Zombie Chasing: Efficient Flash Management Considering Dirty Data in the Buffer Cache

Youngjae Lee, Jin-Soo Kim, *Member*, *IEEE*, Sang-Won Lee, and Seungryoul Maeng

Abstract—This paper presents a novel technique, called *Zombie Chasing*, for efficient flash management in solid state drives (SSDs). Due to the unique characteristics of NAND flash memory, SSDs need to accurately understand the liveness of the data stored in themselves. Recently, the TRIM command has been introduced to notify SSDs of dead data caused by file deletions, which otherwise could not be tracked by SSDs. This paper goes one step further and proposes a new liveness state, called the *zombie* state, to denote live data that will be dead shortly due to the corresponding dirty data in the buffer cache. We also devise new zombie-aware garbage collection algorithms which utilize the information about such zombie data inside SSDs. To evaluate Zombie Chasing, we implement zombie-aware garbage collection algorithms in the prototype SSD and modify the Linux kernel and the Oracle DBMS to deliver the information on the zombie data to the prototype SSD. Through comprehensive evaluations using our in-house micro-benchmark and the TPC-C benchmark, we observe that Zombie Chasing improves SSD performance effectively by reducing garbage collection overhead. Especially, our evaluation with the TPC-C benchmark on the Oracle DBMS shows that Zombie Chasing enhances the Transactions Per Second (TPS) value by up to 22% with negligible overhead.

Index Terms—Solid state drive (SSD), NAND flash memory, flash translation layer (FTL), data liveness, operating systems

1 Introduction

Data stored in the persistent storage can be categorized into either *live* data or *dead* data. Live data is the actual data needed by user applications, which should be preserved at all costs. On the other hand, dead data is the one that is no longer accessible by user applications. Any live data becomes dead when a new version of the data is written into the storage or a file containing the data is deleted. Dead data can be discarded from the storage system at any time.

Such data liveness information makes various optimizations possible within the storage system. For example, the on-disk data layout can be improved by collecting live data together to minimize the number of seek operations of the disk head [1]. Prefetching can be performed more intelligently by caching only the live data inside the disks. Also, the secure delete operation which makes deleted data irrecoverable is only possible when information on data liveness is available in storage systems [2], [3].

In particular, data liveness information plays an important role in the performance of NAND flash-based solid state drives (SSDs). This is because NAND flash memory has several unique characteristics that distinguish it from rotating magnetic platters of hard disk drives (HDDs). Most notably, NAND flash memory does not allow in-place updates; the previous data should be erased before another data is written into the same area. To make matters worse, the unit of erase operation, called the *flash block*, is much larger in size than the read and write unit.

To support the traditional block device interface over NAND flash memory, SSDs commonly employ a special firmware layer called the *flash translation layer (FTL)*. One of the main functionalities of FTL is to handle write requests efficiently, while concealing the erase operation of NAND flash memory. For a given write request from the host, FTL redirects the write request to a previously erased area in NAND flash memory and maintains the mapping information between the logical address and the physical address of the data. As a result of the new write request, the previous data corresponding to the logical address becomes dead and FTL marks the physical area occupied by the dead data as being invalid. Those invalid areas are reclaimed later by FTL via the procedure called *garbage collection (GC)*.

When the number of pre-erased flash blocks falls below a certain threshold, FTL invokes the garbage collection procedure. The goal of garbage collection is to generate clean flash blocks by erasing invalid areas containing dead data. Among the flash blocks, the garbage collection procedure first chooses a victim block which is considered the best to be erased. If the victim block is filled entirely with dead data, it is simply erased and then converted to a clean flash block. Otherwise, the live data within the victim block is copied into another flash block before the victim block is erased. The extra copy operation needed to prevent the live data from being erased during garbage collection is a major source of management overhead in FTL.

To increase the efficiency of garbage collection in SSDs, it is essential to reduce the amount of live data as much as

- *Y. Lee, J.-S. Kim and S.-W. Lee are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Gyeonggi-do 440-746, Republic of Korea. E-mail: yjlee@csl.skku.edu, {jinsookim, swlee}@skku.edu.*
- *S. Maeng is with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, Republic of Korea. E-mail: maeng@camars.kaist.ac.kr.*

Manuscript received 31 Oct. 2012; revised 19 Aug. 2013; accepted 04 Nov. 2013. Date of publication 19 Nov. 2013; date of current version 16 Jan. 2015. Recommended for acceptance by R. Marculescu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.218

possible. Unfortunately, the narrow block device interface, such as SCSI, SAS, or SATA, makes it difficult to identify the live data accurately. An early problem was that data which became dead due to file deletion could not be tracked as SSDs were not aware of file delete operations issued by file systems. Consequently, the data belonging to the deleted file is wrongly considered to be alive and it is unnecessarily copied to another flash block to survive garbage collection.

To address this problem, the TRIM command has been recently introduced as a new standard SATA command for SSDs. When the file system deletes a file, the TRIM command explicitly informs SSDs of the corresponding locations of the deleted data so that they can be treated as the dead data. The TRIM command is known to be very effective in improving the performance of SSDs by eliminating unnecessary copy operations for the deleted data during garbage collection [4]-[6]. The use of the TRIM command is an effort to pinpoint *real* live data inside SSDs.

This paper goes one step further and introduces a new data liveness state called the *zombie* state. We focus on the fact that the data produced by user applications is temporarily placed in the buffer cache for performance reasons. The cached new data, which is usually called the *dirty data*, will be written into SSDs shortly. Assume that the dirty data is overwriting the previous version of the data stored in SSDs. In this case, the live data in SSDs corresponding to the dirty data will become dead when the dirty data is flushed from the buffer cache. We call such live data that is destined to be dead soon, due to the dirty data in the buffer cache, *zombie data*.

In this paper, we propose a novel technique called *Zombie Chasing* for efficient flash memory management in SSDs. The key idea of Zombie Chasing is to chase the zombie data in SSDs and to treat them differently from other live data. Distinguishing the zombie data from the real live data opens up new optimization opportunities for flash memory management. To the best of our knowledge, this work is the first study to optimize SSDs by considering the dirty data in the buffer cache.

More specifically, we present zombie-aware garbage collection algorithms which take into account the zombie data in two ways. First, we use the zombie data information when we select a victim block during garbage collection. Since we know that the zombie data will become dead soon, we postpone reclamation of the flash block containing the zombie data. This minimizes the extra copy operation that might be needed to migrate the soon-to-be-dead zombie data into another flash block. Second, when we have no option other than reclaiming a flash block which has both live data and zombie data, all the zombie data is put into a dedicated flash block to separate zombie data from live data. The flash block filled with zombie data will be cheap to reclaim as most of the zombie data will be dead in the near future.

Zombie Chasing is effective especially when there are random updates into the same region of the storage device, as in the OLTP (Online Transaction Processing) workload. To quantitatively evaluate the impact of Zombie Chasing on the performance of real applications, we have implemented the zombie-aware garbage collection in the prototype SSD. The prototype SSD is based on a popular SSD controller, which is widely used in many commercial SSDs. We have also modified the Linux kernel and the Oracle DBMS to deliver information on zombie data to the prototype SSD. Our evaluation with the TPC-C benchmark on the Oracle DBMS shows that Zombie Chasing improves the TPS (Transactions Per Second) value by up to 22% with negligible overhead.

The rest of this paper is organized as follows. Section 2 briefly describes the technical background of SSDs. In Section 3, we present the Zombie Chasing technique with introducing the new zombie state and the zombie-aware garbage collection. Section 4 explains the implementation detail of the prototype SSD and the necessary modifications in the Linux kernel and the Oracle DBMS for applying Zombie Chasing. Evaluation results are shown in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2 Background

In this section, we briefly describe NAND flash memory, the internal structure of SSDs, and the details of FTL.

2.1 NAND Flash Memory

A NAND flash memory chip consists of a number of *blocks*, or *flash blocks*[1], and each flash block, in turn, has $32 \sim 256$ *pages*. The page is the unit of read and write operations, while the flash block is the unit of erase operation. The size of a page or flash block varies among flash chips.

Another interesting characteristic of NAND flash memory is that overwrites into the same page are not allowed. The new data can be written only into a clean page. The pages which are no longer valid can be converted to clean pages after being erased. Since the erase operation is performed on a per-block basis, we cannot erase a page selectively. Instead, the entire flash block containing the page should be erased at once.

