Reducing Asynchrony in Channel Garbage-Collection for Improving Internal Parallelism of Multichannel Solid-State Disks

LI-PIN CHANG and CHEN-YI WEN, National Chiao-Tung University

Solid-state disks use multichannel architectures to boost their data transfer rates. Because realistic disk workloads have numerous small write requests, modern flash-storage devices adopt a write buffer and a set of independent channels for better parallelism in serving small write requests. When a channel is undergoing garbage collection, it stops responding to inbound write traffic and accumulates page data in the write buffer. This results in contention for buffer space and creates idle periods in channels. This study presents a channel-management strategy, called *garbage-collection advancing*, which allows early start of garbage collection in channels for increasing the overlap among channel activities of garbage collection and restoring the balance of buffer-space usage among channels. This study further introduces *cycle filling*, which is a version of garbage-collection advancing tailored for the operation model of flash planes. Experimental results show that the proposed methods greatly outperformed existing designs of multichannel systems in terms of response and throughput. We also successfully implemented the proposed methods in a real solid-state disk and proved their feasibility in real hardware.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; B.3.2 [**Memory Structures**]: Design Style—*Mass storage*

General Terms: Design, Performance, Algorithms

Additional Key Words and Phrases: Solid-state disks, flash memory, multichannel architectures

ACM Reference Format:
Li-Pin Chang and Chen-Yi Wen. 2013. Reducing asynchrony in channel garbage-collection for improving internal parallelism of multichannel solid-state disks. ACM Trans. Embedd. Comput. Syst. 13, 2s, Article 63 (January 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2544375.2544383

1. INTRODUCTION

Flash-based solid-state disks are a popular design option for storage devices in smart phones, laptop computers, and even enterprise data centers. Flash memory is a type of erase-before-write nonvolatile memory. Solid-state disks implement a firmware layer called the flash-translation layer to emulate a collection of logical disk sectors and make the flash characteristics transparent to the host. Compared to mechanical disks, flash chips are small and do not require coolers for heat dissipation. Thus solid-state disks can stack an array of flash chips to exploit data-access parallelism. Using parallel chip structures is becoming important to high write throughput, as recent flash manufacturing technologies, including multiple-level cells (MLCs) [Samsung 2008], noticeably degrade the write performance of flash chips.

This work is supported in part by a research grant 98-2221-E-009-157-MY3 from National Science Council, Taiwan, ROC, and a joint research project with Global Unichip Corp.
Corresponding author's email: lpchang@cs.nctu.edu.tw.

DOI: http://dx.doi.org/10.1145/2544375.2544383

Modern solid-state disks include an array of flash chips using various chip structures. Advanced flash supports multiplane commands for parallel access to multiple flash planes within a flash chip [Samsung 2008; Micron Technology 2009]. Prior studies have found that flash-operation latencies are a more important issue in flash-storage performance than the bandwidths of flash buses and host interfaces. Kang et al. presented DUMBO [2007], which uses chip-level interleaving to hide flash operation latencies. Seong et al. presented Hydra [2010], which supports interleaving at both the chip level and the bus level. These methods involve hardware-oriented parallelism, and the firmware of the disk controller cannot directly program the internal behaviors of these chip structures.

Hardware-oriented designs achieve very high data throughput under sequential access patterns. However, realistic disk workloads exhibit irregular access behaviors, producing imbalanced utilizations of channel time and flash space. Thus, the firmware of disk controllers should adopt novel strategies for data management over parallel flash-chip organizations. This data management is concerned with *channels*, which refer to memory units that can independently process flash commands and transfer data. Multichannel architectures require strategies to decide the binding between page (sector) data and channels. Chang and Kuo [2002] introduced a dynamic striping policy that dispatches pages of hot data (i.e., frequently updated data) and non-hot data to the channels having the smallest average block erase count and the channels having the lowest space utilization, respectively. Park et al. [2011] and Dirik and Jacob [2009] proposed dispatching page data to channels using an RR (round-robin) policy to ensure the fair use of every channel.

Though dynamic channel binding of page data may have better flexibility of utilizing channel time and flash space, it does not guarantee the maximum page-level parallelism of sequential read, because it may map consecutive logical pages (sectors) to the same channel. Many commodity solid-state disks refuse to sacrifice their super-fast read performance, which has been the iconic advantage of flash-storage devices, and choose static channel binding, like RAID-0 style striping. Many prior studies have been conducted on the static channel binding of page data [Agrawal et al. 2008; Nam et al. 2011; Seong et al. 2010; Park et al. 2012]. With static channel binding, channels can be viewed as substorage devices that adopt their own instances of flash-translation layer for space management. A benefit of this approach is that any existing flash-translation layer designs can migrate to multichannel environments without modification.

Realistic disk workloads produce numerous small write requests [Chang 2010]. Serving small write requests with static channel binding often leads to poor channel utilization, because small requests do not access all channels. In addition, when a channel initiates garbage collection, the other channels may have to wait, because they do not need to reclaim free space yet. To increase the channel utilization, prior studies proposed using a write buffer to collect small write requests and have the write buffer flush buffered pages to the largest number of channels in parallel [Seong et al. 2010; Park et al. 2012].

