A Program Context-Aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory

Keonsoo Ha
School of Computer Science and Engineering
Seoul National University
Seoul, 151-742, Korea
air21c@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
Seoul, 151-742, Korea
jihong@davinci.snu.ac.kr

Abstract

For NAND flash memory-based systems, garbage collection remains a major performance bottleneck. To decrease the garbage collection overhead, data separation techniques based on update frequency are widely used. However, from our observations using the oracle predictor on data update times, separating data by their update times rather than data with high update frequencies is a more important factor in reducing garbage collection overhead. Based on the observations, we propose a novel update time-based data separation technique. The proposed technique predicts what data will be updated together based on program contexts hints which can record data update behaviors. Our technique finds program contexts which generate data with similar update times, and groups the program contexts by estimated update times. A flash translation layer (FTL) using the proposed technique can reduce garbage collection overhead by allocating data with similar update times to the same blocks. Our experimental results show that our technique can reduce the total execution time of garbage collection on average 58% over a data update frequency-based approach.

1. Introduction

NAND flash memory is widely used as a storage device from embedded systems to high-end enterprise servers. Because of its many attractive characteristics for mobile storage devices such as light weight, low power consumption, durability, and high performance, it has been widely used for mobile embedded systems. As the cost per byte is falling while the storage capacity is increasing, large-capacity NAND flash memory devices such as solid state drives (SSDs) are more commonly employed for high-end desktops and enterprise storage servers.

Since NAND flash memory does not allow in-place update operations, garbage collection is necessary to find invalid blocks (whose data were replaced with more recent writes to different blocks) and erase them so that new data can be written into them. Since garbage collection involves slow erase operations and many read and write operations, which can be quite slow, high-performance NAND flash memory-based systems require an efficient garbage collection support. For example, most systems with NAND flash memory activate garbage collection during an idle state as a background process. Nevertheless, garbage collection can decrease the performance of NAND flash memory significantly (e.g., by about 20% [3]).

In order to minimize the garbage collection overhead, many techniques have been proposed [5]. Regardless of garbage collection algorithms used, moving valid data from selected victim blocks to new blocks during garbage collection takes a significant portion in the total execution time of a garbage collection algorithm. Therefore, reducing the total number of copied data from the victim blocks is a key factor in improving the performance of a garbage collection algorithm. To reduce the amount of copied data from the victim blocks, a common approach is to separate data based on their characteristics so that the number of dead blocks (which have no valid data) or near-dead blocks (which have few valid data) can be increased. The more dead or near-dead blocks are generated, the more likely that they can be selected as victim blocks during garbage collection, thus reducing the garbage collection overhead.

One of the most widely used data separation heuristics is to separate data by their update frequency. The basic idea of this data separator is to classify data based on their write temporal locality, and treat data with different temporal locality in a different fashion [2, 3]. This technique assumes that data with high write temporal locality are likely to be invalidated soon by successive update requests, thus the number of dead blocks can increase if data with high

locality are gathered in the same block. For example, the simplest version of this locality-based data separator divides data into two groups, *hot* data and *cold* data, based on the number of updates in a given time window. When data are written to NAND flash memory either by a write request or by a move request from a garbage collector, hot data and cold data are written in separate hot blocks and cold blocks, respectively. By gathering hot data into separate hot blocks, they are more likely to be dead blocks soon.

Although locality-based heuristics work reasonably well, it is not clear if knowing the relative frequency of block updates is sufficient or not in minimizing the garbage collection overhead. In order to better understand the performance of the existing data separator heuristic, we first introduce predictor on *future* update times, which has complete knowledge on future data update times. Using an FTL based on the oracle predictor as an *off-line* optimal FTL, we evaluate the performance of the existing data separation heuristic (based on the write temporal locality). The oracle predictor on data update times is denoted as ORA in this paper. The off-line optimal FTL is based on the idea that the garbage collection overhead is minimized when a garbage collector always selects dead blocks as victim blocks. The most obvious way to make more dead blocks is to predict future data update times and gather data with similar update times in the same blocks. Since the data in such blocks will be invalidated almost simultaneously, these blocks will become dead blocks quickly once the first data in the block is overwritten. Therefore, such blocks will not incur unnecessary page migration overhead from moving valid pages in victim blocks to new blocks during garbage collection. Clearly, the ORA is not implementable in practice because it requires the perfect information on *future* block update times. In this paper, we use the garbage collection overhead of an FTL based on ORA as an upper bound in evaluating garbage collection heuristics.

From our comparative study using ORA, we have observed that gathering data with similar future update times to the same blocks, not data with high update frequencies, is a more important factor in minimizing garbage collection overhead. We have found that data with similar update frequencies were not necessarily updated at similar times. For example, there is no clear correlation on their update times among hot data if they were classified as hot data at different times. If several hot data groups with different locality are stored in the same block, the probability that all data in that block are updated together is small because data with different locality have different update times. One of the main reasons of the poor performance of existing garbage collection heuristics can be attributed to the fact that they ignore data update times in devising their data separation techniques.

