To Collect or Not to Collect: Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes

Sangwook Shane Hahn†, Sungjin Lee* and Jihong Kim†
†Department of Computer Science and Engineering, Seoul National University
*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
shanehahn@davinci.snu.ac.kr, chamdoo@csail.mit.edu, jihong@davinci.snu.ac.kr

ABSTRACT

For NAND flash-based storage systems, managing garbage collection (GC) efficiently is a critical requirement to achieve both high performance and long lifetimes. In this paper, we propose a just-in-time GC technique, called JIT-GC, which invokes background GC operations only when necessary depending on future write demands. JIT-GC was motivated by our measurement study, which strongly suggested that deciding when to invoke background GC operations is a key parameter for efficient GC. By accurately estimating the amount of future SSD writes, JIT-GC can choose the best time to invoke a background GC operation. JIT-GC reserves necessary free space in advance so that high write performance can be achieved while it extends the SSD lifetime by preventing premature block erasures. Our evaluations on real SSDs show that JIT-GC can achieve both high performance and long lifetimes, thus overcoming the shortcomings of existing background GC invocation heuristics.

1. INTRODUCTION

Garbage collection (GC) is the essential operation for NAND flash-based storage systems. Because of the NAND's erase-before-write constraint, when writing new data to a NAND page, an out-place update is used by writing new data to a new NAND page instead of updating the original page. Since out-place updates generate invalid pages with old data, garbage collection is necessary to reclaim invalid pages for future writes.

Since GC requires valid page migrations as well as block erasures, it incurs a significant performance overhead to a NAND flash-based storage system. In particular, if GC is necessary for an outstanding write request (i.e., when a foreground GC (FGC) operation is necessary), there can be a significant degradation in system performance (or user experience). Furthermore, since both the NAND program time and the number of pages per block tend to increase (e.g., 0.2 ms and 64 pages/block at 130-nm NAND chips to 2.3 ms and 384 pages/block at 25-nm NAND chips [1, 2], respectively), the impact of GC on the system performance, when poorly managed, is expected to be even bigger in future high-performance NAND-based storage systems.

In order to hide the performance penalty of a foreground GC operation, background garbage collection (BGC) is commonly used when a storage system is idle so that most foreground GC operations can be avoided. Although background GC operations can be effective in reducing the performance penalty of foreground GC operations, careful considerations are required in deciding *when* to invoke BGCs. An aggressive BGC policy, which reserves a large free space in advance, can avoid most FGC operations and achieves high write performance. However, its performance improvement comes with a shortened lifetime because the aggressive BGC policy tends to erase NAND blocks in a premature fashion. For example, in order to reclaim a required large free space, blocks with soon-to-be invalidated pages may be selected as GC victims, thus incurring a large number of useless page migrations. On the other hand, a lazy BGC policy, which maintains a small free space, does not sacrifice the lifetime of an SSD. However, it can significantly degrade the write performance because some foreground GC operations may be needed. Therefore, selecting the right time for invoking a background GC operation is important for designing a high-performance SSD with a long lifetime.

In this paper, we propose a just-in-time (JIT) GC technique, called JIT-GC, that performs background garbage collection operations only when necessary, thus improving both the performance and lifetime of NAND-based storage systems. JIT-GC aims to overcome the shortcomings of existing BGC policies such as aggressive BGC and lazy BGC. In order to perform garbage collection in a just-in-time fashion, we need to know future write demands in advance because the timeliness of GC invocations depends on how much future writes are performed. To estimate the future write demands, we take advantage of a typical I/O datapath between an application (that generates I/O requests) and a storage device (that eventually serves the I/O requests). JIT-GC employs two different prediction techniques in estimating the amount of future disk writes depending on the type of disk writes. For buffered writes, where written data are first placed in the page cache and later flushed from the page cache to an SSD (according to prespecified rules), we can predict with a high accuracy that when and which of buffered data will be written to the SSD by analyzing the page cache management algorithm. For example, in a Linux page cache, buffered data are flushed to an SSD after a predefined delay (e.g., 30 seconds). By scanning the page cache, we can easily tell how much of data will be written to the SSD, say, after 10 seconds. Furthermore, we can identify soon-to-be-invalidated pages within the SSD by scanning dirty pages in the page cache.

For direct writes, which bypass the page cache, it can be difficult to estimate the amount of future writes because such writes can happen at any time. In JIT-GC, direct writes are managed by using a dedicated over-provisioning space. Although a sufficient over-provisioning area can service direct writes without invoking FGC operations, maintaining a large over-provisioning area has the same disadvantage of an aggressive BGC technique. In order to maintain the right amount of the over-provisioning area for direct writes, JIT-GC employs a heuristic predictor for estimating future direct writes. Our heuristic estimates future direct writes using

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06...$15.00
http://dx.doi.org/10.1145/2744769.2744918 .

the cumulative data histogram of given I/O traffic.

Using an estimated future write demand from two predictors, JIT-GC decides when to invoke a BGC operation. Unlike an aggressive BGC policy or a lazy BGC policy where the future write demands are not fully exploited, JIT-GC can take advantage of an accurate estimate on future writes, thus choosing the best time to invoke a BGC operation. JIT-GC further improves the GC efficiency by exploiting the information on soon-to-be-invalidated pages from the page cache. By modifying a GC victim selection rule to take account of this information, JIT-GC tends to avoid blocks with many soon-to-be-invalidated pages as GC victim blocks.