With a write buffer, a channel can continue to serve its buffered page data, while other channels are undergoing garbage collection. However, garbage collection is a time-consuming task. The experiments in this study indicate that under the disk workload of a Windows desktop, channels spend nearly half of their active time in garbage collection. A channel stops responding to page write requests during garbage collection. As a result, the inbound write traffic starts accumulating the page data of this channel in the write buffer. Because the other channels are still removing their page data from the write buffer, they frequently run out of their buffered pages and must wait until the garbage-collecting channels resume consuming their buffered pages and relinquish some buffer space.

This study introduces a channel management strategy, called *garbage-collection advancing*. It allows early start of garbage collection in channels to increase the overlap among channel activities of garbage collection. The goal is to restore the balance of buffer-space usage among channels and alleviate the contention for buffer space. Because garbage-collection advancing is designed to be independent of flash-translation layers, it works with various mapping schemes, including hybrid mapping and page-level mapping.

This study further presents a tailored version of garbage-collection advancing for flash planes, called *cycle filling*. Flash planes are memory units within flash chips, and a flash chip can use multiplane commands for concurrent plane operations. Multiplane commands are a common feature of commodity flash chips. With multiplane commands, all the involved planes must perform the same type of flash operation (i.e., read/write/erase). Cycle filling synchronizes all channels' garbage-collection activities without restricting flash addressing. In other words, planes can still independently process flash commands with their own block/page addresses.

The rest of this article is organized as follows. Section 2 explains flash characteristics, multichannel architectures, and the problem of buffer-space contention. Section 3 proposes garbage-collection advancing and cycle-filling methods to address this problem. Section 4 shows simulation results of the proposed methods and a case study based on a real solid-state disk. Finally, Section 5 concludes this study.

2. PROBLEM FORMULATION

2.1. Flash-Translation Layer

NAND flash comprises an array of blocks, each of which has a fixed number of pages. The typical sizes of pages and blocks are 4KB and 512KB, respectively [Samsung 2008]. Read and write operations are carried out in terms of pages, but a page must be erased before it is rewritten. However, because flash erases in terms of block, erasing a block can involve other useful page data in this block. In addition, a typical flash block endures only about 10K write-erase cycles before it becomes unreliable.

Flash storage devices implement a firmware layer called *flash-translation layer* to emulate a collection of logical disk sectors and hide flash characteristics from the host. To avoid erasing a block before rewriting each page, this flash-translation layer updates page data out of place (i.e., it writes new data to other free flash space and marks old page data invalid). The flash-translation layer uses address mapping to translate logical sector numbers into flash addresses. Figure 1 depicts two typical mapping schemes. Figure 1(a) shows *page-level mapping*, which uses a page-mapping table to store all mapping pairs of logical sector numbers (*lsns*) and physical page numbers (*ppns*). With page-level mapping, the flash-translation layer writes new data to any free space in flash. DFTL [Gupta et al. 2009] is a representative design of page-level mapping. Figure 1(b) illustrates *hybrid mapping*, which uses a block-mapping table to map logical block numbers (*lbns*) to physical/flash block numbers (*pbns*). To optimize small-write performance, hybrid mapping stores sector updates in a pool of flash blocks, called the *log buffer*, and uses page-level mapping for the sector data in the log buffer. FAST [Lee et al. 2007] is a representative design of hybrid mapping.

Serving write requests consumes free space in flash. When the flash-translation layer uses up free flash space, it begins erasing flash blocks to reclaim the flash space occupied by invalid (outdated) data. Let the block be erased be the *victim block*. For the page-level mapping shown in Figure 1(a), the flash-translation layer first copies the two pages of valid data a' and b' to other free space and then erases the victim block V. For the hybrid mapping shown in Figure 1(b), the flash-translation layer first collects the valid data a' and b' from the victim block V (which is the oldest log block) and the