Based on our observations, we propose a novel data separation technique which predicts data update times by exploiting program contexts [6, 10] as hints. Our technique estimates what data will be updated together based on the data update history of the program context PC. Once data with similar future update times are predicted, the data are allocated into the same block by an FTL using the proposed technique, both when a write request is processed and valid data in a victim block are moved during garbage collection. In this paper, we assume that there is an appropriate interface between an operating system and an FTL to pass the program context information from the operating system to the FTL.

Conceptually, a program context represents one execution phase of a program. Since the program behaves similarly when the same phase is executed, program contexts can be used in predicting future block update patterns when a particular program context is identified with its previous block update history. For example, if a program context PC generates update requests R_1, R_2, and R_3, it is very likely that the same program context PC will generate the same update requests R_1, R_2, and R_3 again when the program context PC is re-executed. We can also group several program contexts into a set of inter-related program contexts where each member program context follows similar update request patterns with other member program contexts. Using the program context-based predictions, our technique identifies a group of data that will be updated at similar times.

In order to evaluate the proposed data separation technique, we have experimented using write traces collected from several programs. The experimental results show that our technique reduces the total execution time of garbage collection on average by 58% compared to a hash-based locality separation technique [7].

The rest of this paper is organized as follows. In Section 2, we show that predicting data update times is more important than estimating data locality in designing a high-performance garbage collector. We describe our proposed technique in Section 3. Section 4 presents the experimental results, and Section 5 summarizes related work. Finally, we conclude in Section 6.

2. Motivations

2.1 Garbage Collection Using ORA

If a garbage collector can choose a dead block as a victim block whenever a garbage collector is invoked, the total execution time of the garbage collection process is reduced to the total execution time of erase operations performed during garbage collection. Although it is not trivial to devise such an optimal garbage collector (even if the complete details of write requests are known *a priori*), if we know future block update times in advance, we can design a very

efficient garbage collector. In this section, we describe a garbage collection process based on ORA.

When a write request arrives, an FTL consults the ORA to get the future update time of the written data. Based on the future update time, the FTL allocates the requested data to the block where data with similar update times were already stored. During garbage collection, a garbage collector moves valid data in a victim block to the block with similar update times, using ORA. Although it is impossible to implement an on-line version of ORA in practice (because we cannot build such an oracle predictor on future block update times), ORA can be built off-line if we have a complete trace of write requests including their request times. In this paper, we implement an off-line version of ORA, which will be used as a data separator, in evaluating other data separation heuristics.

2.2 Evaluation of Existing Locality-based Heuristic

A locality-based data separator has been widely used in various FTLs. In particular, many researchers have proposed different data separation techniques that aim to increase the accuracy of data locality classification. For example, recently proposed techniques include 2-level LRU-based heuristic [3], hash table-based heuristic [7], and request size-based approach [2]. Since the hash table-based heuristic can accurately classify data with a small memory footprint and low time complexity, we use the hash table-based heuristic as a representative locality-based data separator. The hash table-based data separation heuristic is denoted as HASH in this paper. We also assume that a page-level mapping FTL is used in this paper. To evaluate different data separation techniques under the equal conditions, we use a garbage collector (except for a data separation technique) in the same page-level FTL. We use an FTL based on the cost-age-time heuristic [4] which takes account of the cleaning cost, erased counts, and the time elapsed (since the last modification) in selecting a victim block. (Unless confusion arises, we use ORA and HASH to indicate both data separation techniques and FTLs based them.)

Compared to the ORA algorithm, however, the HASH cannot achieve a high performance even if HASH can perfectly identify data update locality. For example, although hot data are clustered in the same block, if the update times are different among the hot data, the block may be half-dead, that is, some hot data in the block remain valid while other data in the same block are invalid. Such half-dead blocks significantly increase the amount of copied data during garbage collection.

Figures 1 and 2 show examples which illustrate a poor performance of HASH over ORA. In Figures 1 and 2, we