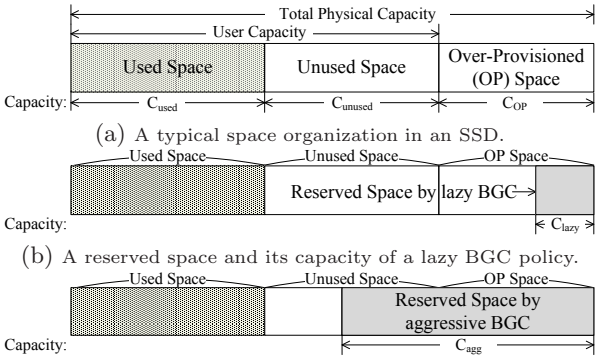
In order to evaluate the effectiveness of the proposed JIT-GC, we have implemented JIT-GC in Samsung SM843T SSDs [3] with a capacity of 240 GB. (The SM843T SSD, which is based on 20-nm MLC NAND flash, are targeted for data center SSDs.) For our JIT-GC implementation, the host interface of SM843T SSD was extended to support several new interface functions required for invoking BGC operations in a just-in-time fashion. (For a more detailed description of an extension to SM843T SSDs, see Sec. 4.)

Our experimental results on SM843T SSD using `YCSB` [4], `Postmark`, `Filebench` [5], `Bonnie++`, `Tiobench` and `TPC-C` benchmarks show that JIT-GC simultaneously achieves both the performance of an aggressive BGC policy and the lifetime of a lazy BGC policy. In our evaluations, the lazy BGC policy and aggressive BGC policy maintain 50% and 150% of SSD's over-provisioning space as reserved space, respectively. For performance, JIT-GC improves IOPS (Input/Output Operations Per Second) by up to 27% over the lazy BGC policy while maintaining a similar IOPS level to that of the aggressive BGC policy. For lifetime, JIT-GC decreases WAF (Write Amplification Factor) by up to 22% over the aggressive BGC policy while maintaining a similar WAF level to that of the lazy BGC policy.

The rest of this paper is organized as follows. In Sec. 2, we quantitatively evaluate how existing BGC invocation heuristics affect the performance and lifetime of SSDs. Sec. 3.2 describes our proposed predictors on future write demands. In Sec. 3.3, we describe the proposed just-in-time garbage collection in detail. Experimental results are given in Sec. 4, and related work is summarized in Sec. 5. In Sec. 6, we conclude with a summary.

2. IMPACT OF BGC INVOCATION TIMES ON PERFORMANCE AND LIFETIME

In order to understand the impact of BGC invocation times on the performance and lifetime of SSDs, we first review a typical SSD space configuration which directly influences the BGC invocation frequency. As shown in Fig. 1(a), most SSDs divide the total physical capacity into a user capacity and an over-provisioning capacity. The over-provisioning (OP) capacity C_{OP}, which is not available for storing user data, is a reserved capacity exclusively for an SSD management software such as FTL. The OP capacity is useful in improving both SSD performance and lifetime. For example, the OP capacity allows to delay GC invocations because free pages from the OP space can be transparently allocated to user data, thus avoiding garbage collections even when there are few free pages left in the unused space. However, as user data fills up the user capacity of the SSD, the OP space is also filled up with invalid pages. In order to prevent OP from being full of invalid pages, BGC is invoked to reclaim invalid pages (up to its reserved capacity) when a storage system is idle.



(a) A typical space organization in an SSD.

(b) A reserved space and its capacity of a lazy BGC policy.

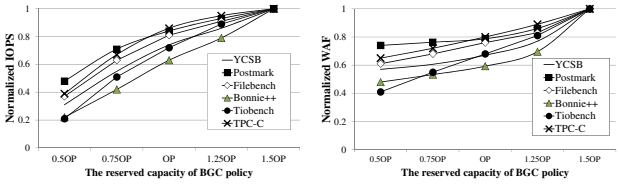
(c) A reserved space and its capacity of an aggressive BGC policy.

Figure 1: SSD space configurations under different BGC invocation schemes.

Since the average BGC invocation rate is largely determined by a reserved capacity C_{resv} of a BGC policy, we say that a BGC policy is lazy when the reserved capacity is small. When a large reserved capacity is required, more frequent BGC invocations are necessary. For such a BGC policy, we call it an aggressive BGC policy. Since there is no clear distinction between a lazy BGC and an aggressive BGC, in this paper, we assume that a BGC policy is lazy when C_{resv} of a BGC policy is smaller than the OP capacity C_{OP}. Fig. 1(b) shows an example SSD space configuration for a lazy BGC policy where the reserved capacity C_{lazy} of a lazy BGC policy is smaller than C_{OP}. For an aggressive BGC policy, we make a similar assumption that the reserved capacity C_{agg} of an aggressive BGC policy is larger than C_{OP}, as shown in Fig. 1(c). For an aggressive BGC policy, we further restrict that C_{resv} cannot be bigger than the sum of C_{unused} and C_{OP} so that we can avoid useless BGC operations when an SSD is filled with a large amount of user data.