2.2 Solid State Drives (SSDs)

SSDs have been emerging as a revolutionary storage device thanks to their attractive features compared to HDDs. In particular, due to the absence of mechanical parts, SSDs are faster, lighter, more robust, and more energy-efficient than HDDs. Each SSD consists of a controller, DRAM, and several NAND flash memory chips. The NAND flash memory chips are connected to multiple channels so that I/O requests can be handled in parallel [7]. This is why SSDs exhibit much higher read/write bandwidths or IOPS (I/O Operations Per Second) than a single NAND flash memory chip.

The SSD controller is composed of host interface logic (e.g., SATA), embedded CPU(s), and other logic including ECC (Error Correction Codes) hardware. One of the most important roles of the controller is to emulate the block device interface while hiding the unique characteristics of NAND flash memory. The emulation is mainly achieved by the sophisticated firmware, called the Flash Translation Layer (FTL), run by the SSD controller. The performance and reliability of SSDs are significantly affected by the various policies used in the FTL.

1. The "block" of NAND flash memory should not be confused with the unit of I/O in the kernel. In this paper, we use the term "flash block" to denote the unit of erase operation in NAND flash memory.

2.3 Flash Translation Layer

To emulate the conventional block device interface over NAND flash memory, FTLs use two software techniques: *address mapping* and *garbage collection*.

2.3.1 Address Mapping

Since in-place update is not allowed in NAND flash memory, SSDs are over-provisioned with a certain amount of *extra flash blocks*. Using these extra flash blocks, FTLs internally keep some number of pre-erased flash blocks called *update blocks* to absorb incoming write requests. For a given write request from the host, FTLs redirect the write request into empty pages of the update blocks and maintain the mapping information between the logical address and the physical page location in NAND flash memory. If the new write request is overwriting the previous data mapped into the same logical address, the pages containing the previous data become *invalid pages*. The pages which have the up-to-date data are called *valid pages*.

According to the granularity of mapping information, FTLs are categorized as either *page-mapped*, *block-mapped*, or *hybrid* FTLs. Page-mapped FTLs maintain the mapping information on a page basis, while block-mapped FTLs organize the mapping information on a flash block basis. Hybrid FTLs are a mixture of these two schemes. They basically use block-level mapping but maintain the page-level mapping information only for a small number of flash blocks. Page-mapped FTLs require a much larger amount of memory than block-mapped or hybrid FTLs because they keep mapping entries for every single page. However, page-mapped FTLs are more flexible due to their smaller mapping unit and usually show better performance than block-mapped or hybrid FTLs. The prototype SSD implemented in this paper is also based on the page-mapped FTL.

2.3.2 Garbage Collection

A garbage collection procedure is necessary to reclaim invalid pages. Because the erase operation is performed on a per-block basis, the garbage collection procedure first selects an appropriate flash block which includes a certain number of invalid pages. The selected flash block is called the *victim block*. Selecting the victim block is based on one of the policies that will be described in the next subsection. Before erasing the victim block, the garbage collection procedure copies the live data within the victim block into empty pages of update blocks in order to prevent it from being discarded. Then, the victim block is changed to an update block after being erased. The extra copy and erase operations performed during garbage collection are the major overhead of FTLs in managing NAND flash memory.

In particular, random writes on SSDs have the potential to increase the overhead significantly. Since random writes update the data across a wide range of the logical address space, invalidated pages will be scattered over numerous flash blocks. To generate a clean flash block, FTLs should erase a large number of flash blocks, which results in many copy operations on valid pages. Even if random writes are buffered in the buffer cache, the overhead is still huge unless incoming random writes are clustered into a set of sequential writes. In this paper, we show that our Zombie Chasing technique reduces the FTL's overhead effectively under the workload including lots of buffered random writes.

The overhead of garbage collection is quantitatively estimated by the value called the *Write Amplification Factor (WAF)*. The WAF value is the ratio of the amount of total data written into NAND flash memory to the amount of data written by the host. The total written data includes not only the data written by the host, but also the data written by copy operations performed during garbage collection. The WAF value represents how efficiently FTL handles write requests from the host.

2.4 Victim Block Selection Policies

When selecting a victim block, most FTLs use one of the following two policies: *greedy* [8] and *cost-benefit* [9], [10]. The greedy policy selects the flash block which has the largest number of invalid pages as the victim block. This policy is easy to implement and generates the maximum number of empty pages after erasing the victim block.

The cost-benefit policy was first introduced in the log-structured file system (LFS) [11]. The policy rates each flash block according to the cost and benefit of reclaiming invalid pages and chooses the flash block with the highest ratio of benefit to cost. In general, the cost is simply the number of read and write operations required to copy valid pages into the update block. The benefit is the product of the number of invalid pages and the age of the flash block which is represented by the most recent modified time of the flash block. The victim block is the flash block that maximizes the following equation:

$$\frac{benefit}{cost} = \frac{a * i}{2(N - i)}, \tag{1}$$

where N is the total number of pages in each flash block and i is the number of invalid pages of the flash block. a indicates the age of the flash block. Although the cost-benefit policy requires more memory and computing power than the greedy policy, it outperforms the greedy policy especially when write requests have high locality.

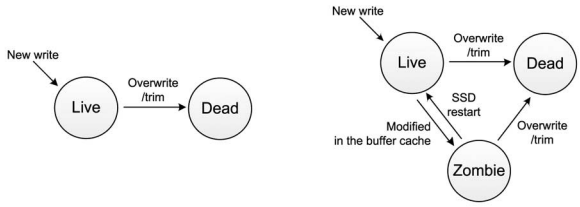
3 Zombie Chasing

In this section, we present a novel technique called *Zombie Chasing* for efficient flash memory management. First, we introduce a new liveness state for the *zombie data* that will be dead shortly. Then, we propose enhanced garbage collection procedures which consider such zombie data.

3.1 Zombie Data

In the current block I/O layers, the data stored in SSDs is either *live* or *dead*, as depicted in Fig. 1(a). When the host writes some data to SSDs, the data has a live state in the SSDs. Later if new data is written, the previous data becomes dead in the SSDs. When the host tells the SSDs via the TRIM command that a file containing the data is deleted by the file system, the data also becomes dead. The dead data can be discarded from SSDs during garbage collection as mentioned in Section 2.3.

We note that user applications access the data stored in SSDs through the buffer cache to increase performance. When



(a) Transitions between live and dead states (b) Transitions among live, dead, and zombie states

Fig. 1. Liveness state transitions.

reading a file, the buffer cache is populated with the data read and/or prefetched from SSDs. When writing a file, the new data is temporarily cached in the buffer cache instead of being written directly into SSDs. The cached new data, or the *dirty data*, is flushed to SSDs later by the virtual memory manager of the kernel. Our approach is to distinguish the live data which has the up-to-date version in the buffer cache from the live data which does not, and to use this information for efficient flash memory management in SSDs.

In this paper, we propose a new data liveness state called the *zombie* state. We further divide the live state shown in Fig. 1(a) into the (real) live state and the zombie state. The zombie data represents data which is currently alive in SSDs, but will be dead soon when the corresponding dirty data in the buffer cache is written into SSDs. Fig. 1(b) illustrates the new zombie state together with the existing states and state transitions among them. If SSDs know that the live data is modified in the buffer cache, its state is transitioned to zombie. How the information on the zombie data is delivered to SSDs is described in Section 3.3. If the corresponding dirty data is later written into SSDs or the host trims the zombie data, it becomes dead. When the dirty data in the buffer cache is lost due to sudden power failure, the zombie data returns to the live state.

We can see that the zombie state is a transient state between the live and dead state. The zombie data is different from the live data in that the zombie data has a fate to die in the near future. Also, the zombie data cannot be accessed by user applications in the usual way since the up-to-date version is available in the buffer cache. However, unlike the dead data, the zombie data should not be discarded from SSDs so that user applications can access it when SSDs are restarted.

How soon the dirty data is flushed into SSDs depends on the policy used in the buffer cache. In the Linux kernel, the dirty data is written to disks at least in 30 seconds by default. In the case of the database system used in our evaluation, more than 70% of the dirty data in the buffer cache is written into disks in less than 20 seconds.

3.2 Zombie-Aware Garbage Collection

The garbage collection procedure is composed of three steps: (1) selecting a victim block, (2) copying valid pages of the victim block to update blocks, and (3) erasing the victim block. Zombie Chasing improves the first two steps using the zombie data information.