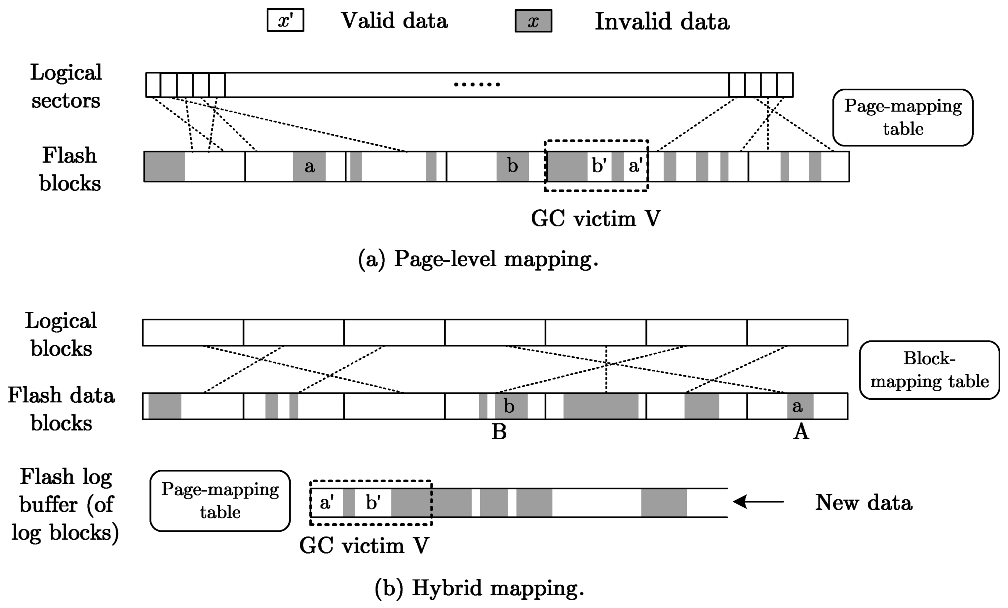


Fig. 1. Two mapping schemes of the flash-translation layer.

valid data from the flash blocks A and B (in which the old data a and b reside), writes all the valid data to two new flash blocks, and then erases the three flash blocks A, B, and V. These copy and erase activities described are called *garbage collection*. The flash-translation layer starts garbage collection upon running out of free space, and it stops responding to incoming write requests during garbage collection.

2.2. Multichannel Architectures

In this study, a channel represents a logical memory unit that independently processes flash commands and preforms data transfer. Nonprogrammable flash-chip organizations, such as gangs and interleaving groups, are considered to be part of the channels.

A basic multichannel architecture uses *synchronized channels*. This design has all channels perform the same flash command with the same flash address. Logically, this method scales up the page size and block size by the total number of channels. Let a *super (flash) page* and a *super (flash) block* denote the units formed by the parallel pages and blocks in channels, respectively. For example, the first super block of a four-channel architecture consists of the four channels' first flash blocks. Synchronized channels read/write and erase in terms of super pages and super blocks, respectively.

Figure 2(a) shows an example of handling six write requests w_1 to w_6 using four synchronized channels. The units of the timeline at the bottom of Figure 2(a) are *channel cycles*, which are virtual time units for measuring the lengths of channel activities. This example assumes that every channel runs out of its free space after writing three pages of data. To handle write requests smaller than a super page, the flash-translation layer must first read the unmodified pages of a super page, combine the new data with the unmodified data, and then write a super page. This procedure is called the *read-modify-write* procedure. The white boxes and gray boxes in Figure 2(a) represent writing the host data and writing the unmodified data, respectively. This design delivers high write throughput if the host write pattern is sequential. However, when the host issues many small write requests, synching channel operations unnecessarily consumes free space

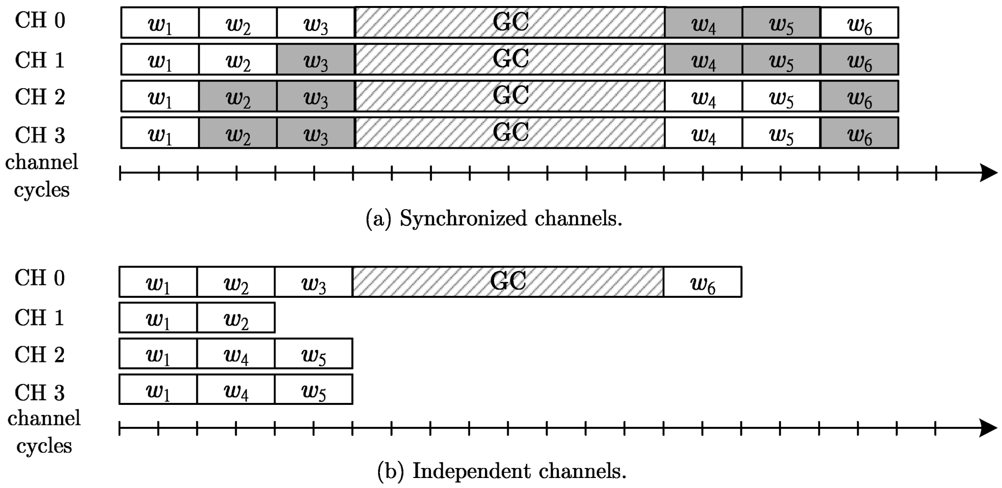


Fig. 2. Handling six write requests with (a) four synchronized channels and (b) four independent channels and a write buffer. The "GC" boxes stand for garbage-collection activities. The gray boxes represent page writes contributed by read-modify-write operations. Synchronized channels suffer from a high read-modify-write overhead, while independent channels experience low channel utilizations during garbage collection.

in channels and severely degrades overall write performance. For example, the channel system in Figure 2(a) writes 24 pages to modify 12 pages.