In a lazy BGC policy where C_{lazy} is small, BGC invocation times are rather delayed until the unused space is almost exhausted. Infrequent BGC invocations in the lazy BGC policy help to avoid unnecessary data migrations so that the SSD lifetime can be extended. On the other hand, when the aggressive BGC policy is used, BGC operations are more frequently invoked so that a bigger C_{agg} can be maintained. These frequent BGC invocations, however, degrade the SSD lifetime by erasing NAND blocks too frequently.

In order to better understand the impact of the laziness/aggressiveness of a BGC policy on the GC efficiency, we measured IOPS and WAF on SM843T SSDs while varying C_{resv} of a BGC policy. We used four benchmarks, `YCSB` (an update-intensive workload), `Postmark`, `Filebench` (a file-server workload), `Bonnie++`, `Tiobench` and `TPC-C`, which were all run on a Linux-based PC with 8 GB memory (For a detailed description on the benchmarks, see Sec. 4). The working set size of each benchmark was set to half of the size of the user capacity. As shown in Fig. 2, C_{resv} was changed from $0.5 \times C_{OP}$ to $1.5 \times C_{OP}$. (For brevity, in the rest of this paper, we denote a lazy BGC policy with $C_{resv} = 0.5 \times C_{OP}$ as L-BGC and an aggressive BGC policy with $C_{resv} = 1.5 \times C_{OP}$ as A-BGC, respectively.) Measurement values shown in Fig. 2 are normalized over values measured by A-BGC. As expected, IOPS improves as C_{resv} increases while WAF gets smaller with a smaller C_{resv}. However, the differences in IOPS and WAF over different C_{resv} values were a lot bigger than ones we expected. As shown in Fig. 2, IOPS can be different by up to five times depending on C_{resv}. WAF can vary by up



(a) Impact of the reserved capacity on IOPS.

(b) Impact of the reserved capacity on WAF.

Figure 2: Impact of the reserved capacity on performance and lifetime.

to twice over different C_{resv}'s.

Our work on JIT-GC was strongly motivated by these measurement results which strongly indicated that the reserved capacity of a BGC policy is a key factor for efficient GC. Since we indirectly control the invocation frequency of a BGC policy by C_{resv}, this measurement suggests that deciding when to invoke BGC operations is a critical design parameter that affects the GC efficiency. Furthermore, Fig. 2 clearly shows that there is a tradeoff between the SSD performance and lifetime using a different C_{resv} value. Obviously, for example, neither L-BGC nor A-BGC can achieve the highest performance with the longest lifetime.

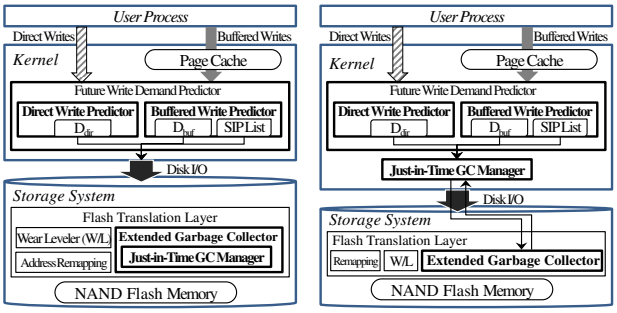
Judging from Fig. 2, the ideal BGC invocation policy is one that can dynamically changes C_{resv} so that only an exact amount of future writes can be reserved in advance. This ideal policy can avoid costly FGC operations while minimally affecting the NAND endurance. In order for such an ideal BGC policy to be feasible, it is important to accurately estimate future write demands in advance. However, it is quite challenging to accurately predict future write demands inside an SSD with device-level access information only. As an alternative solution for an accurate future write predictor, we propose to exploit the page cache and its management algorithm. As described in detail in the next section, a page cache-aware prediction on future write demands enables us to design and implement a near-ideal BGC policy in a practical setting.

3. DESIGN AND IMPLEMENTATION OF JIT-GC

3.1 Overall Architecture of JIT-GC

Fig. 3(a) shows an overall architecture of JIT-GC, which is composed of two main modules, a future write demand predictor and a JIT-GC manager. The future write demand predictor estimates future write traffic and forwards this information to the JIT-GC manager. As mentioned in Sec. 1, for accurate prediction of a future write demand, the future write demand predictor uses two different prediction methods for buffered writes and direct writes, respectively. The JIT-GC manager is responsible for reserving a proper amount of free space (requested by the future write demand predictor) in the SSD so that the expected future writes can be served without incurring FGC. To this end, the JIT-GC manager schedules BGC to reclaim required free space during idle times in the SSD.

Besides the information for future write traffic, the write demand predictor sends a list of logical addresses of dirty data in the page cache to the JIT-GC manager, which we call a soon-to-be-invalidated page (SIP) list. Although the dirty data stays in the page cache at a prediction time, it will be soon evicted to the SSD. Therefore, moving old versions of the dirty data stored in the SSD during BGC can be useless. Using the SIP list, the JIT-GC manager can exclude blocks containing soon-to-be-invalidated pages from BGC.