3.2.1 Zombie-Aware Victim Block Selection Policy

The greedy policy and the cost-benefit policy do not distinguish the zombie data from the live data when selecting a victim block. The greedy policy simply selects a flash block which contains the largest number of invalid pages regardless of the amount of zombie data. The cost-benefit policy considers only the number of invalid pages and the age of each block when estimating the cost and benefit of reclaiming a flash block. Therefore, the victim block selected by one of these two policies might contain the zombie data as well as the live data.

When performing garbage collection, FTL should copy the valid pages of a victim block to update blocks before erasing the victim block. If the selected victim block contains the zombie data, it will also be copied to the update blocks. This is undesirable for the following two reasons. First, since the zombie data will become dead shortly, it would be totally unnecessary to copy the zombie data to update blocks if we waited a little longer for the flash block before choosing it as the victim block. Second, the copied zombie data will stay in the update block only for a short period of time. If the zombie data dies in the update block later, it should be reclaimed by another garbage collection, which apparently increases the garbage collection overhead. Therefore, copying the zombie data into update blocks is inefficient and also wasteful of empty pages in update blocks.

To remedy such shortcomings in the existing policies, we propose new zombie-aware victim block selection policies. The essential of the zombie-aware policies is to put off selecting a flash block which has too much zombie data as a victim block. However, the goal of garbage collection is to make clean pages by reclaiming the space occupied by the dead data. This means that the zombie-aware garbage collection policies should consider the amount of both dead data and zombie data.

In this paper, we extend the greedy policy and the cost-benefit policy so that they are aware of the zombie data. The zombie-aware greedy policy, called the *z-greedy* policy, estimates the benefit of reclaiming a flash block by the following equation:

$$i - min\left(z, \frac{i}{2}\right), \tag{2}$$

where i is the number of invalid pages in the flash block and z denotes the number of *zombie pages*, i.e., the pages that contain the zombie data in the flash block. The z-greedy policy selects the flash block that maximizes Eq. 2 as a victim block.

In the greedy policy, the benefit of reclaiming a flash block is simply equal to the number of invalid pages. Apparently, the greedy policy does not take into account the number of zombie pages in the flash block. However, the z-greedy policy considers it less beneficial to perform garbage collection on a flash block when the flash block has some zombie pages. As we can see in Eq. 2, the estimated benefit is lowered by the number of zombie pages when the number of zombie pages is less than half of the number of invalid pages. Otherwise, we reduce the benefit by up to half of the number of invalid pages in order to prevent a flash block which has too few invalid pages from being selected as a victim block. No matter how many zombie pages a flash block has, the estimated benefit is limited to one-half of the number of invalid pages. Hence, the number of invalid pages in the victim block selected by the z-greedy policy will exceed the half of the number of invalid pages in any other flash block.

The rationale behind Eq. 2 is as follows. Suppose that there are N flash blocks, denoted as B_1, B_2, $\cdots$, B_N, where only B_1 has some zombie pages and it has the largest number of invalid pages among the flash blocks. Let i_k and z_k be the number of invalid pages and the number of zombie pages in B_k, respectively. Normally, the presence of zombie pages in B_1 should postpone B_1's chance of becoming a victim block as we expect that all the zombie pages in B_1 will be dead soon. We realize this by reducing the benefit of reclaiming B_1 by the amount of zombie pages, thus treating B_1 as if it had only $i_1 - z_1$ invalid pages. However, it is necessary to limit the contribution of zombie pages to the estimated benefit.

Assume that some other flash block B_k is selected as a victim block because the benefit of reclaiming B_k is estimated larger than that of B_1, i.e., $i_k > i_1 - z_1$. However, if i_k is smaller than z_1, contrary to our expectations, only i_k zombie pages can be dead after the reclamation of B_k since a single clean page is needed for each zombie page to become dead. This suggests that at most i_k (out of the total z_1) zombie pages should contribute to the estimated benefit of B_1. Therefore, for B_k to be selected as a victim block, B_k's benefit (i_k) should be larger than B_1's benefit (i_1 - i_k), i.e., $i_k > i_1$ - i_k or $i_k > i_1/2$. In other words, any flash block which has invalid pages less than the half of invalid pages in B_1 cannot be selected as a victim block instead of B_1. In order to prevent such flash blocks from being selected as a victim block, the estimated benefit of reclaiming a flash block with zombie pages does not become smaller than $i/2$ as shown in Eq. 2.

Similarly to the z-greedy policy, we define the zombie-aware cost-benefit policy, called the *z-cost-benefit* policy, as a policy that chooses the flash block which maximizes the following equation:

$$\frac{benefit}{cost} = \frac{a * \left(i - min\left(z, \frac{i}{2}\right)\right)}{2(N-i)}, \qquad (3)$$

where the meanings of i and z are the same as those used in Eq. 2. N and a denote the total number of pages in each flash block and the age of the flash block, respectively. The term $(a * i)$ which represents the benefit in the original cost-benefit policy (cf. Eq. 1) has been replaced with $a * \left(i - min\left(z, \frac{i}{2}\right)\right)$ in the z-cost-benefit policy. As the benefit of the flash block which contains the zombie data is estimated lower, the z-cost-benefit policy favors the flash block which has fewer zombie pages as a victim.

The victim block selected by either the z-greedy policy or the z-cost-benefit policy will have a smaller number of zombie pages which certainly reduces the number of copy operations required for the zombie data. However, now it is possible that a flash block which does not have the maximum number of invalid pages is selected as a victim block. In this case, the amount of valid data that should be copied to update blocks increases and reclaiming the victim block will produce fewer empty pages. This appears to have adverse effects on garbage collection efficiency. However, the zombie-aware victim selection policies reduce the FTL's overhead in the long term by reducing the number of extra copy operations for the zombie data and improving the utilization of update blocks.

Figs. 2 and 3 illustrate an example of garbage collection performed with the greedy policy and the z-greedy policy,

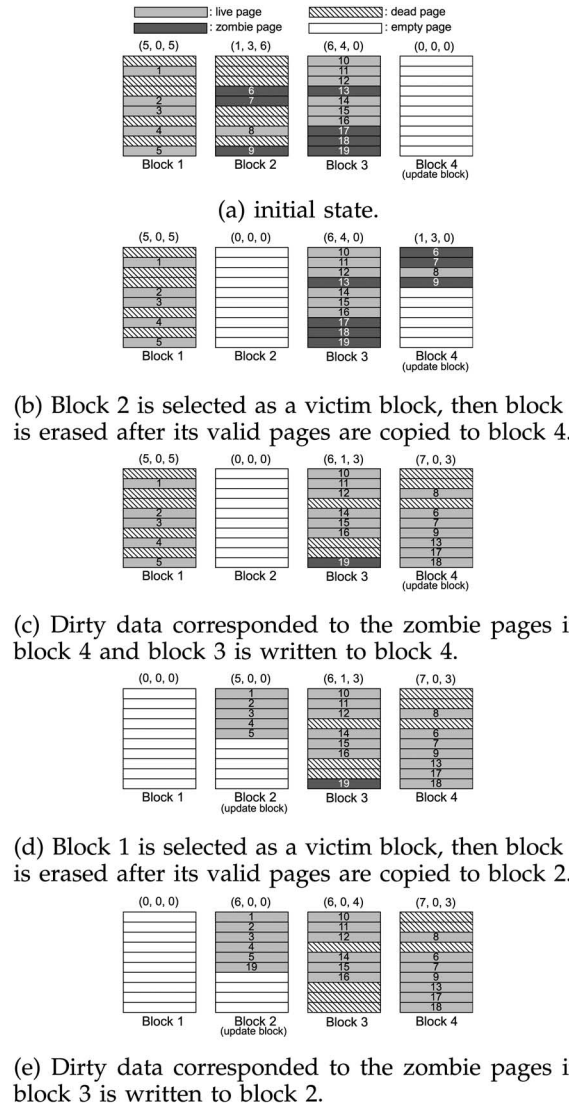


Fig. 2. Example of the greedy policy.

respectively. We assume that there are four flash blocks and each flash block consists of ten pages. The values (x, y, z) shown in the top of each flash block represent the number of live pages storing the (real) live data, the number of zombie pages, and the number of dead pages in the flash block, respectively. The number in each valid page (i.e., live page or zombie page) indicates the logical address number (such as the *sector* number) of the data stored in the valid page.