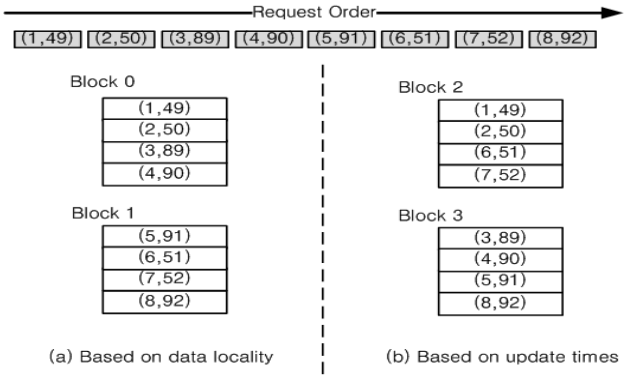


Figure 1. A comparison of data allocation using (a) HASH and (b) ORA

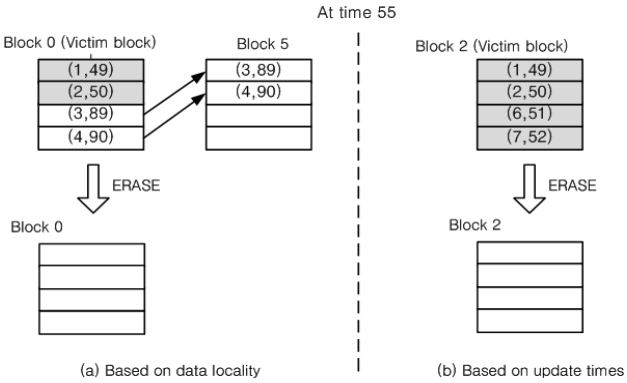
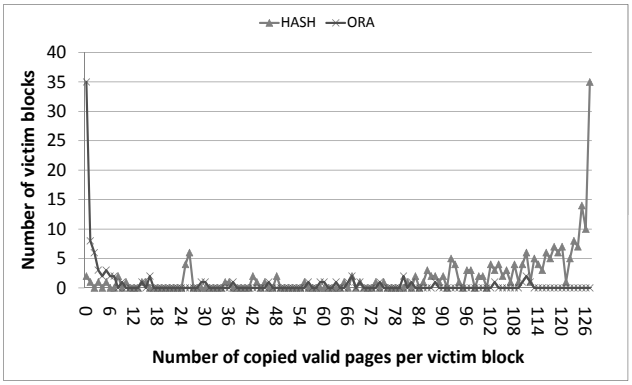
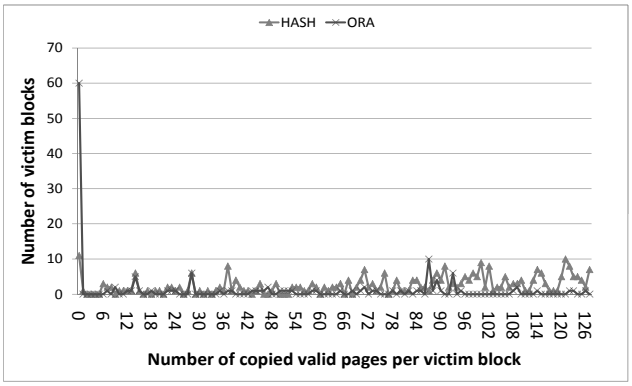


Figure 2. A snapshot comparison of garbage collection using (a) HASH and (b) ORA

assume that eight write requests, which are shown on top of Figure 1 as eight rectangles. Since we assume a page-level mapping FTL, each rectangle represents a page write request. A tuple (i, t_u) in a rectangle represents the *i-th* request with the next update time t_u. We further assume that data written by the write requests in the examples have been already classified as hot data. In Figure 1.(a), since all the data requests were classified as hot data by data locality, an FTL writes them to the same block according to the request order. On the other hand, in Figure 1.(b), an FTL using ORA allocates the requested data to Blocks 2 and 3 grouping requested page writes according to their future update times. Figure 2 shows snapshots of garbage collection using HASH and ORA. Assuming that garbage collection is invoked at time 55, in HASH as shown in Figure 2.(a), because bottom two pages in Block 0 remain valid at time 55, although all pages in Block 0 are full of hot data, the valid pages are moved to Block 5, which is a newly allocated block. Since these pages will not be invalidated until their



(a) gcc



(b) cscope

Figure 3. Distributions of the number of copied pages per victim block

respective update times, time 89 and time 90, they may be copied to other victim blocks several times whenever these pages belong to a victim block. In ORA as shown in Figure 2.(b), Block 2 has been changed to a dead block at time 52, hence Block 2 is reclaimed with only one erase operation.

In order to investigate the efficiency of a locality-based data separator in reducing garbage collection overhead, we have compared ORA and HASH using several benchmark programs. (For a more detailed description of the experimental setup, refer to Sec. 4.) In HASH, only 66.3% of hot data pages in hot blocks were updated in similar times, leaving about 34% of hot data pages still valid after other pages got invalidated. These valid pages produce a large number of half-dead blocks which can increase the garbage collection overhead when selected as victim blocks. Figure 3 shows distributions of the number of copied pages per victim block when HASH and ORA are applied in two programs used in our experiments. The X-axis and the Y-axis denote the numbers of victim blocks and copied valid pages per victim block, respectively. In Figures 3.(a) and 3.(b), HASH tends to copy more valid pages per victim block than ORA. Since these copied valid pages invoke more garbage collection processes, HASH may suffer poor performance of garbage collection. In our observations, HASH increases the total number of copied valid pages during garbage collection by 44 times over ORA. These results strongly suggest that a better data separation technique can be developed if data with similar future update times can be found.