(a) An ideal implementation.

(b) An actual implementation on SM843T.

Figure 3: An overall architecture of JIT-GC.

The most efficient implementation of JIT-GC is, as shown in Fig. 3(a), to run the future write demand predictor in a host OS (because it requires the detailed information of the page cache) and to execute the JIT-GC manager in the SSD controller (because it needs to schedule BGC). The communications between the predictor and the manager (to transfer a predicted future write demand and a SIP list) can be easily supported by adding custom commands between the host and the SSD. (This type of an interface extension is often specially added to deliver host-level information in enterprise-class SSDs.) Because of practical difficulties in modifying the SM843T's FTL, however, it is not easy to implement the JIT-GC manager in the SM843T SSDs. As depicted in Fig. 3(b), instead, we implement the JIT-GC manager in the host, which sends BGC invocation commands and SIP lists to SM843T. The SM843T FTL was slightly modified to perform BGC with a SIP list when it gets explicit a BGC command from the JIT-GC manager.

3.2 Future Write Demand Estimation

In this subsection, we explain the details of the write demand prediction techniques for buffered and direct writes.

3.2.1 Write Demand Predictor for Buffered Writes

For fast I/O performance, most modern operating systems use a write-back cache management policy. Therefore, almost all data written or modified by user applications is first stored in the page cache and later evicted to the SSD. Since the modified data stays in the page cache as dirty data for a relatively long time before its eviction, the dirty data information from the page cache can be very useful in predicting future write traffic.

In the Linux kernel, dirty data in the page cache is evicted when following two conditions are satisfied. The first condition is when dirty data is older than an expiration threshold τ_{expire} (e.g., 30 seconds) since its last update. The second condition is when the total size of dirty data kept in the page cache is larger than a flush threshold τ_{flush} (e.g., 10% of the size of the remaining free space of the page cache). The flusher thread checks two conditions periodically. When two conditions are met, it flushes expired dirty data to the SSD. In this paper, we assume that the flusher thread is executed every p seconds. We call an interval between two consecutive flusher thread activation times as a write-back interval I_{wb}. When the flusher thread is activated at the start time s of the write-back interval $I_{wb} = [s, e]$, we denote its future write-back interval as $I^j_{wb}(s) = [s_j, e_j]$ for $j \geq 1$ where $s_j = s + j \times p$ and $e_j = s + (j+1) \times p$. In the rest of the paper, we assume that τ_{expire} is a multiple of p and we denote (τ_{expire}/p) as N_{wb}.

The proposed write demand predictor for buffered writes, which is designed to exploit the page cache management pol-

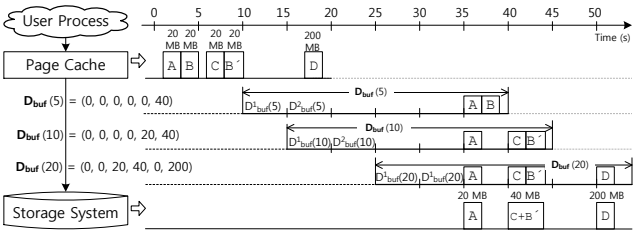


Figure 4: An example of future write demand estimation for buffered writes.

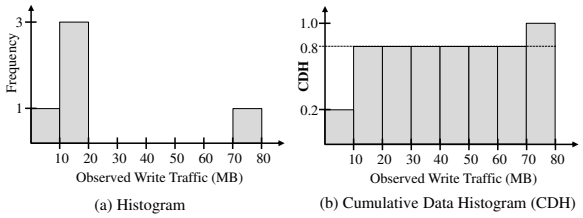


Figure 5: An example of CDH for direct writes.

icy explained above, is invoked right after the flusher thread is executed. The predictor algorithm, when invoked at time t, estimates a sequence $\boldsymbol{D_{buf}}(t) = (D^1_{buf}(t),\ D^2_{buf}(t),\ ...\ ,D^{\tau_{expire}/p}_{buf})$ of upper bounds on write demands to the SSD where $D^i_{buf}(t)$ indicates an upper bound on a write demand to the SSD for $I^i_{wb}(t)$. In predicting an upper bound $D^i_{buf}(t)$, we relax the second flush condition described above. We assume that every dirty data is flushed from the page cache once it gets older than τ_{expire} without checking if the current size of dirty data in the page cache is larger than τ_{flush}. With this simple flush condition, it becomes straightforward to predict an upper bound $D^i_{buf}(t)$ on future buffered writes for $I^i_{wb}(t)$. A simple scanning of dirty pages older than τ_{expire} is sufficient to compute $D^i_{buf}(t)$'s. While scanning dirty pages in the page cache, the predictor algorithm also adds the logical block address of each dirty page to a SIP list L_{SIP}.

Although a tighter future write demand can be predicted when both flush conditions are strictly checked, we relaxed the second flush condition of the flusher thread so that expensive FGC operations can be avoided under some special write patterns. For example, when a large-sized (i.e., $> \tau_{flush}$) buffered write is requested long after the size of dirty data in the page cache has been less than τ_{flush}, if we didn't relax the second condition, on the next write-back interval, a dirty data of the size τ_{flush} will be flushed to an SSD. Since this flush was not predictable in advance by the predictor algorithm, it is likely to cause an FGC operation. In order to avoid such FGC operations, we relaxed the second flush condition in predicting the future write demand. The amount of an over-prediction, however, is always limited by at most τ_{flush}.

