2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00045

Accelerating Sub-Block Erase in 3D NAND Flash Memory

Hongbin Gong*, Zhirong Shen*, Jiwu Shu*†,
*Xiamen University, †Tsinghua University
23020201153744@stu.xmu.edu.cn, {shenzr,jwshu}@xmu.edu.cn

***Abstract*—3D flash memory removes scaling limitations of planar flash memory, yet it is still plagued by the tedious GC process due to the "big block problem". In this paper, we propose SpeedupGC, a framework that incorporates the characteristics of data updates into existing sub-block erase designs. The main idea of SpeedupGC is to guide the hotly-updated data to the blocks that are about to be erased, so as to speculatively produce more invalid pages and suppress the relocation overhead. We conduct extensive trace-driven experiments, showing that SpeedupGC can averagely reduce 64.7% of the GC latency, 21.8% of the read latency, 17.7% of the write latency, and 11.5% of the write amplification when compared to state-of-the-art designs.**

I. Introduction

Conventional 2D (or planar) NAND flash memory is now heavily plagued by the scaling limits to achieve the growth of storage capacity [18], as simply squeezing more information bits into a *flash cell* is hard to compensate the increasing raw bit error rate. As an alternative, *3D NAND flash memory* (also called 3D flash memory for short) paves a new way to break the scaling barrier of the planar flash memory through vertically stacking flash layers [13], [14]. While achieving larger storage capacity, the "big block" problem of the 3D flash memory exacerbates the GC efficiency [23]. Generally, when a GC operation is launched, the controller will select a *victim block* (i.e., the block selected to be erased), relocate the *valid pages* (i.e., the flash pages storing valid data) of this block, and finally erase the entire block to make room for future reusage. Hence, the *relocation latency* and the *erase latency* of the victim blocks collectively contribute to the overall GC latency. As the big block in a 3D flash memory probably contains a large number of valid pages, the relocation latency may be increased.

In view of this, the *sub-block erase* [8], [21], [22] is proposed for reducing the relocation latency during the GC operation. The *main idea* is to partition a block into multiple equal-sized *sub-blocks* and perform GC operations at the sub-block level, such that only those valid pages of the victim sub-blocks are required to be migrated (as opposed to migrating all the valid pages of the entire block in the conventional GC operations). We carefully examine existing sub-block erase designs and find that most of them mainly consider (i) the realization of the sub-block erase via electronic designs [8], [21], [22] and (ii) the selection of the appropriate victim sub-blocks for gaining high GC efficiency [5], [6], [19]. Nevertheless, how to leverage the characteristics of data updates to facilitate the efficiency of the sub-block erase is largely unexplored.

Our motivation is that storage systems often exhibit highly skewed access characteristics [2], [4]. Hence, by guiding the *hotly-updated data* (i.e., the data that are frequently updated in a short space of time) to the sub-blocks that are more likely to be erased, we can speculatively decrease the number of valid pages resided on the victim sub-blocks, hence shortening the relocation process (see Section II-B). We design SpeedupGC, a general framework that can incorporate the characteristics of data updates into existing sub-block erase designs. SpeedupGC first classifies the updated data and the flash blocks based on the hotness degrees and the *recycle benefits* [1] (i.e., the space that can be reclaimed per time unit), respectively. It then directs the data with larger hotness degrees to the sub-blocks that are believed to gain more recycle benefits. SpeedupGC also migrates the valid pages based on the hotness degrees of the corresponding data. To the best of our knowledge, SpeedupGC is the **first work** that incorporates the update characteristics into existing sub-block erase designs. To summarize, our contributions include:

- We analyze a wide range of real-world traces and demonstrate that the GC performance can be remarkably improved by taking the characteristics of data updates into consideration (Section II-B).
- We design SpeedupGC, a general framework that can seamlessly incorporate the characteristics of data updates into a variety of sub-block erase approaches. SpeedupGC carefully chooses the destination of the updated data based on both the hotness degrees and the recycle benefits of blocks (Section III).
- We implement SpeedupGC in the SSDsim simulator [11] and conduct extensive evaluations with real-world traces, showing that SpeedupGC can averagely decrease 64.7% of the GC latency, 21.8% of the read latency, 17.7% of the write latency, and 11.5% of the write amplification (Section IV).

II. Background and Motivations

A. Background

NAND flash memory: Fundamentally, information in NAND flash memory are recognized through the number of electrons

Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn). This work is supported by Natural Science Foundation of China (No. 62072381, 61832011), CCF-Tencent Open Fund WeBank Special Fund, CCF-Huawei Innovation Research Plan (CCF2021-admin-270-202102), Zhejiang Lab (2021KF0AB01), and Xiamen Youth Innovation Fund (3502Z20206052).

[1]Different sub-block erase designs [5], [6] have proposed different formulas to calculate the recycle benefits, while our approach works for them.

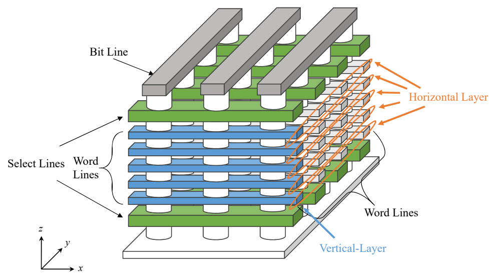
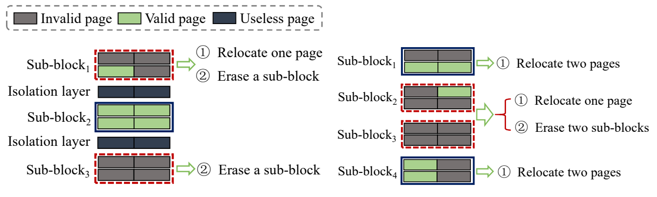


Fig. 1. Bird's eye view of a 3D flash block.