Let the *utilization* of a channel be the time fraction of the active periods of the channel. Independent channels carry out their own flash operations with their own data and addresses, and they need not synchronize their flash operations. Thus, an unreferenced channel can remain idle when the channel system is serving a small write request. However, idle channel cycles decrease channel utilization. A straightforward way to improve channel utilization is to adopt a small write buffer that collects incoming write requests and then issues page writes to the largest number of channels in parallel. Figure 2(b) shows that using independent channels reduces the total write traffic by 50%, compared to Figure 2(a). However, the response time of w_6 is not much improved because it triggers garbage collection in Channel 0. In addition, the overall channel utilization is poor if the other channels have no more pages to write. The following section discusses how this problem can occur.

2.3. Buffer-Space Contention

This section describes the problem of buffer-space contention caused by garbage collection activities in channels. This contention for buffer space creates idle cycles in channels. In the following discussion, the binding between logical sectors and channels is static. The write buffer is shared by all channels, and its capacity is not unlimited. A piece of page data in the write buffer is called a *buffered page* or a *dirty page*.

In the example of Figure 2(b), when the first channel starts collecting garbage, the other three channels do not require garbage collection yet. For better channel utilization, Channels 1, 2, and 3 can continue processing their buffered pages. New write requests arrive at the write buffer at the same time when these three channels consume their dirty pages. Because Channel 0 stops responding to write requests, the inbound write traffic starts accumulating the dirty pages of the Channel 0 in the write buffer. During the same time period, the other three channels consume their buffered pages. Once the three channels run out of their dirty pages (at this time, the channel is filled up with the dirty pages of Channel 0), they must wait until the Channel 0

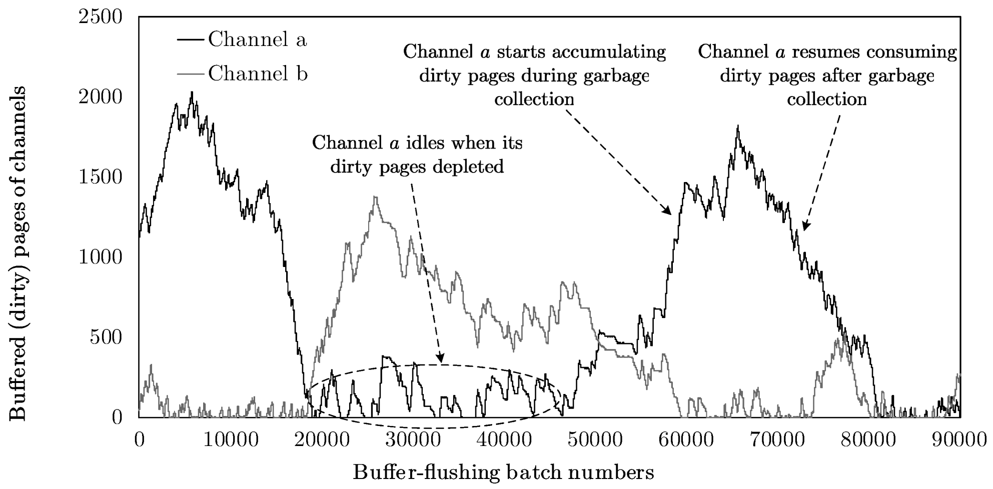


Fig. 3. The runtime amounts of channels' dirty pages of two channels in an eight-channel system. A channel increases its amount of dirty pages when it is undergoing garbage collection, while this amount of a non-garbage-collecting channel decreases. Because the buffer space is limited, once the accumulated dirty pages fill up the buffer space, the other channels have to wait.

completes its garbage collection and resumes removing its dirty pages from the write buffer.

Figure 3 depicts the runtime amounts of all channels' dirty pages during an experiment with eight independent channels. The write buffer is as large as 2,048 pages. Channels *a* and *b* are two among these eight channels. The x-axis depicts the batch numbers of buffer flushing. Between 20,000 and 45,000 batches, Channel *a* was idle involuntarily because it ran out of dirty pages. This is because, in this time period, the garbage-collecting channels, including Channel *b*, used up all the buffer space such that the buffer stopped accepting new write requests. After 45,000 batches, Channel *a* started garbage collection, while Channel *b* was processing dirty pages. Interestingly, during the period between 60,000 and 70,000 batches, Channels *a* and *b* switched their roles of idling and collecting garbage.

This example reveals a potential performance issue of using multichannel architectures. Garbage-collection activities deteriorate the balance among channels' usages of buffer space, creating idle cycles in channels. Such idle cycles usually do not affiliate with a particular channel, because every channel needs garbage collection from time to time. In this study, a channel is experiencing *transient congestion* if it has accumulated too many dirty pages such that another channel must wait. For example, between the 60,000 and 70,000 batches in Figure 3, Channel *a* is experiencing transient congestion.

3. CHANNEL MANAGEMENT

3.1. Channel Binding and Write Buffering

This study focuses on improving the utilization of independent channels. Let every channel adopt its own instance of flash-translation layer. Let the channel binding of logical sectors be static. Static binding is widely used in commodity solid-state disks and had been extensively studied in prior work [Kang et al. 2007; Agrawal et al. 2008; Seong et al. 2010]. Let a logical page be as large as a flash page, and let a logical page store contiguous logical sectors. The mapping of logical pages to channels follows RAID-0 style striping. This mapping maximizes the page-level parallelism of sequential read. Next consider an example in which the logical sector size and the flash page size are

512B and 4KB, respectively. Let a logical sector number (*lsn*) be of 32 bits, and let there be eight channels. The format of *lsn* is

[logical page number:29 [channel number:3]] [sector number:3].