Fig. 4 illustrates how our buffered write predictor estimates $\boldsymbol{D_{buf}}(t)$'s using the page cache information. In this example, p and τ_{expire} are assumed to be 5 and 30 seconds, respectively. Suppose that five write requests were written to the page cache in the order of `A, B, C, B′ and D`, where `B′` is the update write over `B`. At $t = 5$, since there are `A` and `B` in the page cache as dirty data and they were in the page cache less than 5 seconds, the predictor estimates $D^6_{buf}(5)$ as 40 MB while the other $D^i_{buf}(5)$'s as 0 MB, thus $\boldsymbol{D_{buf}}(5)$ is given as (0, 0, 0, 0, 0, 40). Note that although `A` and `B` become expired before $t = 35$, the predictor computes their flush times to the SSD in $I^6_{wb}(5)$ (i.e., [35, 40]) instead of $I^5_{wb}(5)$ (i.e., [30, 35]) because their flush holds until the flusher thread wakes up at the beginning of $I^6_{wb}(5)$. At $t = 10$, the predictor estimates $\boldsymbol{D_{buf}}(10)$ as (0, 0, 0, 0, 20, 40) reflecting `B′` and `C`. Note that $D^5_{buf}(10)$ is 20 MB, not 40 MB because `B` was updated to `B′` (resetting `B`'s age to zero), thus delaying its flush. $\boldsymbol{D_{buf}}(20)$ is similarly estimated as (0, 0, 20, 40, 0, 200) including `D`.

3.2.2 Write Demand Predictor for Direct Writes

The future write demand cannot be accurately predicted only with the page cache information because a nontrivial amount of data is written to the SSD directly, bypassing a page cache. For example, a database management system (DBMS) writes important metadata directly to a storage device without page-cache buffering. File systems also often issue direct writes to complete file-system journaling.

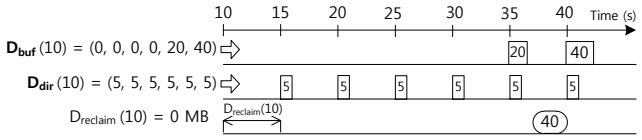
In order to estimate future write demand from direct writes, JIT-GC maintains a cumulative data histogram (CDH) of past direct writes and uses this information to decide a reserved free space for future direct writes. A direct write can be distinguished from a buffered write because it is issued with a special file-system flag (e.g., `O_SYNC`). (Note that an idea of using CDH is not new and is widely used to decide future behaviors of a system (e.g., [6]).)

Since we can predict future write demands for buffered writes only up to τ_{expire} seconds in the future (i.e., up to $I^{N_{wb}}_{wb}(t)$ at time t), the CDH is built based on the amount by direct writes over τ_{expire}-second intervals. Fig. 5 shows an example of how JIT-GC builds the CDH. JIT-GC internally maintains a histogram that keeps track of the amount of written data during past I_{wb}'s. Fig. 5(a) illustrates the resulting histogram when 10, 20, 20, 20, and 80 MB of data are written during the past 5 I_{wb}'s. The x-axis is a histogram bin with a 10-MB range and the y-axis is its frequency. Using the histogram, JIT-GC builds the CDH (Fig. 5(b)) which shows the cumulative probability distribution of how much data was written to the SSD over past τ_{expire}-second intervals. From Fig. 5(b), for instance, we know that, for 80% of the τ_{expire}-second intervals, less than 20 MB data were written to the SSD.

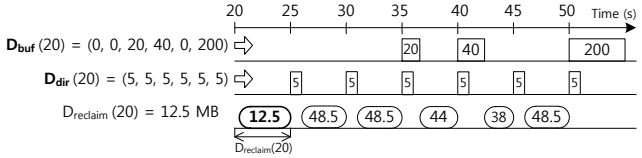
JIT-GC uses the CDH to decide the size of free space for direct writes under the assumption that the amount of data to be written by future direct writes would be similar to the previously observed one. In our current implementation, the predictor chooses $\delta_{dir}(t)$ at time t from the CDH so that reserving $\delta_{dir}(t)$ will avoid an FGC operation for future direct writes at least 80% of probability. For example, in Fig. 5(b), a reserved free space is chosen as 20 MB. It is obvious that, more FGC operations can be avoided with a higher percentage value. However, too high percentage values may negatively affect the overall lifetime of SSDs in a similar fashion as A-BGC. According to empirical observations, a percentage value of 80% seems to be the most suitable, balancing both performance and lifetime. When the predictor algorithm is invoked at time t, it returns a sequence $\boldsymbol{D_{dir}}(t) = (D^1_{dir}(t),\ D^2_{dir}(t),\ \cdots,\ D^{N_{wb}}_{dir}(t))$ of future write demands for direct writes where $D^i_{dir}(t)$, which is set to $\frac{\delta_{dir}(t)}{N_{wb}}$, indicates a write demand for direct writes at $I^i_{wb}(t)$.