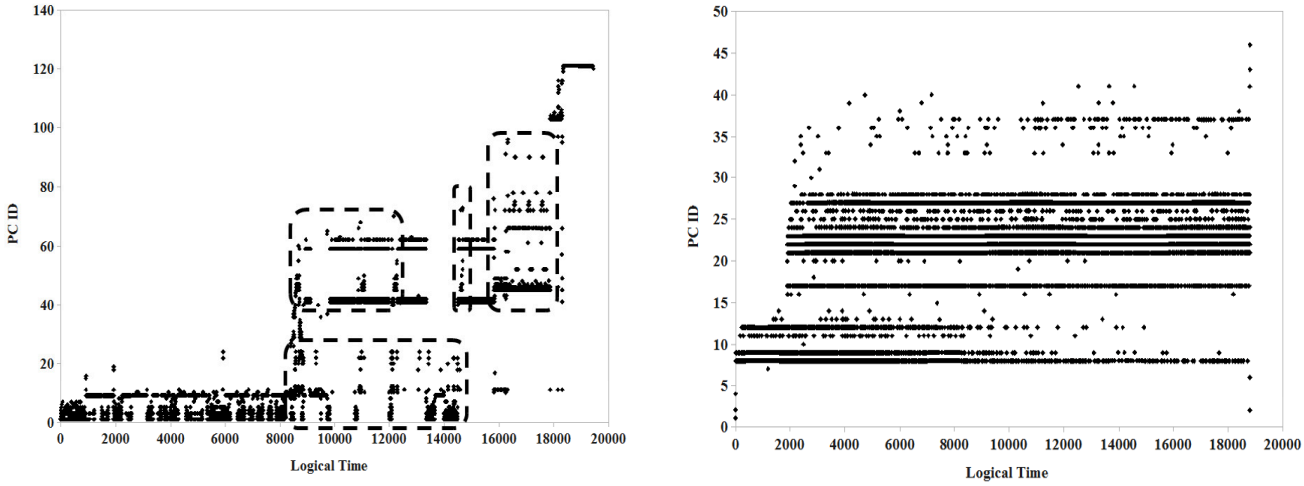
3 Program Context-Aware Data Separation Technique

3.1 Correlation between Program Contexts and Updates

Although taking account of update times of data is useful in reducing garbage collection overhead, it is impossible to know accurate update times before actual update requests occur. In this paper, we indirectly identify data which are updated together in similar time periods by exploiting typical update behavior of a program. We use program contexts to keep track of program's write/update behavior. Since a program context represents a program phase and the same phase is likely to be executed multiple times, program contexts have been used in predicting future behaviors of programs (e.g., [8, 9]). Since a program phase consists of consecutive instructions executed and the instructions can be specified with program counters, program counters can be used to distinguish different program contexts.

We use the *signature program counter* [6] to identify significant program contexts, which are distinguished by different program context (PC) IDs. Each PC is identified by summing program counter pc values of each execution path of function calls that lead to system calls which cause write requests. (To avoid confusions, we use a lowercase pc to indicate the program counter while the program context is denoted by using an uppercase PC.) For instance, a program can issue a write request through system functions such as write() and writev() in the Linux kernel. If the functions a(), b(), and c() were called in sequence before reaching the system functions, the pc values of these three functions can be obtained by a stack frame traversal when the write request is processed. The signature pc for this write request is computed by summing those three pc values. Although this traversal is carried out whenever a write request occurs, it is negligible in processing a write request because the additional overhead of getting a signature pc is 0.19 microseconds on a 2 Ghz Intel Pentium personal computer with 2 GB RAM [6], while performing a write operation in NAND flash memory requires several hundred microseconds. We assume that PC IDs are passed to an FTL through APIs between a file system and an FTL.

In order to evaluate the feasibility of using PCs in predicting data update patterns, we investigated the relation-



(a) cscope+gcc

(b) tpc-r

Figure 4. Distributions of program contexts in some benchmark programs

ship between updated data and their corresponding PCs (which have generated the updated data). Figure 4 shows distributions of PCs which have produced updated data requests in benchmark programs. A horizontal axis indicates the logical times which increase by one whenever data are updated. A vertical axis denotes distinct PCs which have generated the updated data requests at each logical time. For example, if data are written from a PC at logical time t, and the data are updated by new write requests from the PC at logical time $t + \alpha$, where α is a positive number, the PC is shown at logical time $t + \alpha$.

From our analysis, we have observed two key PC characteristics that can be used in designing our heuristic on data update times. First, a small number of PCs dominate. These dominating PCs generate repeatedly a large number of write requests, and the data from the dominating PCs are updated. Figures 4.(a) and 4.(b) show such cases; a few PCs produce repeatedly most of write requests, and the written data are updated in similar times. In the benchmarks used in our experiments, top five dominating PCs of each benchmark generate about 76% of updated data requests. Since data from these PCs tend to be updated consecutively, if an FTL allocates these data to the same blocks, it is likely that data from these PCs in the same blocks will be updated together. Second, data from non-dominating PCs are often highly correlated with dominating PCs, because consecutive update requests are often generated from several PCs. For example, in the dotted boxes in Figure 4.(a), data from non-dominating PCs which appeared infrequently are updated together with data from dominating PCs in similar time periods. In our observations, 63% of PCs which generate updated data requests are involved in sequential update patterns. If an FTL can find these PCs by checking update access patterns of PCs, the FTL can gather data from these PCs into the same blocks. Our observations suggest that PCs are closely related with updated data, which means that PCs can be used as important hints in estimating data updated in a similar time.