(a) Hardware isolation (b) Software isolation

Fig. 2. Examples of hardware isolation and software isolation. The sub-block with dashed lines (in red) denotes the victim sub-block.

trapped in the basic *flash cells* (i.e., physical floating gate transistors) [3]. Flash memory manipulates data via three basic operations: read, write, and erase. In particular, the read and write operations are performed in units of *flash pages* (made up of multiple cells), while the erase operations are launched on *flash blocks* (composed of multiple flash pages). A flash cell cannot be re-written directly. Flash memory adopts the *out-of-place update* manner [24], meaning that the newly updated data will be directed to the clean flash pages, while those storing the old data will be treated *invalid*. In this paper, we call the out-dated data (resp. page) as the invalid data (resp. page). Once the remaining available space reaches a pre-defined threshold (called "GC threshold"), the flash controller will reclaim the space occupied by the invalid data via the following steps: (i) selecting a victim block; (ii) relocating the residual valid pages of this block to other places; and (iii) erasing this entire victim block for future reuse.

3D NAND flash memory: 3D NAND flash memory [1], [7], [9], [20] is a new manufacturing technology that realizes the continuous capacity growth through stacking the flash layers in the vertical direction. Figure 1 illustrates the architecture of a 3D flash block, in which five horizontal layers (in orange) are vertically squeezed into the space between the two *select lines* (SLs, which are used to select the corresponding vertical layer for I/O requests). In this example, each horizontal layer comprises four *word lines* (WLs) and a WL further consists of multiple flash cells for data storage. Therefore, the 3D flash memory can multiply the flash cells piled up within the given area, and hence achieve remarkable storage capacity growth.

Sub-block erase: The "big block" in the 3D flash memory also suffers tedious GC processes, as a victim block is likely to have more valid pages to be migrated. To reduce the GC latency, the *sub-block erase* [8], [21], [22] is proposed to break a block into multiple fixed-size sub-blocks. It then performs GC operations at the sub-block level and hence only those valid pages of the victim sub-blocks need to be relocated. Erasing a number of sub-blocks within the same block still calls for almost the same time as erasing an entire block [6]. The sub-block erase also easily adds the interference to the adjacent horizontal layers [6] (Figure 1), as it may induce the charge leakage of the flash cells of them and arise the error occurrence. To alleviate the interference, existing sub-block erase designs mainly resort to two isolation methods, namely *hardware isolation* [6] and *software isolation* [5]. Specifically, the hardware isolation reserves two horizontal layers that sit right above and below each sub-block to serve as the *dedicated* isolation layers (i.e., without storing any data), such that the inference of erasing a sub-block can be absorbed by the corresponding isolation layers without affecting other sub-blocks. However, reserving dedicated isolation layers will introduce storage loss. Figure 2(a) shows an example of the hardware isolation, which reserves isolation layers (each with two pages) laid between two sequential sub-blocks. Hence, it can erase two victim sub-blocks (i.e., $\texttt{sub-block}_1$ and $\texttt{sub-block}_3$) without affecting the data of $\texttt{sub-block}_2$.

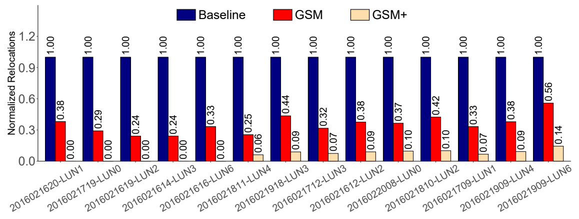
On the other hand, the *software isolation* proposes to *proactively* relocate the valid pages of the corresponding adjacent sub-blocks before erasing a given victim sub-block, such that the storage space can be fully utilized at the expense of additional relocation overhead. To suppress the relocation overhead, the software isolation further suggests selecting *sequential* victim sub-blocks that have the largest *recycle benefit* (defined as the average latency to reclaim an invalid page). Figure 2(b) shows an example of the sub-block erase with the software isolation, where two sequential sub-blocks (i.e., $\texttt{sub-block}_2$ and $\texttt{sub-block}_3$) that have the largest recycle benefit are selected as the victim ones. Besides migrating one remaining valid pages on $\texttt{sub-block}_2$, the system also has to relocate the four valid pages resided in the adjacent sub-blocks (i.e., $\texttt{sub-block}_1$ and $\texttt{sub-block}_4$) before performing the erase operation.

In this paper, we mainly focus on the sub-block erase with the software isolation. We also show that our approach works for the case with the hardware isolation (Section III-E).

B. Motivations

Our *observations* are two-fold. On one hand, existing sub-block erase approaches [5], [6], [8], [19], [21], [22] are most agnostic to the access characteristics, indicating that they choose victim sub-blocks directly based on the given footprints of the invalid pages. On the other hand, today's storage workloads often exhibit highly skewed access characteristics [2], [4]. Hence, how to leverage the skewed access characteristics to generate the footprints of the invalid pages, which favor existing sub-block erase designs, still receives limited attentions.

Our *motivation* is that we can *proactively* produce more invalid pages in the GC operations by directing the hotly-updated data to the sub-blocks that are more likely to be erased in the near future. By doing this, we can speculatively suppress


(a) Finding 1: GSM+ can suppress data relocation in GC operations.

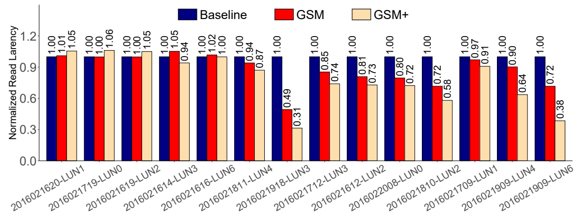

(b) Finding 2: GSM+ can reduce the overall read latency.