3.3 JIT-GC Manager

At the beginning of each $I_{wb} = [s, t]$ (i.e., every p seconds), the JIT-GC manager receives $\boldsymbol{D_{buf}}(s)$ and $\boldsymbol{D_{dir}}(s)$ from the future write demand predictor. As described in Secs. 3.2.1 and 3.2.2, $\boldsymbol{D_{buf}}(s)$ and $\boldsymbol{D_{dir}}(s)$ represent the sequences of future write demands at $[s + p,\ s + p + \tau_{expire}]$ for buffered and direct writes, respectively. The JIT-GC manager also



(a) An example of $T_{idle} > T_{gc}$ with $C_{free}(10) = 50$ MB.



(b) An example of $T_{idle} < T_{gc}$ with $C_{free}(20) = 50$ MB.

Figure 6: Examples of allocating GC in advance with estimated future write demands.

gets the capacity $C_{free}(s)$ of free space available in the SSD at time s. (Note that $C_{free}(s)$ can be obtained from the SSD through the custom command described in Sec. 3.1)

In order for the JIT-GC manager to decide if an BGC operation should be invoked or not at time t, the JIT-GC manager checks if $C_{free}(t)$ is large enough for the sum of future write demands. That is, the JIT-GC manager checks if $C_{free}(t) > C_{req}(t)$ where $C_{req}(t) = \sum_{i=1}^{N_{wb}} (D^i_{buf}(t) + D^i_{dir}(t))$ and $N_{wb} = \frac{\tau_{expire}}{p}$. If $C_{free}(t) > C_{req}(t)$, the JIT-GC manages does not invoke a BGC operation at time t.

If $C_{free}(t) < C_{req}(t)$, the JIT-GC manager schedules required BGC operations as lazy as possible to reserve the required capacity of $(C_{req}(t) - C_{free}(t))$. Assuming that we know an average write bandwidth $B_w(t)$ and an average GC bandwidth $B_{gc}(t)$, we estimate the total amount of idle times $T_{idle}(t)$ in $[t + p,\ t + p + \tau_{expire}]$ as $T_{idle}(t) = \tau_{expire} - T_w(t)$ where $T_w(t)$, computed as $C_{req}(t)/B_w(t)$, is the time taken to write a request of a capacity $C_{req}(t)$ to a SSD. The time $T_{gc}(t)$ for performing BGC operations for reserving $(C_{req}(t) - C_{free}(t))$ is similarly computed as $(C_{req}(t) - C_{free}(t))/B_{gc}(t)$.

Once $T_{idle}(t)$ and $T_{gc}(t)$ are computed, the JIT-GC manager checks if $T_{idle}(t) > T_{gc}(t)$, that is, if it is possible to skip a BGC operation at the current write-back interval. If $T_{idle}(t) > T_{gc}(t)$, the JIT-GC manager does not invoke a BGC operation at this interval. If $T_{idle}(t) < T_{gc}(t)$, the JIT-GC manager invokes BGC operations to reclaim a free capacity of $(T_{gc}(t) - T_{idle}(t)) \times B_{gc}(t)$.

Fig. 6 illustrates that how the JIT-GC manager works with $\boldsymbol{D_{buf}}(t)$, $\boldsymbol{D_{dir}}(t)$ and $C_{free}(t)$. In this example, p and τ_{expire} are set to 5 and 30 seconds, respectively. Both $C_{free}(10)$ and $C_{free}(20)$ are set to 50 MB. We assume that $B_w(10)$ and $B_w(20)$ are 40 MB/sec while $B_{gc}(10)$ and $B_{gc}(20)$ are 10 MB/sec. At $t = 10$, as shown in Fig. 6(a), the JIT-GC manager receives $D_{buf}(10)$ and $D_{dir}(10)$ from the future write demand predictor and compares $C_{req}(10)$ with $C_{free}(10)$. Since $C_{req}(10)$ is 90 MB while $C_{free}(10)$ is 50 MB, JIT-GC manager needs to check whether BGC is necessary or not at the current write-back interval. Since $T_{idle}(10) > T_{gc}(10)$, that is, (30 - 90/40) > 40/10, JIT-GC manager decides not to trigger BGC during [10, 15], thus setting the requested reclaim demand $D_{reclaim}$ to 0. On the other hand, at $t = 20$, the JIT-GC manager estimates much heavier future write requests, that is, $C_{req}(20) = 290$ MB. Since $C_{req}(20) > C_{free}(20)$, the JIT-GC manager computes $T_{idle}(20)$ and $T_{gc}(20)$. Since $T_{idle}(20) < T_{gc}(20)$, that is (30 - 290/40) = 22.75 < 240/10 = 24, the JIT-GC manager invokes BGC during [20, 25] with $D_{reclaim} = 12.5$ MB.

Benchmark	YCSB	Postmark	Filebench	Bonnie++	Tiobench	TPC-C
Buffered writes	88.2%	81.7%	85.8%	72.4%	46.3%	0.1%
Direct writes	11.8%	18.3%	14.2%	27.6%	53.7%	99.9%

Table 1: Breakdowns of write types in six benchmarks.