3.2 Overview of Program Context-Aware Data Separation Technique

We designed a program context-aware data separation technique based on the correlation between PCs and update requests. Figure 5 briefly explains how an FTL based on the proposed technique works. Our technique is used

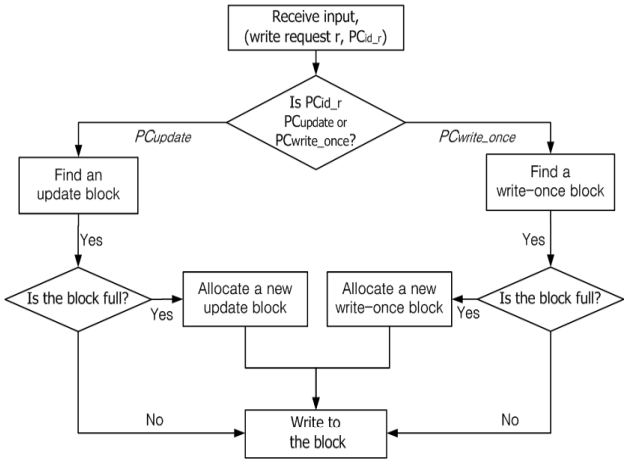


Figure 5. The flowchart of data allocation in an FTL using the proposed technique

both when a write request arrives from a host system and valid pages in a selected victim block are written to a new block during garbage collection. When a write request arrives in both situations, our technique checks the PC which has generated the current write request. If the proposed data separation technique determines that data request from the PC is likely to be updated later, the proposed technique predicts what data will be updated together with the requested data, and provides the update information to the FTL. If a PC generates a write request R and the data written by R are updated later, we call such a PC the *update PC*, denoted as PC_{update}. To allocate a page to the requested data, the FTL finds a block storing data that will be updated together with the requested data. A block is called an *update block* if its data are updated later. If the block is write-once, we call such a block *write-once block*. For a write request from an update PC, we store the requested data to an update block with similar estimated update times. If our technique predicts that the requested data will not be updated based on the PC which has generated the request, the data are written to a write-once block. If a PC generates write requests, and if the written data are not updated later, such a PC is called *write-once PC*, denoted as PC_{write_once}.

Figure 6 shows an example of data allocation based on our technique using the same example in Figure 1. Each write request tuple of Figure 1 is modified to include the PC identifier id_{PC} so that each write request (i, id_{PC}) indicates that the *i-th* write request has been generated from the PC id_{PC}. The id_{PC} of each request is sent to an FTL from a host system whenever a write request occurs. In this example, id_{PC} was computed to 5 by summing pcs of functions a() and b(), and it is delivered to an FTL. Assume that first seven write requests have been processed. When a write request arrives, the FTL searches the program context information table with the requested id_{PC}, 5, to predict whether the requested data will be updated or not. The program context information table stores update information of previously identified PC_{update}s to find data with similar estimated update times. A row in the table stores a tuple $(id_{PC}, id_{UG}, N_{update})$, which means that PC id_{PC} generated N_{update} write requests which were updated in similar time periods together with data from the PC_{update}s in an update group UG, whose identifier is id_{UG}. An update group UG is a group of PC_{update}s that are expected to be updated in a similar time period. (We will describe how a tuple is created in the table in the next subsection.) Since the PC whose id_{PC} is 5 exists in the table, our technique decides that data written by this request will be updated together with data from PC_{update}s in the UG whose identifier id_{UG} is 1, thus the FTL finds an update block for this UG to store the requested data. To keep track of update blocks allocated to UGs, an update block information table records an update block number which stores data from PC_{update}s

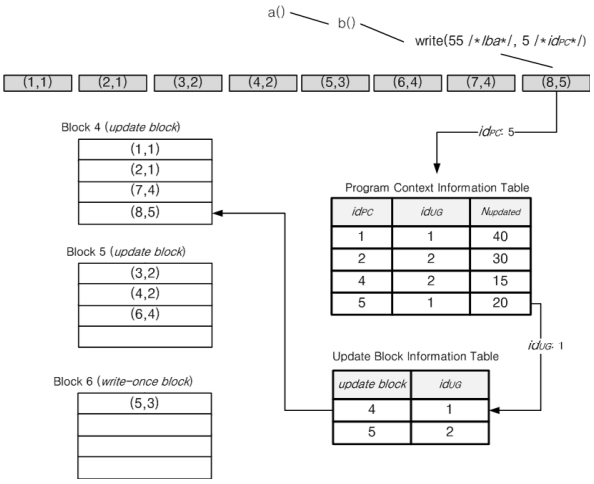


Figure 6. An example of data allocation using the proposed data separation technique

in an UG. Since many update blocks can be used by an UG, the update block number used most recently is stored with id_{UG}. Since this table indicates that data from the PC_{update}s in the UG whose id_{UG} is 1 are stored in Block 4, the requested data are written to Block 4. Prior write requests have been processed in the same way except for the fifth write request. Since the id_{PC} 3 does not exist in the program context information table, our technique regarded the PC whose id_{PC} is 3 as a write-once PC, thus the FTL wrote the data to Block 6, which is a write-once block.