At the initial state depicted in Figs. 2(a) and 3(a), there are a total of 19 live pages among which seven pages are zombie pages. Note that block 4 is initially designated as the update block. Because there remains only one clean flash block, it is the time to reclaim dead pages by initiating the garbage collection procedure. The greedy policy selects block 2 as a victim block since block 2 has the largest number of invalid pages. Meanwhile, the z-greedy policy chooses block 1 which has no zombie pages even if it has a smaller number of invalid pages than block 4. During the first garbage collection, the greedy policy and the z-greedy policy require four and five copy operations for valid pages, respectively. Figs. 2(b) and 3 (b) depict the results after the first garbage collection is performed according to each policy.

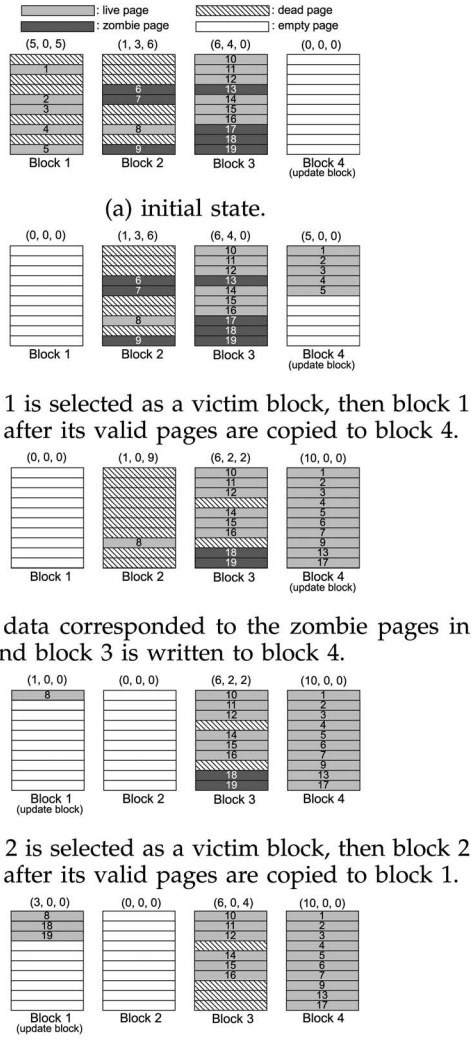


Fig. 3. Example of the z-greedy policy.

Figs. 2(c) and 3(c) show the status of flash blocks after some of the dirty data corresponding to zombie pages are written to SSDs. In both cases, the dirty data flushed from the buffer cache is sequentially written into the current update block (block 4). When the update block runs out of empty pages, the second garbage collection is started. As Fig. 2(d) depicts, the greedy policy selects block 1 as a victim block and five copy operations are carried out. On the other hand, the z-greedy policy selects block 2 as a victim block and performs only one copy operation during the second garbage collection, as Fig. 3 (d) shows. Figs. 2(e) and 3(e) illustrate the final state when all the dirty data are flushed from the buffer cache.

In summary, the greedy policy performs nine copy operations while the z-greedy policy does only six copy operations. Accordingly, three more empty pages remain after all the dirty data are flushed from the buffer cache. We can observe that the z-greedy policy is more efficient than the greedy policy although it requires more copy operations than the greedy policy at times as in the first garbage collection. Similarly, the z-cost-benefit policy shows better efficiency than the cost-benefit policy.

3.2.2 Zombie Block

In the management of NAND flash memory, it is important for FTLs to arrange the hot data (i.e., frequently updated data) in the same flash block in order to reduce the overhead of garbage collection. This is because the hot data has high probability of being overwritten in the near future. Thus, the flash block containing the hot data will be almost full of invalid pages in a short time. Such flash blocks can be reclaimed with lower cost as only a few copy operations are required for the valid data.

There are several previous researches which aim at estimating the degree of hotness for the given data in various ways and clustering the data into several different update blocks according to its degree of hotness [10], [12]-[14]. Most of these approaches are based on the write access patterns of the data in the past which inevitably have some limitation in accuracy.

In the same spirit, we treat the zombie data as hot data since it will be overwritten soon. This classification is clearly more accurate than the previous approaches as it is based on actual write requests that will occur in the near future. To separate the (hot) zombie data from the other (cold) live data, the proposed zombie-aware garbage collection procedures allocate a dedicated update block called the *zombie block* and all the zombie data are copied into the zombie block during garbage collection. We can expect that the zombie block will be filled with invalid pages soon and it can be reclaimed with low overhead.

3.3 Notification of the Zombie Data

In order to realize Zombie Chasing, we need a certain way to notify SSDs of the information on the zombie data. However, in the standard disk protocols such as SATA and SCSI, there is no appropriate command for the notification. There are three approaches to address this problem.

One approach is to add a new command for notification of the zombie data, similar to the TRIM command. The main obstacle of this approach is that it requires the modification of the standard protocols, which involves a significant amount of time and effort. In addition, the overall performance of SSDs can be impaired by the non-negligible overhead of the new command despite the benefit brought by considering the zombie data. Even for the TRIM command which is recently added to the SATA protocol for SSDs, several literatures have discussed its command overhead and how to use the TRIM command properly to reap the benefits it has promised [4], [5].

The second approach is to define a custom vendor-specific command for the purpose of zombie data notification. Most standard disk protocols including SATA and SCSI support a generic extension mechanism by allowing a set of vendor-defined commands to be used as desired by the manufacturer. Such a vendor-specific command can be used as long as the block device driver in the kernel and the storage device agree on it, without requiring any change in the standard protocols. However, the vendor-specific command also incurs non-negligible overhead as in the first approach.

The final approach is to piggyback the zombie data information in the unused fields of the existing read/write commands. This can be easily implemented by modifying the block device driver of the kernel and the SSD's firmware. The associated overhead is very small since it does not require any

additional commands. The downside of this approach is that the zombie data information cannot be delivered to SSDs when the host does not issue any read/write commands to SSDs. However, our zombie-aware garbage collection algorithms described in Section 3.2 do not require full and complete knowledge on the zombie data, i.e., the zombie data information need not be transferred to SSDs instantly. Delaying notification of the zombie data may make some zombie data disguise itself as live data for a moment, but it does not harm the correctness of the zombie-aware garbage collection. For these reasons, we have used this approach for notification of the zombie data. Its implementation details will be described in the following section.

4 Implementation

The implementation of Zombie Chasing can be broken into three parts. First, we have implemented our own prototype SSD which receives the zombie data information from the host and implements zombie-aware garbage collection algorithms. Second, we have modified the block I/O layers of the Linux kernel to deliver the zombie data information to the prototype SSD. Finally, at the top level, we consider two application scenarios of Zombie Chasing; one is the virtual file system (VFS) layer of the Linux kernel with the page cache, and the other is the Oracle DBMS which has its own buffer cache layer.

4.1 Prototype SSD

We have implemented a prototype SSD using the development board equipped with MLC (Multi Level Cell) NAND flash memory, 64 MB SDRAM, and the Indilinx Barefoot controller whose internals are publicly released by the OpenSSD project [15]. The Barefoot controller has been widely used in many commercial SSDs. Also, there are previous researches which utilize the OpenSSD platform employing the Barefoot controller [16], [17].

The prototype SSD has 16 *banks* of NAND flash memory. Since the size of each bank is 8 GB, the total capacity of the prototype SSD is 128 GB. A single bank contains two NAND flash memory *chips*. Each chip has 8,192 flash blocks organized into two *planes* of 4,096 flash blocks. Each flash block, in turn, consists of 128 4KB-pages.

The 16 banks are connected to the controller and DRAM via four channels. The controller can issue different NAND flash commands to multiple banks in parallel and each bank handles the command independently. In addition, the controller is designed in such a way that two chips of a bank form virtually a single chip as if the page size is 8KB. Each chip also supports *two-plane* mode commands. If the mode is enabled, the two planes of each chip operate as a single plane so that the *effective page* size becomes 16KB. By combining four flash blocks together (i.e., 1 flash block/plane * 2 planes/chip * 2 chips/bank), the *effective flash block* size of a bank is also increased to 2 MB. In this way, although the page size of each NAND flash memory chip is only 4KB, read and write bandwidths are easily quadrupled. We enable the two-plane mode by default.

We have modified the SATA controller's firmware of the prototype SSD slightly to receive the information on the zombie data. In the SATA standard revision 2.6, the SATA read/write command has the unused fields, or the *reserved area* whose size is five bytes. As described in Section 3.3, we use four bytes of the reserved area for specifying the sector number of the zombie data and the remaining one byte for its length in sectors. For a given SATA read or write command, the modified firmware in the prototype SSD extracts the sector number and length of the zombie data from the command, and sends the information to FTL.