For example, the sectors at *lsns* 0 to 7 are mapped to the first logical page in the first channel, and the sectors at *lsns* 8 to 15 are mapped to the first logical page in the second channel.

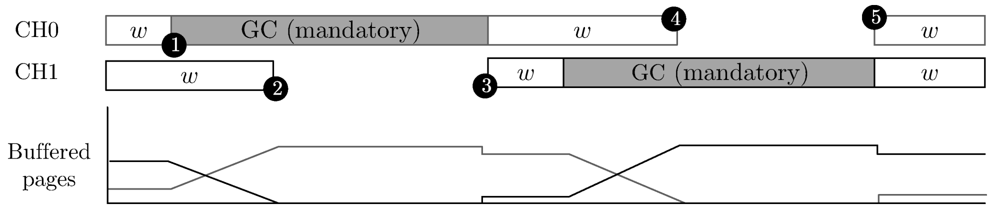
As mentioned in Section 2.2, serving small write requests without a write buffer can leave idle cycles in independent channels. This problem can be addressed using a write buffer. Let the memory space of this buffer be shared by all the channels. This buffer accepts write requests from the host as long as it has free space. When the buffer is full, it starts flushing dirty pages to channels. For the maximum page-level write parallelism, the write buffer writes to the largest number of channels (ideally, all the channels) at the same time. The buffer does not write to a channel if this channel has no buffered pages or is undergoing garbage collection.

One additional benefit of using a write buffer is to maintain page-level sequentiality of continuous write bursts. Recall that sectors are smaller than pages. Write requests may not be aligned to page boundaries, and two continuous write requests can partially update the same logical page twice. The write buffer concatenates these two sector-continuous but not page-aligned write requests and restores their page-level sequentiality. The buffer replacement algorithm in this study adopts a first in, first out (FIFO) policy. Because the problem of buffer-space contention are likely independent of buffer-replacement policies, the channel management strategies proposed in the following sections can be used with existing buffer-replacement algorithms, including those Kim and Ahn [2008], Kang et al. [2009], and Chang and Su [2011].

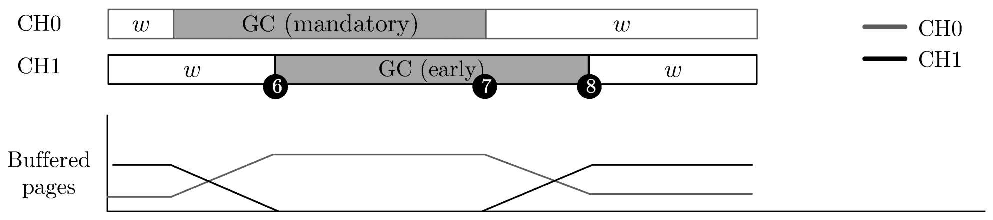
3.2. Garbage-Collection Advancing

3.2.1. The Basic Design. This section introduces a technique that alleviates transient congestion in channels. As explained in Section 2.3, garbage collection in channels increases the imbalance of buffer-space usage among channels, and this imbalance will further create idle cycles in non-garbage-collecting channels. This study proposes increasing the overlap among channel activities of garbage collection to restore the balance of channels' buffer-space usage. Let garbage collection be *mandatory* if it is triggered by the event of running out of free flash space (i.e., not running out of dirty pages). Now, whenever a channel has no more dirty pages to write, it checks whether there is another channel performing mandatory garbage collection. If this condition is true, then this channel moves garbage collection forward and starts reclaiming free space. This technique is referred to as *garbage-collection advancing*. Let garbage collection be *early* if it is triggered by garbage collection advancing. Early garbage collection is not to recycle idle channel cycles for garbage collection. An idle channel remains idle if there are no other channels performing mandatory garbage collection.

Figure 4(a) shows a scenario involving two channels without garbage-collection advancing. The bottom half of Figure 4(a) depicts the runtimes of channels' dirty pages. Assume that the write requests of the two channels arrive at the same rate. At time point 1, Channel 0 starts garbage collection and stops processing incoming dirty pages. Its amount of dirty pages increases, while that of the other channel, Channel 1, decreases, because Channel 1 is still consuming its dirty pages. At time point 2, the buffer is entirely filled up with Channel 0's dirty pages, so Channel 1 must wait. At time point 3, Channel 0 finishes collecting garbage and resumes consuming its dirty pages. Upon that, Channel 0 relinquishes some buffer space, and dirty pages of Channel 1 arrive at the write buffer so Channel 1 resumes writing. After this, analogously, Channel 1 starts garbage collection and later becomes congested at time point 4, and this time



(a) Without garbage-collection advancing.