4. EXPERIMENTAL RESULTS

4.1 Experimental Settings

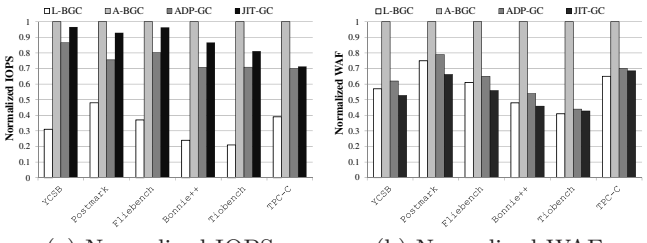
In order to evaluate the effectiveness of JIT-GC, we implemented JIT-GC on the extended SM843T SSD connected to a PC host running the Linux kernel (version 3.11.1). The extended SM843T has the user capacity of 240 GB with the C_{OP} capacity of 16 GB. As shown in Fig. 3(b), we implemented the future write predictor module in the Linux kernel. Furthermore, we added the JIT-GC manager module as well in the Linux kernel (instead of the inside SM843T) so that the existing SM843T FTL can be minimally modified in supporting JIT-GC. In addition to exchanging $\boldsymbol{D_{buf}}(t)$, $\boldsymbol{D_{dir}}(t)$, L_{SIP} and C_{free} using an extended SM843T host interface for JIT-GC, the extended host interface also supports several profiling functions (that can be called from the host). For example, using one of these functions, we can measure WAF values from SM843T. In the current implementation, the extend interface functions use the `SG_IO (SCSI generic I/O) ioctl` command. The `SG_IO ioctl` command, which is available from Linux 2.6, allows the host to send SCSI commands to a storage device. The `SG_IO ioctl` command requires a time overhead of about 160 μs in transferring data between SM843T and the host.

The six benchmarks were used for our evaluations: `YCSB` (Yahoo! Cloud Serving Benchmark running on Cassandra), `Postmark` (a mail-server workload benchmark), `Filebench` (a file-server workload benchmark), `Bonnie++` (a file system and storage system performance benchmark), `Tiobench` (a multi-thread I/O benchmark) and `TPC-C` (one of on-line transaction processing benchmarks running on MySQL). As shown in table 1, six benchmarks are different in their ratios between buffered writes and direct writes. The working set size of each benchmark was set to 120 GB (which is half the size of the 240 GB user capacity of SM843T) so that the unused user space can be used for the reserved space of GC policies such as A-BGC and JIT-GC.

In our evaluations, we compared IOPS values and WAF values of several different BGC management techniques, including JIT-GC, L-BGC and A-BGC. All the measurement values were normalized over the values measured with A-BGC. Since C_{OP} of SM843T was set to 7% of the 240 GB user capacity, the reserved space capacities, C_{lazy} and C_{agg}, of L-BGC and A-BGC are set to 8 GB (i.e., 0.5 $\times$ C_{OP}) and 24 GB (i.e., 1.5 $\times$ C_{OP}), respectively.

4.2 Evaluation Results

Fig. 7(a) shows normalized IOPS for six benchmarks with four different techniques. The adaptive GC technique (ADP-GC) dynamically changes the capacity of the reserved space but the future write demand estimation is completely performed inside an SSD so that the future write predictor does not distinguish between direct writes and buffered writes. The future write demand predictor of ADP-GC is based on the same write demand predictor used for direct writes for JIT-GC (which was described in Sec. 3.2.2). Furthermore, the ADP-GC technique does not exploit the SIP information in selecting a GC victim block. JIT-GC improves IOPS by 182%, on average, over L-BGC. The improvement ratios on IOPS are proportional to the prediction accuracy of future write demands. For example, as shown in Table 2, our predictor can predict very accurately the future write demands



(a) Normalized IOPS (b) Normalized WAF

Figure 7: Comparisons of normalized IOPS and WAF.

of YCSB, thus achieving almost the same IOPS as A-BGC. On the other hand, JIT-GC wasn't so efficient in predicting the future write demands of TPC-C (which consists of mostly direct writes). With the prediction accuracy of 72.5%, the IOPS value of JIT-GC was about 72% of that of A-BGC.

Fig. 7(b) shows that JIT-GC also improves WAF by 44%, on average, over A-BGC. Furthermore, JIT-GC achieves lower WAF values over (even) L-BGC for YCSB, Postmark, Filebench and Bonnie++ because of an efficient GC victim-block filtering method of JIT-GC. The improvement ratio on WAF is roughly proportional to the number of filtered blocks shown in Table 3. For example, JIT-GC can decrease the WAF value of Postmark by 14% over L-BGC. Combining Figs. 7(a) and 7(b), JIT-GC shows that it can closely achieve the IOPS level of A-BGC while it outperforms the WAF level of L-BGC for YCSB, Postmark, Filebench and Bonnie++. On the other hand, for Tiobench and TPC-C, JIT-GC doesn't perform as good as A-BGC on IOPS (although it is as good as L-BGC on WAF.) This is because it is fundamentally difficult to predict future write demands of direct writes (which are dominant in Tiobench and TPC-C) with the 90% prediction accuracy.