Inside the prototype SSD, we have implemented a page-mapped FTL based on *DAC* [10], which is one of the most popular page-mapped FTLs. We integrate our zombie-aware garbage collection procedures including the z-greedy and z-cost-benefit policies into the FTL. For fair comparison, the traditional policies, greedy and cost-benefit, are also implemented. The use of the zombie block is disabled when the FTL is configured to use one of the traditional policies. Also, the FTL performs garbage collection procedures on a per-bank basis due to performance issues. When clean flash blocks run out in a bank, a garbage collection procedure for the bank is performed. A victim block is selected among flash blocks in the bank and its valid pages are copied to other flash blocks in the same bank. When selecting a victim block, the FTL estimates the value of the corresponding equation (i.e., Eq. 2 for the z-greedy policy and Eq. 3 for the z-cost-benefit policy) for each flash block in a bank and searches the flash block that maximizes the value. Therefore, the time complexity of the zombie-aware victim block selection is O(N), where N is the number of flash blocks in the bank.

4.2 Linux Kernel Modification

We have modified the block I/O layers in the Linux kernel to process the zombie data information as follows. First, we have added a dedicated mode in the `ioctl()` system call for the explicit notification of the zombie data. If some data cached in the buffer cache becomes dirty, the corresponding sector number and its length are passed to the generic block I/O layer by the `ioctl()` call. Depending on circumstances, `ioctl()` is called by user applications or by one of the Linux kernel components. In the following subsection, we will show some use cases of the `ioctl()` call in more detail.

The information on the zombie data is buffered in a dedicated zombie data queue of the generic block I/O layer. Whenever a read or write request towards the prototype SSD is received, the generic block I/O layer inserts the information on one of the zombie data's locations into the request and forwards it to the I/O scheduler queue. After the request is dispatched from the queue, it is transformed into the SATA read or write command by filling the reserved area with the location of the zombie data. Finally, the SATA command is transferred to the prototype SSD.

4.3 Applications of Zombie Chasing

In order to demonstrate the feasibility of Zombie Chasing, we consider two typical scenarios: (1) when an application issues random updates to the prototype SSD through the page cache of the Linux kernel, and (2) when a popular commercial DBMS, Oracle DBMS, runs the OLTP workload with its own buffer cache. For each scenario, this subsection describes the required kernel-level or user-level modification for applying Zombie Chasing.

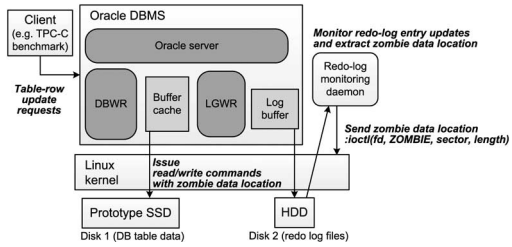


Fig. 4. The Oracle DBMS with Zombie Chasing.

4.3.1 Applying Zombie Chasing to VFS

To demonstrate that user applications can benefit from Zombie Chasing transparently, we consider the case when a user application opens a block device file of the prototype SSD (e.g., `/dev/sdx`) and issues write requests with the `write()` system call. The data written by each write request is cached in the page cache before it is actually written to the prototype SSD[2]. In this circumstance, the data stored in the prototype SSD, which will be overwritten by the corresponding dirty data in the page cache, is the zombie data. To extract the information on the zombie data, we have modified the VFS layer of the Linux kernel.

The `write()` system call issued by an application invokes the `vfs_write()` routine in the kernel. We have modified `vfs_write()` so that it notifies the prototype SSD of the zombie data information using the `ioctl()` call after placing the written data in the page cache. Since the application is writing data to the block device file directly, the location of the zombie data is identical to the location of the current write request. In case user applications are writing the data through a file system, the overall process is similar except that the location of the zombie data should be obtained from the file system after translating < file inode, file offset > into the corresponding block offset.

4.3.2 Applying Zombie Chasing to Oracle DBMS

Fig. 4 depicts the overall architecture of the Oracle DBMS extended with Zombie Chasing. If clients update some records of a table in the database, the updated data is first cached as dirty data in the internal buffer cache of the Oracle DBMS. Later, the *DBWR* thread eventually flushes the updated data to the prototype SSD.

In order to keep track of the zombie data, we have implemented a daemon application which performs online monitoring of redo-log files in the Oracle DBMS[3]. Before caching the updated data in its buffer cache, the Oracle DBMS creates redo-log records corresponding to the updated data. The redo-log records contain the meta information on the update requests so that they can be recovered from various emergency cases. According to the well-known WAL (write-ahead-logging) protocol [18], the *LGWR* thread synchronously flushes the cached log records to one of the redo-log files in a separate disk (disk 2 in Fig. 4) before *DBWR* writes the dirty data in the buffer cache to the prototype SSD. The location of the zombie data can be obtained by analyzing the meta information in the redo-log file as it contains the location where the dirty data will be written. Whenever a new record is appended to one of the redo-log files, our online monitoring daemon extracts this information from the new record and sends it to the prototype SSD using the `ioctl()` system call.

2. In the Linux kernel, the buffer cache is integrated into the page cache.

3. Note that monitoring redo-log files is an implementation technique to identify dirty data in the buffer cache. If the source codes of the Oracle DBMS are publicly available, we would modify the internals of the Oracle DBMS instead of implementing the daemon application.

TABLE 1
The Descriptions of Four Types of Garbage Collection Policies

Type	Victim Block Selection Policy	Zombie Block
GC-GD	greedy	no
GC-CB	cost-benefit	no
ZGC-GD	z-greedy	yes
ZGC-CB	z-cost-benefit	yes

5 Evaluation

In this section we present the evaluation results of Zombie Chasing. The results show that Zombie Chasing improves the SSD's performance effectively under the workloads including lots of buffered random writes.

5.1 Methodology

We have evaluated Zombie Chasing on a machine equipped with an Intel i7 950 processor (quad-core, 3.07 GHz) and 16 GB of main memory, running Oracle Enterprise Linux 5 with the Linux kernel version 2.6.34. The prototype SSD is attached to the machine via the SATA 2.0 interface. The FTL inside the prototype SSD is a page-mapped FTL based on DAC [10], which employs the hot-data-aware garbage collection technique. For the DAC's hot-data clustering algorithm, the number of regions and the time threshold for state switching are configured to three and zero, respectively[4].

For ease of evaluation, we have limited the physical capacity of the prototype SSD to 32 GB in the following two configurations. The first configuration is to make the prototype SSD to use only four banks. In the second configuration, the prototype SSD is set to utilize all 16 banks. Basically, if there is no additional comment, the evaluations described in this section were conducted on the prototype SSD of the first configuration. The logical capacity of the prototype SSD is set to 29.8 GB. About 6% of the logical capacity is hidden from the host and used as over-provisioned extra flash blocks. The remaining capacity is set aside for bad block remapping and also for storing mapping information, bad block lists, and firmware image.

The proposed Zombie Chasing technique is evaluated with two benchmarks: the in-house micro-benchmark and the TPC-C benchmark. The in-house micro-benchmark is conducted to the VFS layer of the Linux kernel as described in Section 4.3.1, while the TPC-C benchmark is performed on the Oracle DBMS as presented in Section 4.3.2. The details of each benchmark will be described in the following subsections.

For each benchmark, we consider four types of garbage collection policies as shown in Table 1. *GC-GD* and *GC-CB* are

4. Overall, this configuration has shown the best performance in our evaluations.

TABLE 2
The Flush Daemon's Configuration

Parameter	Value
dirty_writeback_centisecs	500
dirty_expire_centisecs	3000
dirty_backgroud_ratio	10
dirty_ratio	20

conventional garbage collection policies which do not utilize the zombie data information. *ZGC-GD* and *ZGC-CB* are the zombie-aware garbage collection policies which use the zombie block and implement the z-greedy and z-cost-benefit victim selection policies, respectively. Also, we have carried out the evaluation varying the buffer cache size in order to investigate its impact on the overall performance. As the buffer cache size increases, the amount of zombie data available to the prototype SSD also grows and this allows for better victim block selection.