Fig. 3. Trace-driven analysis with hotness-aware updates.

the relocation latency, hence favoring the overall system performance. To demonstrate, we select 14 representative traces with different volumes of write sizes from a state-of-the-art trace repository [17]. We track the hotly-updated data using multiple independent hash functions and a multi-dimensional hash table [10].

We compare three approaches: the *Baseline* (which erases an entire block with the most invalid pages in a GC operation), GSM [5] (a sub-block erase approach with the software isolation), and GSM+. GSM+ is amended atop GSM, which guides hotly-updated data to the sub-blocks to be erased and applies the same sub-block erase strategy as GSM. We focus on the *read latency* and the *number of relocated valid pages in GC operations*, which can systematically characterise the access performance and endurance of the 3D flash memory. Figure 3 shows the results, and we make two findings.

Finding 1: *GSM+ can reduce 84.8% and 94.2% of valid page relocations on average compared to GSM and the Baseline, respectively (Figure 3(a)).*

Since the hotly-updated data identified are prone to be updated again in a short space of time, by writing the hotly-updated data to the sub-blocks that are likely to be erased in the near future, GSM+ can dramatically reduce the number of valid pages to be relocated in GC operations.

Finding 2: *GSM+ can reduce 12.4% and 21.5% of the read latency on average compared to GSM and the Baseline, respectively (Figure 3(b)).*

The rationale is that by reducing the number of valid pages being relocated, GSM+ can greatly shorten the GC latency, hence decreasing the overall latency for foreground reads.

III. SpeedupGC Design

We design SpeedupGC, a framework that incorporates the update hotness into the sub-block erase operations for improving the performance of 3D flash memory. SpeedupGC is a general design that works for both the hardware isolation and the software isolation. It is built atop the following design primitives: (i) a *hotness-aware update algorithm*, which directs the hotly-updated data to the sub-blocks that are more likely to be erased, so as to speculatively aggravate the space utilization of the victim sub-blocks; and (ii) a *hotness-aware relocation strategy*, which relocates the remaining valid pages of the victim sub-blocks based on their hotness degrees, so as to opportunistically reduce the number of valid pages relocated in next GC operations.

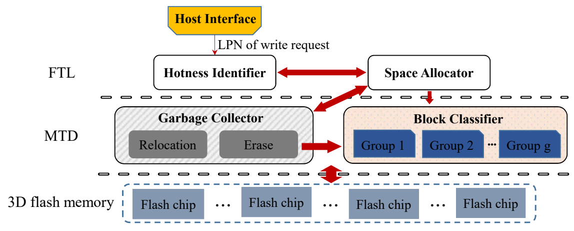


Fig. 4. The architecture of SpeedupGC.

A. Overview of SpeedupGC

Figure 4 depicts the system architecture of SpeedupGC, which is implemented across the *flash translation layer* (FTL) and *memory technology device* (MTD) layer. SpeedupGC comprises the following components: (i) a `hotness identifier`, which timely records the hotness degrees of the updated data; (ii) a `block classifier`, which classifies the blocks into groups based on their recycle benefits; (iii) a `space allocator`, which pinpoints the physical locations (including the block ID and the sub-block ID) for the incoming data, and (iv) a `garbage collector`, which periodically relocates the valid pages and reclaims the space.

We summarize the workflow of SpeedupGC as follows. When an update request arrives, the `hotness identifier` first gets the hotness degree of the updated data based on the corresponding *logical page number* (LPN). The `space allocator` then pinpoints the destination to relocate the updated data. When the number of free pages drops below a given GC threshold, the `garbage collector` chooses a block and relocates the remaining valid pages of the victim sub-blocks based on their hotness degrees. After the updated data is persisted to the flash memory, the `block classifier` recalculates the recycle benefit of the destination block.

B. Identifying Hotness and Classifying Blocks

Hotness identifier: We elaborate the functionalities of the `hotness identifier` and `block classifier`. Similar as the previous study [10], the `hotness identifier` of SpeedupGC also employs h independent hash functions (where $h \geq 1$) coupled with a multiple dimensional hash table, so as to record the hotly-updated data in a memory-efficient way. Hence, when an update request reaches, SpeedupGC hashes the corresponding *logical page number* (LPN) into h entries of the hash table, where the counter of each entry associated with the LPN is then increased by one. Given an LPN, SpeedupGC uses the *smallest* value of the associated h counters to serve as its hotness degree (see Figure 6).

Algorithm 1 Adjustment of block groups

```
Input: $\{\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_g\}$ ($g$ block groups)
       $m$ (the most blocks that a block group can contain)
       $B$ (a block that is updated)
       $r'$ (the updated recycle benefit of $B$)
Output: The updated block groups
1:  procedure MAIN
2:      // Pinpoint the block group
3:      Set $l = g$
4:      while $l >= 1$ do
5:          if $r' > r_l^*$ then
6:              break
7:              $l = l - 1$
8:          end if
9:      end while
10:     // Insert the block into the block group
11:     if $|\mathcal{BG}_l| == m$ then
12:         EVICT($\{\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_l\}$)
13:     end if
14:     Set $\mathcal{BG}_l = \mathcal{BG}_l \cup B$
15:     return $\{\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_l\}$
16: end procedure
17: // Evict a block to make room for the insertion
18: function EVICT($\{\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_l\}$)
19:     Set $B_l^* = \arg\min\{r_{l,j} | B_{l,j} \in \mathcal{BG}_l\}$
20:     Set $\mathcal{BG}_l = \mathcal{BG}_l - B_l^*$
21:     if $|\mathcal{BG}_{l-1}| == m$ then
22:         EVICT($\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_{l-1}$)
23:     end if
24:     Set $\mathcal{BG}_{l-1} = \mathcal{BG}_{l-1} \cup B_l^*$
25:     return $\{\mathcal{BG}_1, \mathcal{BG}_2, \cdots, \mathcal{BG}_l\}$
26: end function
```

The hotness degree of each LPN will decay by half after performing a given number of updates [10].