(b) With garbage-collection advancing.

Fig. 4. Channel schedules with and without garbage-collection advancing. (a) Channels 0 and 1 get congested in turn, and the buffer-space contention creates idle periods in these two channels. (b) Channels 0 and 1 overlap their garbage collection activities using garbage-collection advancing. No channel is idle, and the channel schedule expedites.

Channel 0 must wait. Channel 1 completes its garbage collection at time 5, and both channels resume writing pages.

Figure 4(b) shows how garbage-collection advancing works. In Figure 4(b), shortly before time point 6, Channel 0 performs mandatory garbage collection and accumulates dirty pages. At time point 6, Channel 1 runs out of dirty pages, and then starts early garbage collection because Channel 0 is undergoing mandatory garbage collection. Between time points 6 and 7, Channels 0 and 1 are both carrying out garbage collection, and no channel is idle. At time point 7, Channel 0 finishes its mandatory garbage collection and resumes processing its dirty pages. Between time points 7 and 8, the early garbage collection of Channel 1 is still in progress, and the buffered pages of Channel 1 increase. At time point 8, Channel 1 finishes its early garbage collection and resumes processing its dirty pages. In this example, these two channels never idle, and the channel schedule length of Figure 4(b) is much shorter than that of Figure 4(a).

We must emphasize that an idle channel starts early garbage collection only if there are one or more channels performing mandatory garbage collection. This is because after a channel finishes its mandatory garbage collection, it will resume consuming buffered pages and freeing buffer space, and new dirty pages of the other channels will arrive at the write buffer. Many channels then become busy writing their new dirty pages, and thus, starting early garbage collection without any ongoing mandatory garbage collection does not increasing the overlap among channel activities of garbage collection.

3.2.2. Preemption of Early Garbage Collection. When a channel is carrying out early garbage collection, its dirty pages could arrive at the write buffer at any time. For example, at time point 7 of Figure 4(b), Channel 1 receives new dirty pages while it is performing early garbage collection. Because a channel still has free flash space when it starts early garbage collection, it may try to suspend the undergoing early garbage collection to process its new dirty pages. However, the procedure of garbage

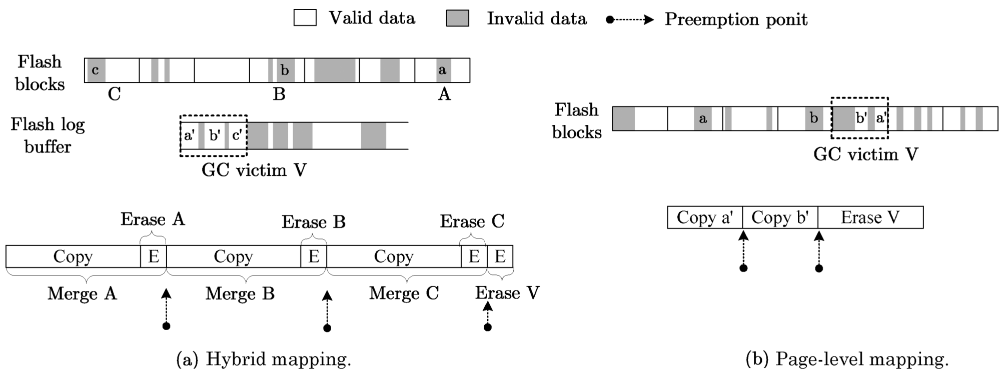


Fig. 5. Preemption points of garbage collection.

collection cannot be arbitrarily interrupted, or the integrity of the flash-translation layer will be compromised. As a result, this channel waits until its early garbage collection completes, and unnecessarily delays processing its new dirty pages. As depicted in Figure 4(b), between time points 7 and 8, Channel 1 waits until its early garbage collection completes.

This study proposes adding preemption points to garbage-collection activities. A preemption point is a time at which garbage collection can safely be suspended without corrupting the integrity of flash-translation layer. Figure 5 shows examples of garbage collection for hybrid mapping and page-level mapping and their possible preemption points. For hybrid mapping in Figure 5(a), garbage collection recycles the oldest log block V. The victim block V has valid page data related to three (flash) data blocks A, B, and C. For each of these data blocks, garbage collection first creates a newer version of the data block, collects valid page data from the old data block and the victim block V, writes the valid data to the new data block, and then erases the old data block. After all these, garbage collection erases the victim block V. This procedure is called *full merge*. Preemption points of full merge can be placed between the copy-and-erase operations for two data blocks, as Figure 5(a) shows. Garbage collection for page-level mapping is relatively simpler. As Figure 5(b) shows, it copies valid page data out of the victim block and then erases the victim block. Preemption points can be between any two consecutive page copy.

