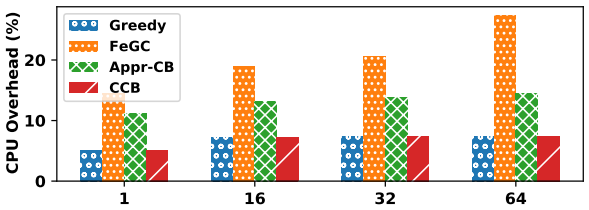


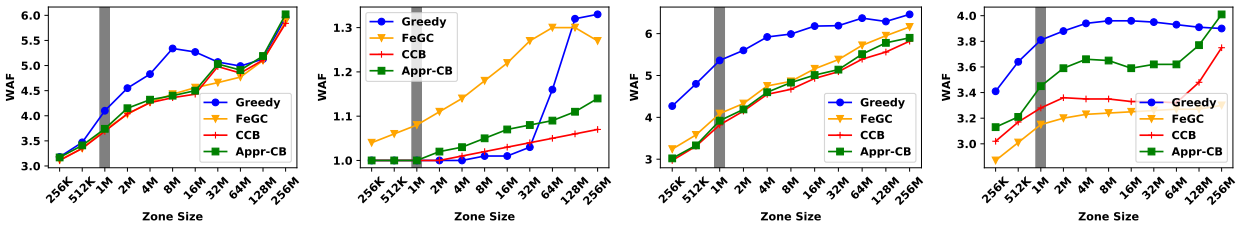
Fig. 5: CPU overhead of GC policies for various thread counts

size, here, Appr-CB is less sensitive to the number of concurrent writes than FeGC. This is because it does not require complex operations for updating data structures. The CPU overhead of CCB (as well as greedy) also increased slightly, since it needs to hold a lock on the linked-list for a small amount of time. Scaling from 1 to 64 threads only increases the CPU overhead of CCB by 15%, while Appr-CB and FeGC see an increase of 30% and 86%, respectively.

C. Write amplification

The main goal of GC policies is to reduce the WAF. Although the WAF of GC policies has been evaluated in previous works, ZNS SSDs impose limitations on some of these policies. For instance, the wear-leveling approach in FeGC significantly degrades the performance, and hence, cannot be employed in ZNS SSDs. Additionally, GC policies assume a relatively small zone size and are mostly optimized toward such values. ZNS SSDs can have significantly larger zones, i.e., >32MB. Here we evaluate the WAF of GC policies on different zone sizes, to 1) show whether employing CCB can degrade SSD lifetime, and 2) analyze the effect of increasing the zone size on WAF. Fig. 6 shows the WAF for various workloads. A zone size of 1MByte is highlighted, since we have used this value as the zone size in all previous experiments.

For the *YCSB A* workload (Fig. 6a), FeGC, Appr-CB, and CCB provide an almost identical WAF until a zone size of 32MByte where CCB and Appr-CB have a spike. Our analysis shows that many zones have an almost identical score in our experiments and CCB has selected the ones which have resulted in a higher WAF. Appr-CB performs similarly to CCB, since there has existed a group of hot zones which have been easily identified by Appr-CB. Greedy has produced higher WAF values for 1MByte up to 32MByte and has a spike for 8MByte zones, which shows that employing larger zones will not yield a higher WAF in all cases. After 64MByte, all policies suffer from a jump in WAF, which is caused by mixed hot/cold data in the zones. It shows that separating hot/cold data provides more benefits when larger zones are employed.



(a) YCSB (b) src1_0 (c) TPCC (d) spc1

Fig. 6: Write amplification factor for various zone sizes

For the *src1_0* workload (Fig. 6b), CCB provides lower WAF than other policies when zones are smaller than 2MByte. Between 2MByte and 32MByte, greedy offers slightly better WAF, and afterward, CCB again offers the best WAF. The WAF of FeGC increases at a faster pace than CCB, which results in a wide gap for larger zones. The WAF of greedy suddenly jumps when zones become larger than 32MByte, however, CCB maintains its efficiency even for large zones. Contrary to the previous workload, Appr-CB cannot accurately identify a suitable victim zone, and hence, its WAF diverges from CCB. However, it maintains the same pattern as CCB.

For *TPCC* workload (Fig. 6c), CCB provides lower WAF than other policies. However, the difference between CCB, Appr-CB, and FeGC is small. The difference between greedy and other policies is decreased when using larger zones. By increasing the zone size, WAF increases for all evaluated policies. ZNS SSDs with 256MByte zones have 2× the WAF of 256KByte zones. This shows that the current GC policies are not efficient for ZNS SSDs in some workloads.

The *Spc1* workload (Fig. 6d) shows a different behavior compared to the previous workloads. For all GC policies, until 2MByte, the WAF increases. Afterward, the WAF remains constant, with the exception of between 64MByte and 512MByte in CCB, and of course, Appr-CB. Employing larger zones for this workload reduces the overhead of GC policies without a penalty on the SSD lifetime. FeGC provides a lower WAF than CCB for all zone sizes because it considers the invalidation time of individual pages in the block, rather than the last invalidation time as being evaluated in CB and CCB. For the same reason, the spike for large zone sizes in this workload is higher for CCB than for other workloads. In this workload, also the difference between CCB and Appr-CB is higher than other workloads. This is because the cost-benefit values are constantly changing and Appr-CB selects victim zones based on outdated cost-benefit values.