We also evaluated the effect of the buffered write predictor (which utilizes the page cache information) by comparing JIT-GC and ADP-GC. As shown in Figs. 7(a) and 7(b), JIT-GC outperforms ADP-GC by 15% on average for IOPS and reduces WAF by 11% on average over ADP-GC. These differences can be explained by the difference in the prediction accuracy as shown in Table 2. The prediction accuracy of JIT-GC is higher than that of ADP-GC by up to 20.4%.

5. RELATED WORK

Performing garbage collection in background to hide its high overhead has been studied by several researchers [6, 7]. For example, Park et al. presented an adaptive garbage collection technique that triggered BGC only when long idle periods were expected [7]. This technique can be useful in minimizing degradations in the user-perceived I/O response time by preventing from too aggressively exploiting idle times for background GC operations. Lee et al. [6] proposed a BGC technique that delays BGC as late as possible so that useless BGC operations can be avoided. The proposed JIT-GC is, however, fundamentally different from existing techniques in that it performs BGC in a just-in-time manner based on accurate system-level estimations on future write demands. Furthermore, unlike most existing techniques, which focus on only one aspect of quality metrics of an SSD (e.g., user response time or WAF), JIT-GC can improve both performance and lifetime of the SSD at the same time because it can proactively reserve only the required free space for future writes.

Exploiting the current status of a page cache for improving a NAND-based storage is not new. For example, Lee et al. [8] and Lee et al. [9] presented techniques that avoided useless copies for soon-to-be-invalidated pages in a page cache (or a buffer cache). While these techniques exploit the page cache for a limited purpose only (e.g., such as reducing extra I/O operations during GC), JIT-GC takes a full advantage of the page cache for improving the performance and lifetime of the NAND-based storage.

Benchmark	YCSB	Postmark	Filebench	Bonnie++	Tiobench	TPC-C
Prediction accuracy of JIT-GC (%)	98.9	93.2	97.3	89.8	86.1	72.5
Prediction accuracy of ADP-GC (%)	87.7	72.8	82.0	73.4	74.1	71.2

Table 2: Prediction accuracy of future write predictors of JIT-GC and ADP-GC.

Benchmark	YCSB	Postmark	Filebench	Bonnie++	Tiobench	TPC-C
Filtered GC victim blocks	12.2%	20.6%	17.5%	8.7%	4.9%	1.1%

Table 3: The effect of the SIP lists.

6. CONCLUSIONS

We have presented a just-in-time GC technique, called JIT-GC, which invokes BGC operations only when necessary based on predicted future write demands. JIT-GC is motivated by our empirical finding that deciding right BGC invocation times affects the GC quality significantly. By accurately estimating the amount of future write demands with host-side information, JIT-GC chooses the best time to invoke a BGC operation. JIT-GC creates an exact free space required for future writes in advance while preventing premature block erasures. Our evaluation results with Samsung SM843T SSDs showed that JIT-GC improved IOPS by 182%, on average, over L-BGC and reduced WAF by 44%, on average, over A-BGC, overcoming the shortcomings of existing BGC heuristics. Furthermore, JIT-GC outperforms a page cache-oblivious adaptive GC technique, ADP-GC, on average by 15% and 11% for IOPS and WAF, respectively.

ACKNOWLEDGEMENTS

We would like to thank Samsung Electronics for providing SM843T SSDs with an extension for our research. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A2A2A01068260). The ICT at Seoul National University and IDEC provided research facilities for this study.

REFERENCES

[1] H. Nakamura et al., "A 125mm^2 1Gb NAND Flash Memory with 10MB/s Program Throughput," in *Proc. IEEE Solid-State Circuits Conf.*, pp. 106-107, 2002.

[2] M. Goldman et al., "25nm 64Gb 130mm^2 3bpc NAND Flash Memory," in *Proc. IEEE Int. Memory Workshop*, 2011.

[3] SAMSUNG 843T Data Center Series, http://memorysolution.de/mso_upload/out/all/SM843T_Specification_v1.0.pdf

[4] B.F. Cooper et al., "Benchmarking Cloud Serving Systems with YCSB," in *Proc. ACM Symp. Cloud Computing*, pp. 143-154, 2010.

[5] P. Sehgal et al., "Evaluating Performance and Energy in File System Server Workloads," in *Proc. USENIX Conf. File and Storage Technologies*, pp. 19-33, 2010.

[6] S. Lee et al., "Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory," *IEEE Trans. Computers*, vol. 63, no. 10, pp. 2445-2458, 2014.

[7] S.-H. Park et al., "An Adaptive Idle-Time Exploiting Method for Low Latency NAND Flash-Based Storage Devices," *IEEE Trans. Computers*, vol. 63, no. 5, pp. 1085-1096, 2014.

[8] S. Lee et al., "BAGC: Buffer-Aware Garbage Collection for Flash-Based Storage Systems," *IEEE Trans. Computers*, vol. 62, no. 11, pp. 2141-2154, 2013.

[9] Y. Lee et al., "Zombie Chasing: Efficient Flash Management Considering Dirty Data in the Buffer Cache," *IEEE Trans. Computers*, no. 1, pp. 1, PrePrints, 2013.