A channel suspends early garbage collection at the next preemption point when its dirty pages arrive at the buffer. It can later resume the preempted garbage collection. In hybrid mapping, a channel always resumes the previously preempted full merge for the same block, because the victim block is always the oldest log block. For example, in Figure 5(a), if new dirty pages arrive between the first and the second preemption points, garbage collection will suspend after it merged the data blocks A and B and leave the valid data c' in the victim block V. Later on, garbage collection (early or mandatory) continues to merge the data block C and erase victim block V. It is possible that new page writes invalidate the valid data c' before garbage collection resumes. In this case, garbage collection goes directly to erase the victim block V. In page-level mapping, garbage collection can choose different victim blocks before and after its suspension because the new page writes that arrive after the suspension of garbage collection can change the best candidate for garbage collection. Consider that in Figure 5(b), garbage collection suspends after copying the data a'. After the host write some data to the storage device, there could be a better victim block whose page data are all invalid. In this case, the resumed garbage collection goes to the new victim block.

3.3. Cycle Filling

This section introduces an improved version of garbage-collection advancing. This method is optimized for small write buffers and, more importantly, it is compatible with flash plane operations.

3.3.1. Initiator and Followers. A channel can suspend its early garbage collection upon the arrival of new dirty pages. However, this suspension is not effective until the next preemption point. To reduce this waiting time, it is desirable to align one of the preemption points of early garbage collection to the preemption points of mandatory garbage collection (but mandatory garbage collection needs not be preempted). This way, as soon as a channel finishes its mandatory garbage collection, other channels can suspend their early garbage collection. However, this alignment still inserts a short idle period before a channel can start early garbage collection. In addition, if there are multiple channels performing mandatory garbage collection, it will be difficult to decide which instance of mandatory garbage collection a channel should align its early garbage collection to. Since a root cause of idle cycles is the asynchrony among garbage-collection activities in channels, this study proposes that a channel should initiate early garbage collection as soon as another channel starts mandatory garbage collection. This way, the preemption points of early garbage collection are all aligned to that of mandatory garbage collection. This technique, called *cycle filling*, synchronizes the initiation of garbage collection in every channel.

Let a channel that initiates mandatory garbage collection be the *initiator*. There can be only one initiator. If many channels initiate mandatory garbage collection at the same time, tie-breaking is arbitrary. All channels other than the initiator are *followers*. When a channel (the initiator) initiates mandatory garbage collection, all the other channels (followers) must start early garbage collection simultaneously. With cycle filling, all the followers fill their channel cycles with early garbage collection during the time frame of the mandatory garbage collection for the initiator. The early garbage collection of all the followers is suspended as soon as the initiator finishes its mandatory garbage collection. The benefits of cycle filling are twofold. First, every instance of early garbage collection has a preemption point that is aligned to the end of mandatory garbage collection. Second, it completely overlaps the garbage-collection activities in channels and thus levels the buffer-space usage of all channels. The following section shows how cycle filling works with hybrid mapping and page mapping.

3.3.2. Cycle Filling for Hybrid Mapping and Page Mapping. Figure 6 shows a scenario of using cycle filling with hybrid mapping. Let the flash-translation layer be FAST. Figure 6(a) depicts the layout of data in the flash log buffer and flash data blocks. In this example, Channel 1 is the initiator, while Channel 0 is the follower. Figure 6(b) shows the channel schedules. The initiator (Channel 1) triggers mandatory garbage collection to erase its victim block V_i, which is related to three data blocks A, B, and C. The entire full merge procedure involves (1) three times of copying valid data from an old data block and erasing the old data block and (2) erasing the victim block. During this period, the follower (Channel 0) performs the same type of operations. It first chooses its own victim block V_{f_1}, which is related to two data blocks D and E, and then repeats the procedure of valid data copying and block erasing for the data blocks D and E. At this time, the victim block V_{f_1} has no valid data and is ready for erasure. However, to align with the initiator's operations, the follower proceeds to its second-best victim block V_{f_2}, which has a piece of valid data f' related to the data block F. The follower then performs data copy and block erase for this third data block F. At this point, both the initiator and the follower can erase their victim blocks V_i and V_{f_1} in parallel.

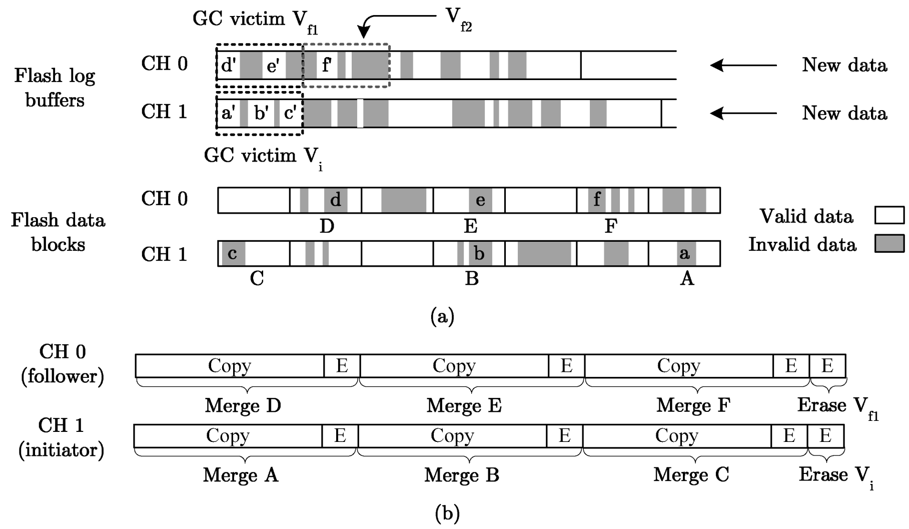


Fig. 6. Cycle filling with hybrid mapping. Channels 0 and 1 are the follower and the initiator, respectively. (a) Data in log blocks and data blocks, and (b) the channel schedules.