5.2 Micro-Benchmark

5.2.1 Overview

We have implemented a micro-benchmark to generate synthetic I/O workloads which contain lots of buffered random writes. When the micro-benchmark generates the synthetic workloads, the main consideration is the locality of write requests. In fact, the locality of the write requests and the buffer cache size are two main factors affecting the overall performance of Zombie Chasing. In particular, the degree of the locality influences the distribution of the zombie data. If write requests have a high degree of locality, flash blocks with lots of dead data may contain much zombie data as well. In such a situation, the zombie-aware victim block selection policies make better decisions since the greedy or cost-benefit policy does not consider the zombie data at all. Thus, we measure the I/O performance improvement under synthetic workloads which have various degrees of locality among write requests.

Before the micro-benchmark is executed, we first fill the prototype SSD with 29.8 GB of dummy data. Once executed, the micro-benchmark invokes 16 threads. Each thread opens the block device file associated with the prototype SSD and generates a certain number of random read/write requests using the `read()`/`write()` system calls. The total amount of data to write is fixed to 5.12 GB and the total amount of data to read varies according to the ratio of read requests to write requests. After all read/write requests from 16 threads finish, the micro-benchmark reports the elapsed time.

Each thread issues read/write requests as follows. The read requests retrieve the data stored in random locations within the entire logical address space of the prototype SSD. In order to control the degree of locality among write requests, write requests are created according to the *Pareto principle* (also known as the *80–20 rule*); the $(100 - h)\%$ of write requests store the data to random locations within the top $h\%$ of the logical address space of the prototype SSD. The smaller the value of h is, the higher the degrees of both spatial and temporal locality among write requests are. The locations of the read/write requests are aligned to a 16KB boundary and the data size of each request is also set to 16KB, which is the same as the effective page size of the prototype SSD.

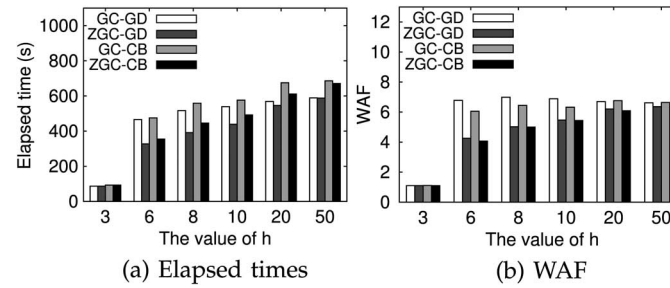


Fig. 5. Elapsed times and WAF values when the ratio of read requests to write requests is 0.5 and the page cache size is 4 GB.

The impact of Zombie Chasing on the performance of the micro-benchmark is affected by the behavior of the `flush` daemon in the Linux kernel, which is in charge of flushing dirty pages in the page cache periodically into the prototype SSD. We have utilized the default configuration for the `flush` daemon whose main parameter values are shown in Table 2.

5.2.2 The Effect of the Locality among Write Requests and the Buffer Cache Size

First of all, we have conducted the micro-benchmark varying the locality among write requests and the page cache size. Figs. 5 and 6 show the evaluation results using the micro-benchmark with various values of h. In these figures, the ratio of read requests to write requests is 0.5 and the page cache size is 4 GB and 2 GB, respectively. In order to control the size of the page cache, we have restricted the memory size available to the kernel by using the kernel boot option (*mem*).

As Figs. 5 and 6 show, the general trend is that the prototype SSD shows better performance with the zombie-aware garbage collection algorithms. When the value of h is between 6 and 20, *ZGC-GD* and *ZGC-CB* outperform *GC-GD* and *GC-CB*, respectively, in terms of both the elapsed time and the WAF value. In particular, when the page cache size is 4 GB and the value of h is 6, the elapsed times of *ZGC-GD* and *ZGC-CB* are 29.8% and 25.3% lower than those of *GC-GD* and *GC-CB*, respectively. The WAF value of *ZGC-GD* is reduced by 47.3% as compared to that of *GC-GD* and the *ZGC-CB* 's WAF value is also decreased to 67.1% of the *GC-CB* 's value. We can see that the I/O performance of the prototype SSD improves as the overhead of garbage collection decreases when write requests have a certain degree of locality. Note that when the value of h is 3, the overhead of garbage

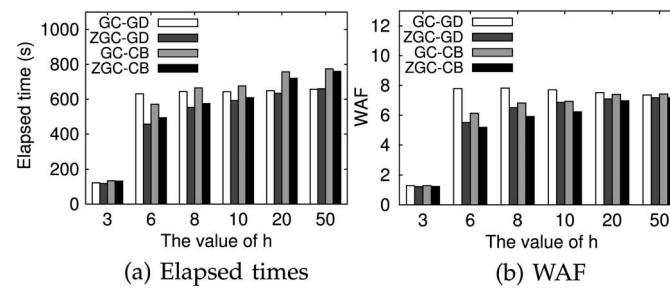


Fig. 6. Elapsed times and WAF values when the ratio of read requests to write requests is 0.5 and the page cache size is 2 GB.

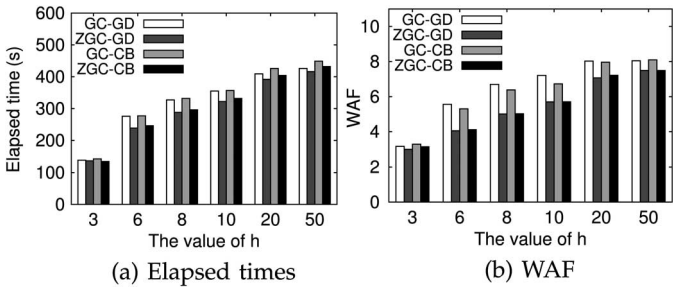


Fig. 7. Elapsed times and WAF values when the prototype SSD is configured to use all 16 banks.

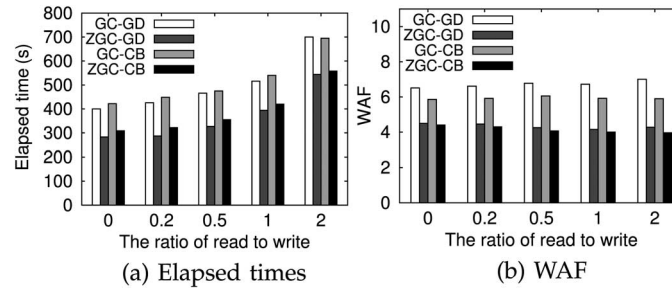


Fig. 8. Elapsed times and WAF values when the page cache size is 4 GB and the value of h is 6.

collection is so low that there is no room for the performance improvement by Zombie Chasing.

As the locality among write requests decreases or the page cache gets smaller, the improvement gained by Zombie Chasing diminishes. For instance, the elapsed times and the WAF values of *ZGC-GD* and *ZGC-CB* are improved by less than 5% when there is no locality among write requests (i.e., when $h = 50$.). Also, the overhead of garbage collection is reduced by about 30% when the page cache is 2 GB and the value of h is 6 while the overhead decreases by more than 35% when the page cache is 4 GB for the same value of h.

Note that there seems to be inconsistency between the elapsed time and the WAF value. For example, when the page cache size is 2 GB and the value of h is 10, *ZGC-CB* takes a longer elapsed time even if it has a smaller WAF value compared to *ZGC-GD*. This is because *ZGC-CB* (cf. Eq. 3) has a larger overhead in selecting a victim block among flash blocks in each bank than *ZGC-GD* (cf. Eq. 2). On the prototype SSD, *ZGC-CB* takes 7.65 ms on average to select a victim block among flash blocks in a bank, while *ZGC-GD* takes only 3.19 ms to do the same operation. Therefore, although the garbage collection efficiency of *ZGC-CB* represented by the WAF value is better than that of *ZGC-GD*, *ZGC-CB* has a penalty in the elapsed time.

We have performed the same benchmark on the prototype SSD that is configured to provide the maximum parallelism by utilizing all 16 banks. Overall, the evaluation results are similar to those of the prototype SSD that is configured to use only 4 banks. As Fig. 7 depicts, Zombie Chasing enhances the I/O performance of the prototype SSD successfully. The overhead of garbage collection procedure is reduced by up to 26% and the elapsed time is improved by 14.4% at the same time.

5.2.3 The Impact of the Ratio of Read Requests to Write Requests

In this subsection, we investigate the impact of the ratio of read requests to write requests on the overall performance.