In this way, the FTL gathers data from PC_{update}s in the same UG to the same update blocks. However, if an update block is selected as a victim block before all the pages in the block are invalidated, many read and write operations may occur during garbage collection. To avoid this situation, the FTL gives the lowest priority to update blocks when a victim block is chosen. By allowing enough time to update blocks, data in the update block are very likely to be invalidated before it is selected as a victim block.

3.3 Program Contexts Grouping Algorithm

Based on the relationship between PCs and updated data, our technique groups PCs which generate data with similar estimated update times into the same UGs. As mentioned in Sec. 3.1, since a small number of PCs generate repeatedly many write requests which tend to be updated consecutively, our technique finds a *dominating* PC, PC_d, and inserts the PC_d to an UG. Moreover, our technique finds a *sequential* PC, PC_{seq}, which repeatedly generates updated data requests forming sequential update access pat-

Algorithm 1 Program Context Grouping Algorithm

Input : id_{PC}, lba
Output : id_{UG}

```
1:  I_pc←program_context_information_table(id_PC)
2:  if I_pc = NULL then
3:      I_PC.id_PC ← id_PC
4:      I_PC.id_UG ← NULL
5:      I_PC.N_update ← 1
6:      insert_tuple(I_PC)
7:  else
8:      I_pc.N_update←I_pc.N_update+1
9:      if I_pc.id_UG = NULL then
10:         if (I_pc.N_update≥threshold) then
11:             N_UG←N_UG + 1
12:             I_pc.id_UG←N_UG
13:         else
14:             if ((prev_lba + size_of_page)= lba)
                && (prev_UG≠NULL) then
15:                 I_pc.id_UG←prev_UG
16:             end if
17:         end if
18:     end if
19: end if
20: prev_lba ← lba
21: prev_UG ← I_pc.id_UG
22: return I_pc.id_UG
```

terns with data from other PC_ds, and inserts the PC_{seq} to the UG including the PC_ds.

Algorithm1 describes how our technique groups update PCs into an UG. This algorithm takes as input PC ID id_{PC} and logical block address lba, and returns UG ID id_{UG}. Note that the id_{PC} in this algorithm is not the identifier of the PC which generates current data request, but the identifier of the PC which has generated the data updated by the current write request. Since the objective of this algorithm is to find update PCs which generate data expected to be updated together and to group into an UG, only the PCs of the updated data are considered. Whenever an update write request occurs, this algorithm searches the program context information table with the id_{PC} to check whether the PC has been determined as an update PC or not (line 1). If there is no update information of the PC, which is denoted as I_{PC}, a new tuple is created and its id_{PC}, id_{UG}, and N_{update} are set (lines 3-5). Since the data from the PC is updated for the first time, UG and N_{update} are set to NULL and 1, respectively, and this tuple is inserted into the table (line 6). The size of the memory required for keeping the program context information table depends on how many PCs in a program are related to update requests. For the applications used in our experiments, update requests are generated from 48 to 89 PCs.

If the I_{PC} for the PC exists, the number of updates increments by one (line 8), because data from the PC are updated. To find out whether the PC has been included in an UG or not, this algorithm checks the UG information of the PC (line 9). If an UG including the PC does not exist, this algorithm decides either to create a new UG for the PC or insert the PC to one of existing UGs based on N_{update} and update request pattern. If N_{update} of the PC is greater than a predefined threshold value, this algorithm regards the PC as a dominating PC, thus making a new UG and including the current PC to the new UG (lines 11-12). If N_{update} is less than the predefined threshold value, our technique checks the current PC is a sequential PC or not. If the current update request forms sequential update patterns with data updated previously, and if the previous data are from an update PC included in an UG, our technique inserts the PC into the same UG (lines 14-15). For the determination, the lba and id_{UG} previously processed in this algorithm are stored (lines 20-21). By returning the id_{UG} to an FTL, proper blocks can be allocated to write requests.

4 Experiments

To evaluate the proposed technique, we used a trace-driven FTL simulator. The FTL in the simulator triggers garbage collection if the total number of remaining blocks in NAND flash memory is less than 5% of the total number of blocks in NAND flash memory, and the garbage collection process is finished if 15% of the total number of blocks in NAND flash memory are reclaimed. Since garbage collection does not occur if there is enough free space to write new data, we filled 90% of the NAND flash memory space with meaningless values, before each experiment is performed. Respective latencies of read, write, and erase operation are 25μs, 200μs, and 1.2ms [1]. HASH and ORA are used for comparisons, and our technique is denoted as $PC\text{-}aware$ in our experiments.