V. Conclusion

The performance and lifetime of SSDs heavily rely on FTL functionalities like garbage collection. Since FTLs have been traditionally implemented inside SSDs, evaluating them directly in state-of-the-art hardware implementations has not been possible. Emerging SSD architectures like ZNS allow the fine-grained data placement within FTLs to be managed by the host, and hence, evaluating them becomes possible. In this paper, we have shown that current GC policies suffer from several scalability limitations, which renders them as not being practical for large SSDs. Our analysis has revealed that the victim selection time is not the limiting factor for many GC policies, but the maintenance of their internal data structures. To mitigate such limitations, we have proposed the constant cost-benefit policy, which has an identical processing overhead compared to the greedy strategy, while offering the same WAF as cost-benefit. The experimental results have shown that CCB reduces the CPU overhead by up to 74%, compared to previous policies, while having comparable WAF.

In the future, we plan to further analyze the behavior of GC policies in large zone sizes and improve their WAF in such cases. Also, we aim to take wear-leveling into account and optimize GC policies based on it.

References

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. I. T. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *Proceedings of the EuroSys Conference, Nürnberg, Germany*, 2009, pp. 145–158.

[2] Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, p. 23, 2011.

[3] H. Sun, X. Qin, F. Wu, and C. Xie, "Measuring and analyzing write amplification characteristics of solid state disks," in *IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), San Francisco, CA, USA, August 14-16*, 2013, pp. 212–221.

[4] P. Desnoyers, "Analytic modeling of SSD write performance," in *The 5th Annual International Systems and Storage Conference (SYSTOR), Haifa, Israel, June 4-6*, 2012, p. 14.

[5] L. Nagel, T. Süß, K. Kremer, M. U. Hameed, L. Zeng, and A. Brinkmann, "Time-efficient garbage collection in SSDs," *CoRR*, vol. abs/1807.09313, 2018.

[6] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. Ganger, and G. Amvrosiadis, "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs," in *USENIX Annual Technical Conference (ATC)*, 2021, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/bjorling

[7] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete," in *Workshop on Hot Topics in Operating Systems (HotOS), Ann Arbor, Michigan, USA, June 1-3*, 2021, pp. 144–151.

[8] M. Bjørling, "NVM Express Zoned Namespaces Command Set 1.1," Jun. 2020, available from http://www.nvmexpress.org/specifications.

[9] H. Holmberg, "dm-zap: Host-based FTL for ZNS SSDs," https://github.com/westerndigitalcorporation/dm-zap, 2021.

[10] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The Linux Open-Channel SSD Subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, 2017, pp. 359–374.

[11] L. Chang, Y. Liu, and W. Lin, "Stable greedy: Adaptive garbage collection for durable page-mapping multichannel SSDs," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, pp. 13:1–13:25, 2016.

[12] O. Kwon, K. Koh, J. Lee, and H. Bahn, "FeGC: An efficient garbage collection scheme for flash memory based storage systems," *Journal of Systems and Software*, vol. 84, no. 9, pp. 1507–1523, 2011.

[13] J. Menon and L. Stockmeyer, "An age-threshold algorithm for garbage collection in log-structured arrays and file systems," in *High Performance Computing Systems and Applications*, 1998, pp. 119–132.

[14] S. Jung and Y. H. Song, "LINK-GC: a preemptive approach for garbage collection in NAND flash storages," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC), Coimbra, Portugal, March 18-22*, 2013, pp. 1478–1484.

[15] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the Technical Conference on UNIX and Advanced Computing Systems (USENIX), New Orleans, Louisiana, USA*, 1995, pp. 155–164.

[16] M. Chiang and R. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[17] L. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proceedings of the ACM Symposium on Applied Computing (SAC), Seoul, Korea*, 2007, pp. 1126–1130.

[18] M. Wu and W. Zwaenepoel, "envy: A non-volatile, main memory storage system," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, USA*, 1994, pp. 86–97.

[19] Y. Yang, V. Misra, and D. Rubenstein, "On the optimality of greedy garbage collection for ssds," *SIGMETRICS Performance Evaluation Review*, vol. 43, no. 2, pp. 63–65, 2015.

[20] J. Zhang, J. Shu, and Y. Lu, "Parafs: A log-structured file system to exploit the internal parallelism of flash devices," in *USENIX Annual Technical Conference (ATC), Denver, CO, USA, June 22-24*, 2016, pp. 87–100.

[21] H. Yan, Y. Huang, X. Zhou, and Y. Lei, "An efficient and non-time-sensitive file-aware garbage collection algorithm for NAND flash-based consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 65, no. 1, pp. 73–79, 2019.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), Indianapolis, Indiana, USA*, 2010, pp. 143–154.

[23] D. Narayanan, A. Donnelly, and A. I. T. Rowstron, "Write off-loading: Practical power management for enterprise storage," in *6th USENIX Conference on File and Storage Technologies (FAST), February 26-29, San Jose, CA, USA*, 2008, pp. 253–267.