TABLE 3
The Amount of Zombie Data Which the Prototype SSD Is Aware of during the Experiment of Fig. 8

	The ratio of read requests to write requests				
	0	0.2	0.5	1	2
Volume	432MB	562MB	699MB	702MB	703MB
Ratio	61%	79%	98%	99%	99%

We vary the ratio of read requests to write requests fixing the page cache size to 4 GB and the value of h to 6. We expect that if the ratio of read requests rises, the performance improvement achieved by Zombie Chasing increases as the zombie data information is piggybacked to the prototype SSD more often. Table 3 shows the amount of zombie data which the prototype SSD is aware of and its ratio to the amount of dirty data in the page cache. As our expectation, the higher the ratio of read requests is, the more zombie data the prototype SSD is conscious of.

Fig. 8 illustrates the elapsed times and the WAF values of the above experiment. The WAF values of *ZGC-GD* and *ZGC-CB* decreases as the ratio of read requests increases. For example, the WAF value of *ZGC-CB* is 3.96 when the ratio is 2 while it is 4.40 when the ratio is 0. However, the elapsed time is maximally improved when the ratio is 0.2. This is because the effect of Zombie Chasing on the overall performance of the micro-benchmark lessens as the amount of read requests increases.

5.2.4 The Breakdown of the Performance Improvement

The performance improvement achieved by Zombie Chasing is due to the utilization of both the zombie-aware victim block selection policy and the zombie block. In order to analyze their contribution to the overall performance separately, we have additionally evaluated the zombie-aware garbage collection procedure which only employs the zombie-aware victim block selection policy, but not the zombie block.

Fig. 9 shows the evaluation results using various values of h, when the page cache size is 4 GB and the ratio of read requests to write requests is 0.5. The *ZGC-GD-NZ* and the *ZGC-CB-NZ* denote the type of garbage collection which utilizes the z-greedy policy and the z-cost-benefit policy, respectively, without the use of the zombie block. We can

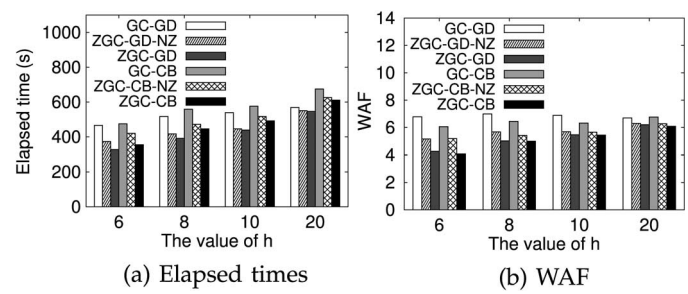


Fig. 9. The evaluation results of the zombie-aware garbage collection which utilizes only the zombie-aware victim block selection policy, but not the zombie block.

TABLE 4
The Description of TPC-C Configurations

Name	Warehouses (DB tables' size)	Users	Transaction time (warm-up + sampling)
TPC-C1	300 (28.8GB)	16	1 hour
TPC-C2	275 (26.5GB)		(15 min. + 45 min.)

see that as the locality among write requests decreases, the performance improvement gained by the use of the zombie block diminishes. When the value of h is 6, the WAF value of *ZGC-GD* is about 18% lower than those of *ZGC-GD-NZ*. However, the difference between their WAF values is less than 3% when the value of h is 20.

5.2.5 The Overhead of Zombie Data Notification

In order to investigate the overhead associated with sending the zombie data information to the prototype SSD, we measured the elapsed times of the micro-benchmark when enabling and disabling the notification mechanism in the block I/O layers. As we expect, there is no meaningful difference in the elapsed times whether the notification is enabled or disabled. This suggests that the overhead incurred in delivering the zombie data information to the prototype SSD is almost negligible.

Additionally, in order to examine the feasibility of the notification mechanism using an additional SATA command (e.g., the first two approaches mentioned in Section 3.3), we have measured the overhead when the prototype SSD is notified of the zombie data information via the TRIM command with the NO_BARRIER mode. We evaluated the performance of the micro-benchmark when such notification mechanism is used. From the evaluation results, we confirmed that the overhead is serious. Although the zombie-aware garbage collection is utilized, there is no improvement on the performance of the micro-benchmark and the elapsed time increases by more than 20% in some cases.

5.3 TPC-C Benchmark

5.3.1 Overview

The TPC-C benchmark is an Online Transaction Processing (OLTP) benchmark for database systems, developed by the Transaction Processing Performance Council (TPC) [19]. The TPC-C benchmark measures the OLTP performance of database systems, which is usually represented by the TPS (Transactions Per Second) value. The TPS value is the number of transactions the database system under test can process for each second. When the TPC-C benchmark runs, the storage system of the database system experiences lots of buffered random writes. Also, the write requests generated by the TPC-C benchmark have a certain degree of locality. In this evaluation, we measure the improvement in the TPS value for the TPC-C benchmark when the database system is extended with Zombie Chasing.

We have used a commercial benchmark tool to generate TPC-C workloads on the Oracle DBMS 11gR2. The Oracle DBMS is configured to access the prototype SSD directly without any file system. The block size of database tables is set to 16KB, which is the same as the effective page size of the

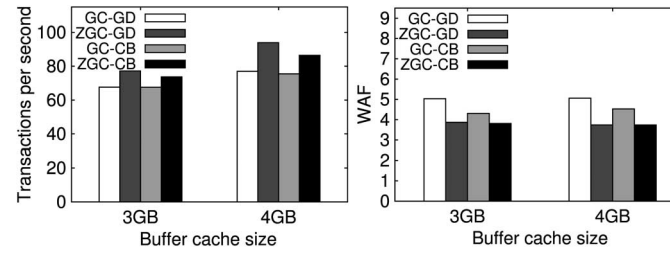


Fig. 10. TPS and WAF values when the configuration of the TPC-C benchmark is TPC-C1.

prototype SSD. We use two configurations of the TPC-C benchmark as described in Table 4. The number of warehouses in TPC-C1 and TPC-C2 is 300 and 275, respectively. In both TPC-C configurations, the number of users is 16 and the transaction time is 1 hour. Note that since the size of the warehouses' tables is smaller than the prototype SSD's logical space, the remaining space is used as over-provisioned extra flash blocks additionally. Therefore, the actual amount of extra flash blocks becomes 9.6% and 19% of the total SSD capacity for TPC-C1 and TPC-C2, respectively.

5.3.2 Results

Fig. 10 depicts the results of TPC-C1 when the buffer cache size of the Oracle DBMS is set to 3 GB and 4 GB. Overall, similarly to the results of the micro-benchmark, the zombie-aware garbage collection procedures show better performance in all configurations. Particularly, when the buffer cache size is 4 GB, the TPS value of *ZGC-GD* is 22% higher than that of *GC-GD* while the WAF value is 26% smaller. In the case of *ZGC-CB*, its TPS value is improved by 14.4% and its WAF value is reduced by 17.5%, as compared to those of *GC-CB*. When the buffer cache size is 3 GB, the performance is improved less since the amount of zombie data available to the prototype SSD gets smaller. Note that when the buffer cache size is 4 GB, the amount of zombie data that the prototype SSD is aware of is about 1.28 GB on average while the amount is only about 870 MB when the buffer cache size is 3 GB. The ratio of the amount of zombie data in the prototype SSD to the amount of dirty data in the buffer cache is about 70% in both cases.

Table 5 shows the internal statistics of the Oracle DBMS in the TPC-C1 configuration. Whenever a process of the Oracle DBMS waits for something, it is recorded using one of the predefined wait events. By examining the wait records, users can identify performance bottlenecks and possible causes of the bottlenecks. Table 5 shows the top three wait events that have the most significant percentage of wait time, along with the average wait times of the wait events for each garbage collection algorithm. According to the official technical document for the Oracle DBMS [20], the top three wait events are possibly caused by slow I/O systems, which means that the performance bottleneck in this evaluation is the prototype SSD. We can confirm that the I/O performance of the prototype SSD is improved by utilizing the zombie-aware garbage collection so that the average wait times of the wait events are shortened by more than 10%. Thus, the TPS value is also raised as Fig. 10 depicts.

TABLE 5
The Average Wait Time of the Top Three Database Events

Event type	DB time(%)	Avg. wait time(ms)			
		GC-GD	ZGC-GD	GC-CB	ZGC-CB
db file sequential read	38.09	17	15	18	16
free buffer wait	23.95	389	313	369	348
write complete waits	22.22	20,333	17,147	20,874	18,462

We have also performed the TPC-C benchmark on the prototype SSD that is configured to use all 16 banks. Fig. 11 illustrates the results of both TPC-C1 and TPC-C2 when the buffer cache size is set to 4 GB. Similar to the previous experiments with 4 banks, *ZGC-GD* and *ZGC-CB* outperform *GC-GD* and *GC-CB*, respectively, in terms of both the TPS and WAF values. Especially, we observe that the performance is improved by more than 10% when the ratio of the over-provisioned extra flash blocks is 19% in the prototype SSD.