SpeedupGC classifies LPNs based on their hotness degrees. To lessen the metadata management overhead, SpeedupGC first defines a *threshold* φ ($\varphi \geq 1$), indicating that only the LPNs whose hotness degrees are larger than φ will be treated as hotly-updated. It then establishes g *hotness groups* (where $g > 1$), denoted by $\{\mathcal{HG}_1, \mathcal{HG}_2, \cdots, \mathcal{HG}_g\}$. Let $\{t_1, t_2, \cdots, t_g\}$ be the hotness thresholds for classifying the LPNs into g hotness groups, where $\varphi = t_1 < t_2 < \cdots < t_g$. Hence, an LPN with the hotness degree d can be classified into the hotness group $\mathcal{HG}_i$ once $t_i \leq d < t_{i+1}$ (where $1 \leq i \leq g-1$) or the hotness group $\mathcal{HG}_g$ once $t_g \leq d$.

Block classifier: Except classifying the hotly-updated LPNs, SpeedupGC also arranges the blocks into g *block groups* based on their recycle benefits. Suppose that there are n blocks in total. We first specify the maximum number of the blocks that a block group can contain as follows:

$$m = \left\lfloor \frac{n}{g-1} \right\rfloor - 1. \tag{1}$$

The configuration of m ensures that each of the g block groups will definitely contain at least a block [2]. We also

[2]We can easily prove it via contradiction. If there is a block group contains no block, then there must exist a block group that has more than m blocks, hence violating our requirement.

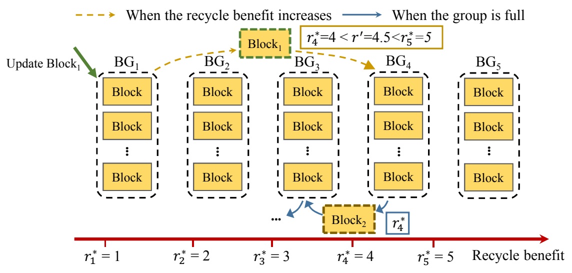


Fig. 5. Example of adjustment of block groups.

require $m \geq 2g - 3$, ensuring that all the n blocks can be organized into the g block groups (i.e., $m \times g \geq n$). At the very beginning, as all the blocks have the same recycle benefits, we can randomly assign them to the g block groups, ensuring that each block group contains no more than m blocks. With the update requests being served, the numbers of invalid pages of the blocks gradually differ and hence the recycle benefits of them get diverse. At this time, SpeedupGC will dynamically adjust the positions of the blocks, ensuring that each block group still has no more than m blocks. Algorithm 1 elaborates the detailed steps to find a position for inserting a block in the groups with smaller indices. If we cannot find such one, we can seek another one in the groups with larger indices using similar steps.

Details of Algorithm 1: For a block group $\mathcal{BG}_i$ (where $1 \leq i \leq g$), we use r_i^* to denote the *smallest* recycle benefit of the blocks in $\mathcal{BG}_i$. Let B denote a block that is updated and r' be its new recycle benefit after being updated. We first pinpoint the block group to which B should belong by finding a block group $\mathcal{BG}_l$ with the *largest* ID, such that $r' > r_l^*$ (Lines 2-9). SpeedupGC then checks if the number of blocks in BG_l (represented by $|\mathcal{BG}_l|$) is equal to m. If the block group BG_l is full, then SpeedupGC evicts the block whose has the smallest recycle benefit (i.e., r_l^*) in $\mathcal{BG}_l$ and inserts B into $\mathcal{BG}_l$ (Lines 10-14). For the block B_l^* (with the recycle benefit r_l^*) expelled from $\mathcal{BG}_l$, SpeedupGC will feed it in the block group $\mathcal{BG}_{l-1}$ (Lines 19-24). By doing this, Algorithm 1 ensures that the recycle benefit of any block in $\mathcal{BG}_i$ (where $2 \leq i \leq g$) is no smaller than that of any block in $\mathcal{BG}_{i-1}$.

Example: Figure 5 shows an example of the adjustment of block groups. Suppose that the $\texttt{Block}_1$ is updated and the corresponding recycle benefit changes to $r' = 4.5$. We insert $\texttt{Block}_1$ into the $\mathcal{BG}_4$ as $r_4^* < r' < r_5^*$. If $\mathcal{BG}_4$ is full, we evict the $\texttt{Block}_2$ with the recycle benefit r_4^* from $\mathcal{BG}_4$.

C. Hotness-Aware Update Algorithm

Besides classifying LPNs and blocks, SpeedupGC further devises a hotness-aware update algorithm, with the aim of speculatively reducing the number of valid pages to be relocated for the victim sub-blocks. Suppose that the data of the LPN in the hotness group $\mathcal{HG}_i$ (where $1 \leq i \leq g$) are updated, the hotness-aware update strategy will prioritize to direct the updated data to the blocks in the $\mathcal{BG}_i$, which has the same ID as the $\mathcal{HG}_i$. To this end, the hotness-aware update strategy should address the following two questions beforehand: (i)

Algorithm 2 Hotness-aware update algorithm

```
Input: an LPN with data updated
Output: the locations to store the updated data
1:  Get the hotness degree $d$ of the LPN
2:  // Check if the data of the LPN are hotly-updated
3:  if $d < \varphi$ then
4:      Write the data to a randomly selected block
5:      return complete
6:  end if
7:  Get the ID $i$ of the hotness group to which the LPN belongs
8:  for each block $B_{i,j} \in \mathcal{BG}_i$ do
9:      for each sub-block $S$ in the candidate victim sub-blocks do
10:         if $S$ has free pages then
11:             Write the updated data to $S$
12:             return complete
13:         end if
14:     end for
15: end for
16: for each block $B_{i,j} \in \mathcal{BG}_i$ do
17:     if $B_{i,j}$ has free pages then
18:         Write the updated data to any free page of $B_{i,j}$
19:         return complete
20:     end if
21: end for
22: Write data to any block of the neighboring block groups with
    free pages
23: return complete
```

which block in $\mathcal{BG}_i$ should be selected; and (ii) which explicit sub-block of the selected block should be chosen to keep the updated data. Conversely, if we cannot find the LPN in the hotness group (i.e., the data are cold as the hotness degree is smaller than φ), we choose to write the updated data in a randomly selected block.