In order to generate input traces for our simulator, we used four programs and two application sets as shown in Table 1. Although most applications used in our experiments do not generate many update write requests, they are enough to evaluate garbage collection overhead because they trigger a large number of garbage collection processes in NAND flash memory which is almost fully filled with data. For example, `gcc` is known as a CPU-bound application. However, it creates many object files and a ker-

Benchmarks	Scenario	N_{write}	N_{update}
cscope	Linux source code examination	17575	15398
gcc	Building Linux Kernel	10394	3840
viewperf	Performance measurement	7003	119
tpc-h	Accesses to database	23522	20910
tpc-r	Accesses to database	21897	18803
multi1	cscope+gcc	28400	19428
multi2	cscope+gcc+viewperf	35719	20106

N_{write}: the number of write requests (unit : page)
N_{update}: the number of update write requests (unit: page)

Table 1. Summary of various benchmarks

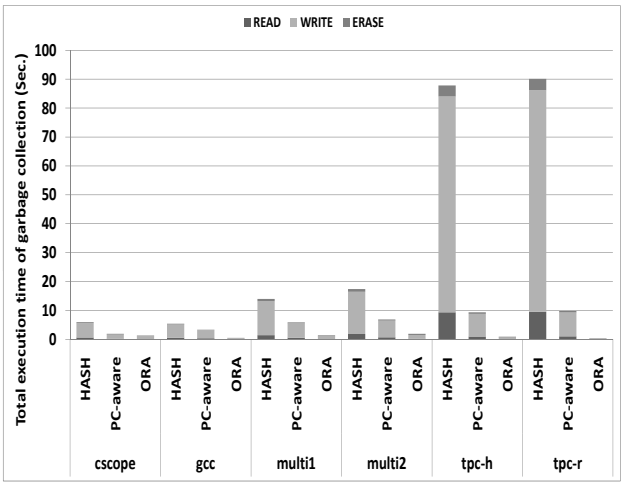


Figure 7. Total execution times of garbage collection in various traces

Benchmarks	Separator	N_{dead}	N_{avg_cped}	N_{erase}
cscope	HASH	1.0	125.4	318
	PC-aware	42.3	60.6	189
	ORA	61.0	35.8	199
gcc	HASH	0.0	108.2	225
	PC-aware	0.2	105.6	154
	ORA	8.0	25.1	94
multi1	HASH	0.0	110.5	655
	PC-aware	39.8	88.7	303
	ORA	77.0	24.6	280
multi2	HASH	0.0	109.3	793
	PC-aware	34.7	89.1	345
	ORA	83.0	30.4	325
tpc-h	HASH	0.0	123.4	3178
	PC-aware	0.3	90.0	385
	ORA	144.0	26.9	250
tpc-r	HASH	0.0	123.8	3235
	PC-aware	0.2	93.6	423
	ORA	145.0	26.2	165
arithmetic average	HASH	0.2	116.8	1400.7
	PC-aware	19.6	87.9	299.8
	ORA	86.3	28.2	218.8

Table 2. A comparison of the proposed technique over HASH and ORA for garbage collection overhead

nel image file while it compiles Linux kernel source in our experiment.In the case of `tpc-h`, it is a read-dominant workload with a small number of update requests, which come from concurrent data modifications. Nevertheless, it causes a large number of garbage collection processes because we start with almost full NAND flash memory.In the case of `viewperf`, since it does not have a lot of update requests, we performed experiments with `multi2` combining `viewperf` with `cscope` and `gcc`.

Figure 7 shows the total execution time of garbage collection where each data separation technique is applied. The X-axis and the Y-axis denote data separation techniques and the total execution time of garbage collection, respectively. The results show that our technique reduces the total execution time of garbage collection on average 58% over HASH. Since HASH determines data locality based on data update frequency in a given time window, if data are not updated frequently in the time window, HASH does not work well. In the cases of `tpc-h` and `tpc-r`, since they have relatively random write patterns over other programs, it is unlikely that data are written again and again in the same time window.

In order to understand why our technique outperforms HASH and how our technique is compared to ORA, we compared the number of dead blocks, the average number of copied pages per victim block, and the number of erased blocks. Table 2 shows the total number N_{dead} of dead blocks generated, the average number N_{avg_cped} of copied pages per victim block, and the total number N_{erase} of erased blocks when each data separation technique is applied.Compared to HASH, our technique generates more dead blocks, and reduces valid page copies per victim block by 25%. Moreover, HASH erases on average about five times more blocks than our technique. These results clearly mean that predicting data to be updated together with program contexts hints is effective in reducing garbage collection overhead. On the other hand, ORA produces about four times more dead blocks than our technique. Besides the number of generated dead blocks, our technique copies about three times more valid pages per victim block than ORA. ORA erases on average 27% less blocks than our technique. Although ORA works based on off-line analysis, these results strongly suggest that there may be a room for improving our technique further.