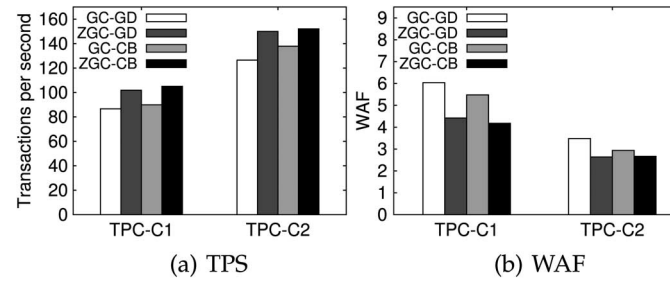


Fig. 11. TPS and WAF values when the prototype SSD is configured to use all 16 banks.

6 Related Work

Sivathanu et al. discussed data liveness at the block-level and described various storage optimization techniques that utilize information on data liveness [2]. They formalized the notion of data liveness in various types and presented two approaches, explicit notification and implicit detection, to impart the information on data liveness to storage systems. While they classified the data into either live data or dead data, this paper further divides the live data into (real) live data and zombie data and proposes enhanced garbage collection algorithms which consider zombie data as well as live/dead data.

There are several previous studies on improving the performance of flash storage systems with a new storage interface. Saxena et al. presented *solid-state cache (SSC)*, a flash device which has a new interface designed for caching [21]. State information (e.g., dirty, clean, or evicted) of every data cached inside SSC is imparted to SSC via the new interface. The FTL inside SSC utilizes the state information in performing garbage collection. Mesnier et al. presented a new scheme for classifying data in file systems and conveying the classification information to storage system [22]. Then, they proposed new allocation/eviction algorithms for SSD-based cache storage, which utilize the classification information. The classification information is piggybacked to storage systems via the Group Number field of the read/write SCSI command. In this paper, the zombie data information is piggybacked to the prototype SSD similarly via the reserved areas of SATA read/write commands.

For the efficiency of garbage collection, it is important for FTLs to identify soon-to-be-dead data (i.e., the data will be dead soon) and to avoid unnecessary copy operations of the soon-to-be-dead data during garbage collection. The hot-data-aware garbage collection of previous FTLs considers hot data (i.e., the data frequently updated in the *past*) as soon-to-be-dead data [8], [10], [12]-[14]. Instead, Zombie Chasing classifies zombie data as soon-to-be-dead data. This classification is more accurate since it is based on the write requests that will occur in the near future whereas the classification of the hot-data-aware garbage collection relies on the past write requests. Unless sudden power failure occurs, there is guarantee that zombie data will be dead in the near future since all dirty data in the buffer cache must be eventually written to disks. However, only a part of hot data is actually soon-to-be-dead data. Also, the hot-data-aware garbage collection cannot deal with a situation fully when the data classified into cold data is modified in the buffer cache. On the other hand, Zombie Chasing can recognize that the cold data will be dead soon. From the evaluations, we confirm that Zombie Chasing is more effective than the hot-data-aware garbage collection in reducing the overhead of garbage collection.

7 Conclusion

This paper proposes Zombie Chasing, an efficient flash management technique that considers dirty data in the buffer cache. First of all, we introduce a new data liveness state called the zombie state to denote live data that will be dead soon when the corresponding dirty data in the buffer cache is flushed into the storage system. Also, we devise enhanced garbage collection algorithms which utilize information on such zombie data inside SSDs. By distinguishing zombie data from real live data, the zombie-aware garbage collection reclaims invalid pages more efficiently than previous approaches. In order to show the feasibility of Zombie Chasing, we have implemented a prototype SSD based on a commercially-successful SSD controller and applied Zombie Chasing to the VFS layer of the Linux Kernel and the Oracle DBMS.

Through various evaluations using the in-house micro-benchmark and the TPC-C benchmark, we confirm that Zombie Chasing improves the SSD's performance effectively by reducing the garbage collection overhead under the workloads including lots of buffered random writes. In particular, our evaluation with the TPC-C benchmark shows that Zombie Chasing improves the TPS value by up to 22%, reducing the garbage collection overhead by about 26% at the same time. The performance improvement gained by Zombie Chasing rises as the buffer cache size increases or the degree of locality among write requests gets higher.

As future work, we plan to apply Zombie Chasing to popular file systems and study its impact on the performance of the file systems.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government

(MSIP) (No. 2013R1A2A1A01016441). This work was also supported by the IT R&D program of MKE/KEIT (No. 10041244, SmartTV 2.0 Software Platform).

References

[1] C. Ruemmler and J. Wilkes, "Disk shuffling," HP Lab., Tech. Rep., 1991.

[2] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Life or death at block-level," in *Proc. 6th USENIX Symp. Oper. Syst. Des. & Implementation (OSDI)*, 2004, pp. 379–394.

[3] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proc. 6th USENIX Security Symp. Focusing Appl. Cryptography (SSYM)*, 1996, pp. 77–90.

[4] C. Hyun, J. Choi, D. Lee, and S. H. Noh, "To TRIM or not to TRIM: Judicious triming for solid state drives," *presented at the 23rd ACM Symp. Oper. Syst. Principles (SOSP)*, 2011.

[5] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2010, pp. 187–200.

[6] J. Kim, H. Kim, S. Lee, and Y. Won, "FTL design for TRIM command," in *Proc. 5th Int. Workshop Softw. Support Portable Storage (IWSSPS)*, 2010, pp. 7–12.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2008, pp. 57–70.

[8] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-FTL: A memory-efficient flash translation layer supporting multiple mapping granularities," in *Proc. 8th ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2008, pp. 21–30.

[9] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Architectural Support Program. Languages Operat. Syst. (ASPLOS)*, 2009, pp. 229–240.

[10] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw.—Pract. Exp.*, vol. 29, pp. 267–290, 1999.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[12] P. Dongchul and D. H. C. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, 2011.

[13] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.

[14] M. Changwoo, K. Kangnyeon, C. Hyunjin, L. Sang-Won, and E. Young Ik, "SFS: Random write considered harmful in solid state drives," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 139–154.

[15] Computer Systems Laboratory at Sungkyunkwan University. (2010). *The OpenSSD Project* [Online]. Available: http://www.openssd-project.org.

[16] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proc. 11st USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 119–132.

[17] M. Saxena, Y. Zhang, M. M. Swift, A. C. A. Dusseau, and R. H. A. Dusseau, "Getting real: Lessons in transitioning research simulations into hardware systems," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 215–228.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

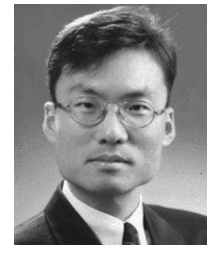
[19] Transaction Processing Performance Council (TPC). (2010). *TPC-C* [Online]. Available: http://www.tpc.org/tpcc.

[20] T. Morale, *Oracle Database Reference 11g R2(11.2)*, Oracle, Oct. 2010.

[21] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: A lightweight, consistent and durable storage cache," in *Proc. 7th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2012, pp. 267–280.

[22] M. P. Mesnier and J. B. Akers, "Differentiated storage services," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 45–53, Feb. 2011.

Youngjae Lee received the BS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Republic of Korea, in 2005 and 2014. He is currently a postdoctoral researcher at Sungkyunkwan University, Suwon, Republic of Korea. His research interests include flash memory, storage systems, and operating systems.



Jin-Soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently a professor at Sungkyunkwan University, Suwon, Republic of Korea. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Sang-Won Lee received the PhD degree from the Computer Science Department of Seoul National University, Republic of Korea, in 1999. He is an associate professor with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea. Before that, he was a research professor at Ewha Women University and a technical staff at Oracle, Korea. His research interest includes flash-based database technology.

Seungryoul Maeng received the BS degree in electronics engineering from Seoul National University (SNU), Republic of Korea, in 1977, and the MS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Republic of Korea, in 1979 and 1984, respectively. Since 1984, he has been a faculty member of the Department of Computer Science at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include micro-architecture, parallel processing, cluster computing, and embedded systems.