Algorithm 2 shows the main procedures to direct the updated data. We first identify the LPN that the update data reside and also the associated hotness group. We then pinpoint the corresponding block group (Line 7) and scan each block in the selected block group. We always prioritize to write the hotly-updated data to a block of the designated block group, whose *candidate victim sub-blocks* (i.e., the victim sub-blocks currently identified if the sub-block is erased right now) still have free pages (Lines 8-15). If we cannot find such a block, then we turn to search *any* block that still has free pages in this block group (Lines 16-21). If all the blocks in this block group do not have any free page, then we will pick out any one with free pages from the neighbouring block groups (Line 22).

Example: Figure 6 shows an example of the hotness-aware update algorithm with four independent hash functions. Given an LPN whose data are updated, we first identify its hotness degree as five (i.e., the smallest counter of the LPN). We then pinpoint that the LPN resides in $\mathcal{HG}_2$ based on its hotness degree. Hence, we prioritize to direct the updated data to the block of $\mathcal{BG}_2$.

D. Hotness-Aware Relocation Strategy

When the ratio of free space is smaller than the given GC threshold, SpeedupGC will trigger the GC operation by selecting a number of victim sub-blocks for being erased within

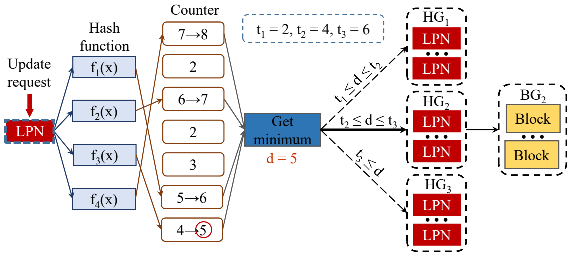


Fig. 6. Example of the hotness-aware update algorithm.

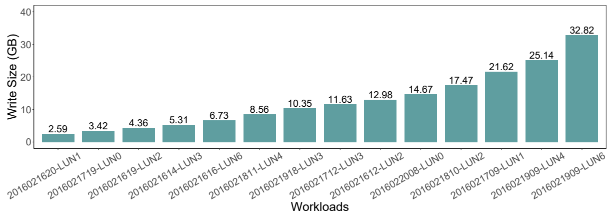


Fig. 7. Characteristic of the select traces [17].

a block. SpeedupGC devises a hotness-aware relocation strategy, whose *main idea* is to relocate the valid pages based on their hotness degrees, so as to opportunistically suppress the data relocations in next GC operations. Specifically, given a number of victim sub-blocks within the same block, we scan each valid page of them. We still rely on the `hotness identifier` to get the hotness degree and also the hotness group of the corresponding LPN. If the data of the valid page to be relocated are hotly-updated, we give higher priority to migrate them to another block within the block group that has the same ID as the hotness group. On the other hand, if the data to be relocated are cold, then SpeedupGC migrates them to a randomly selected block.

E. Analysis

Apply to hardware isolation: SpeedupGC can be effortlessly extended to the sub-block erase with the hardware isolation. Compare to the software isolation, the hardware isolation (e.g., the performance booster [6]) also differs in the calculation of the recycle benefit. For example, the performance booster [6] prioritizes to select the victim sub-blocks with more invalid pages. Hence, we can organize the blocks into groups based on the number of *invalid sub-blocks* (i.e., the sub-blocks with all invalid pages) they have. We then employ the hotness-aware update and relocation strategies to direct the updated data, and migrate the remaining valid pages.

Computational complexity: We first analyze the computational complexity of Algorithm 1. The complexity to pinpoint the block group (Lines 4-9 in Algorithm 1) is $O(g)$, where g is the number of groups. Let n be the number of blocks. Evicting a block from a group (Lines 18-26) takes the complexity of $O(n)$, as it may recursively scan each block in the worst case. So the computational complexity of Algorithm 1 is $O(n)$.

For Algorithm 2, it has to scan each block and the corresponding sub-block. Suppose that a block comprises s sub-blocks. As a block group contains no more than m blocks, the complexity is $O(ms)$.

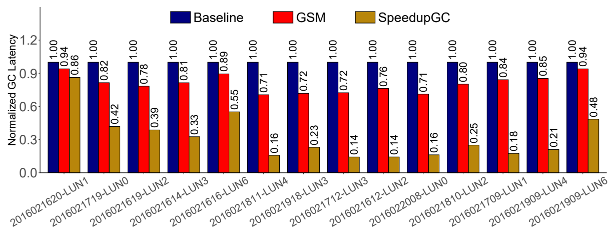


Fig. 8. Experiment 1 (GC latency).

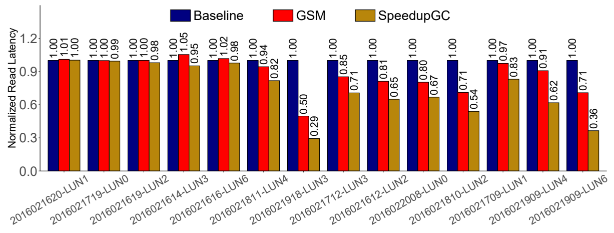


Fig. 9. Experiment 2 (Read latency).