Figure 8 shows the normalized garbage collection overhead under different threshold values of Algorithm 1. The X-axis and the Y-axis denote various thresholds in each program and normalized total execution time of garbage collection, respectively. The results of each program are normalized to the worst case of each program. As shown in Figure 8, each program has different threshold values which cause the smallest execution time of garbage collection. Since a combination of PCs included in each UG can be changed by the threshold, the influence of the threshold on garbage collection overhead is noticeable. If the threshold is small, most of PCs which have produced updated data may be classified as update PCs, thus more UGs are created. If the PCs classified as update PCs generate many write requests, and the written data are updated later, the proposed technique works fine. On the contrary, if data from the PCs are not updated later, the data cause more extra copies and trigger more garbage collection. In the case of `gcc`, since PCs classified early as update PCs generate many write requests, and most of the data are updated later, `gcc` has the minimal total execution time when a threshold is 1. A large

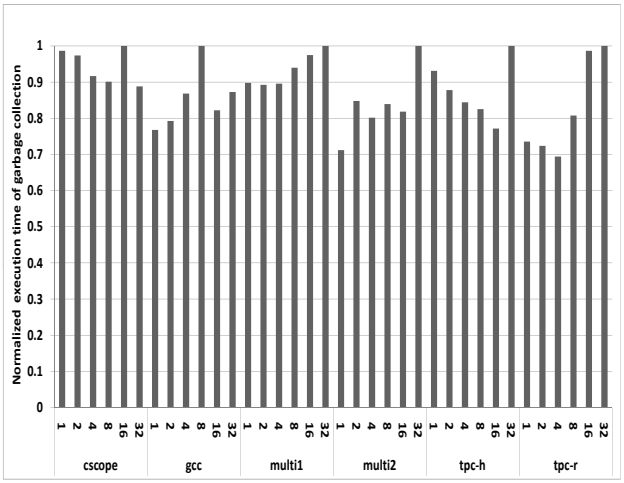


Figure 8. Normalized execution times of garbage collection under different threshold values

threshold may suppress creating UGs, because PC_{update}s which issue a lot of updated data requests can be inserted to an UG. In the case of `tpc-h`, by preventing PCs with a small number of updated write requests from being classified as update PCs, the total execution time of garbage collection is minimal when a threshold is 16.

5 Related Work

Data separation techniques have been studied to reduce garbage collection overhead. Chang et al. [3] suggested two-level LRU lists for hot-cold identification. The first level list stores logical block addresses of data written twice or more in a short period, while the second level list stores logical block addresses which have been accessed once in recent time or have been evicted from the first level list. Chang [2] proposed a size-based prediction technique. Based on the observation that small-sized requests tend to be accessed frequently, this technique classifies data locality according to sizes of requested data. Hsieh et al. [7] suggested a hash table-based data separation technique. This technique keeps track of the number of all write requests on each logical block addresses, and records the access information in a table. To reduce size of the table, this technique adopts multiple hash function which makes several logical block addresses share an entry in the table. Since this technique periodically divides the number of written counts of data by 2, it is likely that frequently updated data in recent times are considered as hot. The objective of these techniques is to accurately identify data. However, although data have similar write temporal locality, if they are updated in different times, frequency-based approaches are not effective in reducing garbage collection overhead. Moreover, these techniques do not work if there is no distinct locality in given data.

6 Conclusions

We have proposed a novel program contexts aware data separation technique. Taking account of the correlation between program contexts and update write patterns, the proposed technique predicts update times of data by examining program contexts which have produced the write requests. By gathering the data with similar update times to the same blocks, an FTL based on our technique can reduce total execution time of garbage collection operations by 58% over a hash-based hot/cold separation scheme. These results prove that predicting update times of data with program contexts can be a remarkable method in data separation technique.

Our work can be extended to several directions. For example, we plan to extend our technique for efficient wear-leveling management. Using the proposed technique, an FTL may identify cold pages more efficiently based on *future* update time information.

Our technique can be also extended for hybrid mapping-based FTLs. As well as update times of data, write patterns such as random and sequential are also important factors in reducing garbage collection overhead of hybrid mapping-based FTLs. In addition to estimating data update times, we plan to investigate the correlation between program contexts and write access patterns.The program context-aware approach may maximize the advantages of hybrid mapping-based FTLs such as the small sized memory footprint and high performance with sequential accesses by separating the sequential and random access patterns.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20100018873, No. R33-2010-10095, No. 2010-0020724). This work was also supported by the Brain Korea 21 Project in 2011. The ICT at Seoul National University provided research facilities for this study.

References

[1] Samsung Electronics, 2Gx8 bit NAND Flash Memory (K9WAG08U1A), Data sheet, 2006.

[2] L. Chang. Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 428–433, 2008.

[3] L. Chang and T. Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.

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

[5] T. S. Chung, D. J. Park, S. Park, D. H. Lee, S. W. Lee, and H. J. Song. System software for flash memory: a survey. *Embedded and Ubiquitous Computing*, 4096:394–404, 2006.

[6] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of Symposium on Opearting Systems Design and Implementation*, pages 395–408, 2004.

[7] J. Hsieh, L. Chang, and T. Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of ACM Symposium on Applied Computing*, pages 838–842, 2005.

[8] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of Annual International Symposium on Computer Architecture*, pages 139–148, 2000.

[9] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of Annual ACM/IEEE International Symposium on Microarchitecture*, pages 42–53, 2000.

[10] F. Zhou, R. von Behren, and E. Brewer. AMP: program context specific buffer caching. In *Proceedings of USENIX Annual Technical Conference*, pages 371–374, 2005.