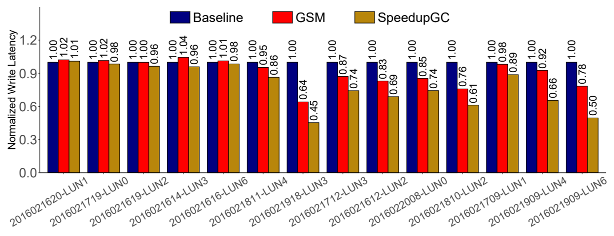


Fig. 10. Experiment 3 (Write latency).

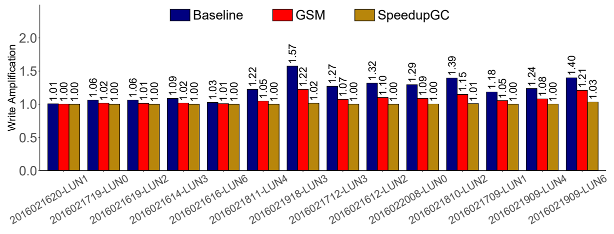


Fig. 11. Experiment 4 (Write amplification).

RAM overhead: SpeedupGC keeps a multiple dimensional hash table for tracking update hotness. It also maintains the information about the hotness group and block group. Based on the configurations borrowed from a real product [12] (see Section IV-A), we have measured that SpeedupGC merely induces 510.0 KB of RAM overhead for a 269 GB SSD.

IV. Performance Evaluation

We finally conduct extensive experiments to uncover the property, sensitivity, and generality of SpeedupGC. Our major findings are concluded as follows: (i) SpeedupGC can reduce 21.8% of the read latency, 17.7% of the write latency, 64.7% of the GC latency, and 11.5% of the write amplification on average (Section IV-B); (ii) SpeedupGC can sustain its advantages under different parameters (Section IV-C); and (iii) SpeedupGC is a general design that also works for the sub-block erase with the hardware isolation (Section IV-D).

A. Experimental Setup

Preparation: We implement SpeedupGC in the trace-driven simulator SSDsim [11]. We borrow the major configurations from a 64-stacked 3D NAND flash [12]. Specifically, the size of the flash memory is 269 GB with two channels, where each channel contains two chips with one die per chip. Each die is composed of 5,748 blocks and each block consists of 768 pages with 16 KB per page. We configure the program, read, and erase latencies as 700 μs, 60 μs and 3,500 μs, respectively [12]. Besides, we select seven independent hash functions for hotness degree maintenance and treat an LPN as hotly-updated if its hotness degree is no smaller than two (Section III-B). We separate the LPNs and the blocks into seven groups. We run our experiments on a server with Ubuntu 18.04.1 LTS, which is equipped with Intel Xeon CPU E3-1225 v6 (3.30GHz) and 16 GB RAM.

Default configurations: Unless otherwise specified, we select the following configurations throughout the evaluation. We set the number of sub-blocks that a block comprises to 16, which is also considered in previous studies [5], [6]. We set the GC threshold to 20%.

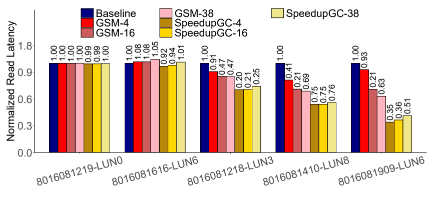
Counterparts: We compare SpeedupGC against another three erase approaches: (i) the Baseline (which erase a block in a GC operation), (ii) the *greedy scanning method* [5] (GSM, a sub-block erase with the software isolation), and (iii) the *performance booster* [6] (PB, a sub-block erase with the hardware isolation).

Methodology: We select 14 block-level I/O traces with different volumes of updated data from a state-of-the-art repository [17], which is collected from an enterprise virtual desktop infrastructure. To trigger GC operations, we warm up the flash memory via simply writing random data until the residual storage capacity is smaller than GC threshold. We then replay each trace, and record the average latencies to complete the read, write, and GC operations. For clear presentation, we normalize the experimental results (except the write amplification) to those of the Baseline.

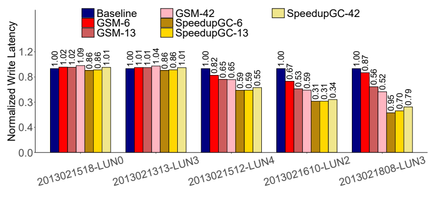
B. Experiments on Property

Exp# 1 (GC latency): We first compare the average GC latency, which comprises the erase latency and the relocation latency. Figure 8 shows that SpeedupGC can reduce 67.8% and 61.6% of GC latencies when compared to the Baseline and GSM, respectively. Such significant reductions are a result of SpeedupGC's advantage on mitigating the data relocation.

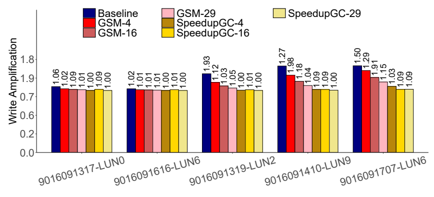
Exp# 2 (Read latency): We then assess the read latency. Figure 9 shows that SpeedupGC introduces the lowest latency, where it can cut down 25.9% and 17.6% of the read latency on average compared to the Baseline and GSM, respectively. The savings should attribute to the alleviation of the data relocations in the GC operations. We also observe that for some traces (e.g., `2016021719-LUN0` and `2016021619-LUN2`), the reduction of the read latency brought by SpeedupGC is trivial. We uncover that the root cause is the average time interval between two successive read requests in these traces is longer than the time taken in GC operations.



(a) Read latency.

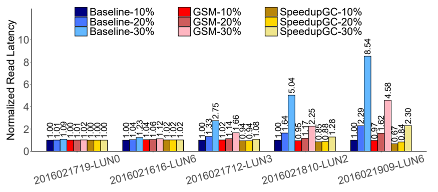


(b) Write latency.

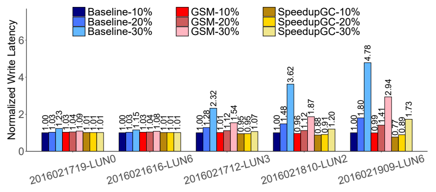


(c) Write amplification.

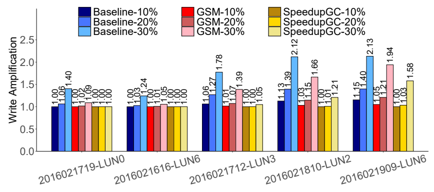
Fig. 12. Experiment 5 (Impact of number of sub-blocks within a block). The smaller value is better.



(a) Read latency.



(b) Write latency.



(c) Write amplification.

Fig. 13. Experiment 6 (Impact of GC threshold).

Exp# 3 (Write latency): We then evaluate the average write latency. Figure 10 shows that SpeedupGC can reduce 21.2% and 14.2% of the write latencies on average, when compared to the Baseline and GSM, respectively. The underlying reason is that SpeedupGC can significantly shorten the GC process by decreasing the number of valid pages being migrated and hence mitigate the write stalls.

Besides, the Baseline induces the highest write latency for most of the traces, as it always erases an entire block in a GC operations and needs to relocate the most valid pages.

Exp# 4 (Write amplification): We also measure the write amplification, which is defined as the ratio calculated via dividing the amount of data actually performed in the underlying flash device by that written from the host [19]. Hence, the write amplification is definitely no smaller than one (if we do not consider data deduplication and compression). Figure 11 shows that SpeedupGC reaches the smallest write amplification (i.e., 1.01 on average) among the three approaches, while the Baseline induces the largest one (i.e., 1.22 on average). The rationale lies in that by performing the GC operation at a finer granularity, SpeedupGC can avoid unnecessary data relocations in the GC operations. This experiment can also to some extent show that SpeedupGC can favor the endurance of the flash memory.

C. *Experiments on Sensitivity*

We then assess the sensitivity by varying the parameters using in the evaluation. We select five typical traces from Figure 7 with different volumes of data written.

Exp# 5 (Impact of number of sub-blocks in a block): We first evaluate the impact of the number of sub-blocks in a block, which is varied from 8 to 32. Figure 12 shows the read latency, the write latency, and the write amplification under different numbers of sub-blocks in a block.

We can make three findings. First, the read and write latencies of GSM both decrease with the number of sub-blocks comprised in a block, while those of SpeedupGC showcase an opposite trend. The root cause analysis shows that when the number of sub-blocks increases, GSM has more options to select the more appropriate victim sub-blocks with higher recycle benefit, hence favoring the read and write performance. On the other hand, SpeedupGC has already touched the lowest read and write latencies when a block is partitioned into eight blocks; it needs more computational time when the number of sub-blocks within a block increases, hence conversely calling for higher read and write latencies.

Second, the write amplifications in GSM and SpeedupGC both drop when the number of sub-blocks in a block increases. The reason is that the erase granularity is finer when a block is partitioned into more sub-blocks, and hence we can avoid more unneeded data relocations in GC operations.

Third, SpeedupGC reaches the lowest read and write latencies, and the smallest write amplification. To average, it reduces 23.4% of the read latency, 19.3% of the write latency, and 11.8% of the write amplification when compared to the Baseline and GSM.

Exp# 6 (Impact of GC threshold): We further uncover the impact of the GC threshold. We vary the GC threshold from 10% to 30%. Figure 13 shows the results.

Our observations are two-fold. First, the read and write latencies both increase with the GC threshold (Figure 13(a) and Figure 13(b)). We identify the reason is that when the GC threshold is larger, the system needs to migrate more valid pages (see Figure 13(c)), hence deteriorating the access performance. Second, SpeedupGC still causes the lowest read and write latencies, and the write amplification under different GC thresholds.

D. *Experiment on Generality*

Exp# 7 (Generality): We finally show that SpeedupGC also works for PB [6] (Section III-E). We select five traces from Figure 7 and measure the performance of the Baseline, PB, and SpeedupGC. Figure 14 shows that SpeedupGC reduces the read latency by -0.4-56.9%, the write latency by 0.5-45.1%,

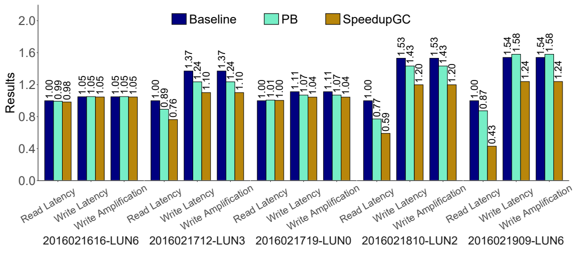


Fig. 14. Experiment 7 (Generality).

and the write amplification by 0.3-21.7% when compared to the Baseline and PB.

V. Related Work

The concept of partial erase had been proposed in the planar flash memory [15], [16] and begins to receive tremendous attentions when 3D flash memory is widely deployed.

Some studies focus on the realization of sub-block erase with electronic designs. D'Abreu et al. [8] propose to electrically isolate sub-blocks for enabling the sub-block erase without interference. However, it needs additional hardware support. Oh et al. [21] and Kim et al. [22] relieve the erase interference by reducing the voltage differences between the victim sub-blocks and their adjacent ones, but it may come with reliability flaws (e.g., incomplete erase or climbing raw bit error rates).

Some approaches also consider *how* to select the appropriate victim sub-blocks. Chen et al. [6] prioritize to select the ones with more invalid pages. However, it requires to reserve dedicated word lines to serve as the isolation layers for absorbing the erase interference, hence leading to storage loss. GSM [5] selects the sequential victim sub-blocks and migrating the valid pages resided on the adjacent sub-blocks to resist interference. It avoids storage loss at the cost of inducing additional data relocation. PEN [19] is an architectural design to enable partial-erase. It can adaptively find the optimal erase granularity for both the block-level FTL and hybrid FTL.

All above studies neglect how to leverage the characteristics of upper-layer's access to improve the efficiency of sub-block erase. SpeedupGC fills in this blank by carefully directing the hotly-updated data, so as to produce the footprints of invalid pages that favor the performance of the sub-block erase.

VI. Conclusion

This paper presents SpeedupGC, an efficient sub-block erase approach for 3D flash memory. The design principle is to classify the updated data based on their hotness degrees and separate the blocks based on their recycle benefits. It then carefully directs the updated data to the destination sub-blocks, so as to speculatively decrease the number of valid pages to be migrated in next GC operations. We implement SpeedupGC and conduct extensive performance evaluation with 14 real-world traces, demonstrating the effectiveness and generality of SpeedupGC.

References

[1] P. Arya. A Survey of 3D Nand Flash Memory. *EECS Int'l Graduate Program, National Chiao Tung University*, 11, 2012.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of A Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[3] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[4] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proc. of ACM SoCC*, 2010.

[5] H.-Y. Chang, C.-C. Ho, Y.-H. Chang, Y.-M. Chang, and T.-W. Kuo. How to Enable Software Isolation and Boost System Performance with Sub-block Erase over 3D Flash Memory. In *Proc. of IEEE/ACM/IFIP CODES+ISSS*, 2016.

[6] T.-Y. Chen, Y.-H. Chang, C.-C. Ho, and S.-H. Chen. Enabling Sub-Blocks Erase Management to Boost the Performance of 3D NAND Flash Memory. In *Proc. of ACM DAC*, 2016.

[7] J. Choi and K. S. Seol. 3D Approaches for Non-Volatile Memory. In *Proc. of 2011 Symposium on VLSI Technology-Digest of Technical Papers*, 2011.

[8] M. A. d'Abreu. Partial Block Erase for A Three Dimensional (3D) Memory, May 19 2015. US Patent 9,036,428.

[9] Y.-H. Hsiao, H.-T. Lue, T.-H. Hsu, K.-Y. Hsieh, and C.-Y. Lu. A Critical Examination of 3D Stackable NAND Flash Memory Architectures by Simulation Study of the Scaling Capability. In *Proc. of IEEE IMW*, 2010.

[10] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transactions on Storage*, 2(1):22–40, 2006.

[11] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *Proc. of ACM ICS*, 2011.

[12] C. Kim, D.-H. Kim, W. Jeong, H.-J. Kim, I. H. Park, H.-W. Park, J. Lee, J. Park, Y.-L. Ahn, J. Y. Lee, et al. A 512-Gb 3-b/Cell 64-Stacked WL 3-D-NAND Flash Memory. *IEEE Journal of Solid-State Circuits*, 53(1):124–133, 2017.

[13] J. Kim, A. J. Hong, S. M. Kim, K.-S. Shin, E. B. Song, Y. Hwang, F. Xiu, K. Galatsis, C. O. Chui, R. N. Candler, et al. A Stacked Memory Device on Logic 3D Technology for Ultra-High-Density Data Storage. *Nanotechnology*, 22(25):254006, 2011.

[14] J. Kim, A. J. Hong, S. M. Kim, E. B. Song, J. H. Park, J. Han, S. Choi, D. Jang, J.-T. Moon, and K. L. Wang. Novel Vertical-Stacked-Array-Transistor (VSAT) for Ultra-High-Density and Cost-Effective NAND Flash Memory Devices and SSD (Solid State Drive). In *Proc. of 2009 Symposium on VLSI Technology*, 2009.

[15] J.-k. Kim. Partial Block Erase Architecture for Flash Memory, Sept. 23 2014. US Patent 8,842,472.

[16] J.-K. Kim, H.-B. Pyeon, H. Oh, R. Schuetz, and P. Gillingham. Low Stress Program and Single Wordline Erase Schemes for NAND Flash Memory. In *Proc. of IEEE NVSMW*, 2007.

[17] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In *Proc. of ACM SYSTOR*, 2017.

[18] Y. Li and K. N. Quader. NAND Flash Memory: Challenges and Opportunities. *Computer*, 46(8):23–29, 2013.

[19] C.-y. Liu, J. Kotra, M. Jung, and M. Kandemir. PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs. In *Proc. of USENIX FAST*, 2018.

[20] A. Nitayama and H. Aochi. Vertical 3D NAND Flash Memory Technology. *ECS Transactions*, 41(7):15, 2011.

[21] E. C. Oh and J. Kong. Nonvolatile Memory Device and Sub-Block Managing Method thereof, Feb. 24 2015. US Patent 8,964,481.

[22] S. H. I. Se-Hyun Kim. Erasing Method of Non-Volatile Memory Device, 2015. US Patent 9,025,389.

[23] C. Sun, A. Soga, T. Onagi, K. Johguchi, and K. Takeuchi. A Workload-Aware-Design of 3D-NAND Flash Memory for Enterprise SSDs. In *Proc. of IEEE ISQED*, 2014.

[24] K. Zhou, S. Hu, P. Huang, and Y. Zhao. LX-SSD: Enhancing the Lifespan of NAND Flash-Based Memory via Recycling Invalid Pages. In *Proc. of IEEE MSST*, 2017.