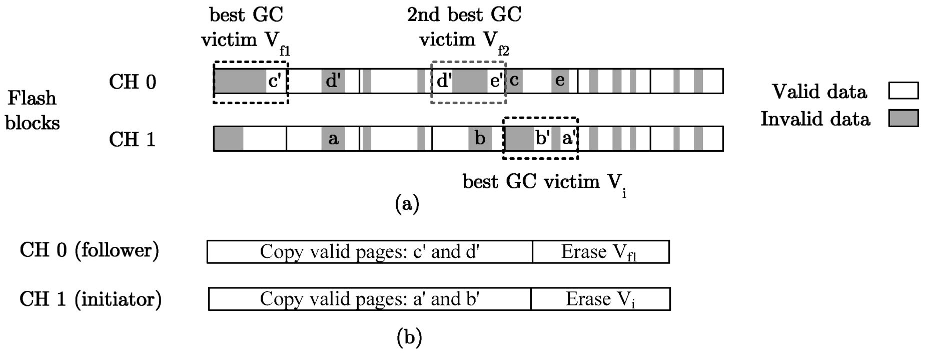


Fig. 7. Cycle filling with page-level mapping. Channels 0 and 1 are the follower and the initiator, respectively. (a) Data in flash blocks and (b) the channel schedules.

Note that in Figure 6, the follower performs valid data copying and block erasing for one extra data block F. This is to have the follower perform the same type of operation as the initiator does. Without this design, the follower must wait until the initiator erases its victim block. The side effect of copying valid data and erasing a data block related to the second best victim block V_{f_2} can be very limited, because the data f' is in the second oldest log block and is unlikely to be invalidated before the log block V_{f_2} becomes the oldest block. In contrast to the example in Figure 6, if the victim block of the initiator has fewer valid data than the victim block of the follower, then the initiator will start erasing a victim block earlier than the follower. As soon as the initiator starts erasing a victim block, the follower should halt early garbage collection, since its victim block is not eligible for erasure. Thus, cycle filling may produce idle periods whose lengths are up to the duration of one block erase cycle.

Figure 7 shows an example of using cycle filling with page-level mapping. Channel 1, as the initiator, copies the valid page data a' and b' out of its victim block V_i. In the meantime, to align to the initiator's operations, Channel 0 (as the follower) copies valid page data c' out of its best victim V_{f_1}. After copying c', Channel 0 finds no valid data in its victim block V_0, so it proceeds to its second-best victim V_{f_2} and copies the valid data d'. After both the initiator and the follower have copied two pieces of valid data, they erase their victims V_i and V_{f_1} in parallel.

3.3.3. Cycle Filling for Flash Planes. Commodity solid-state disks employ hybrid architectures of planes and channels. This section introduces the operation model of flash planes and shows how cycle filling works with planes.

Planes are programmable memory units within a flash chip. The flash-translation layer can operate memory planes using multiplane commands. Multiplane commands are now a common feature of many flash products [Samsung 2008; Micron Technology 2009; Open NAND Flash Interface 2011]. However, with multiplane commands, all planes of a chip must perform the same type of flash operation (e.g., read, write, or erase). For example, a plane cannot write a page while another plane erases a block. Multiplane commands can send different flash addresses to different planes. However, flash products may have different restrictions on plane addressing. For example, some flash products can send different block addresses to planes, but the page addresses within blocks must be the same [Micron Technology 2009]. This study assumes that there is not any restriction on plane addressing.

Typical solid-state disks adopt multiple independent channels, each of which controls a flash chip of multiple planes. Because multiplane commands require every plane to perform the same flash operation, many commodity solid-state disks treat planes like synchronized channels. As mentioned in Section 2.2, synching plane address/operation effectively enlarges the page size and block size, introducing read-modify-write overheads.

With cycle filling, the initiation of mandatory garbage collection switches all channels from serving buffered pages to collecting garbage. Cycle filling is compatible with multiplane commands, because all channels (planes) always perform the same flash operation during garbage collection, as Figures 6 and 7 show, but the flash addresses sent to different channels (planes) need not be the same. Thus, for example, with cycle filling, a solid-state disk of four channels and two-plane chips is effectively equivalent to having eight independent channels. In the case that one plane of a two-plane chip idles with cycle filling, the chip can switch back to one-plane commands.

This study assigns a channel threshold N_s to limit the runtime total number of spare blocks in each channel. A channel can start early garbage collection if it has no more than N_s spare blocks. This mechanism helps protect channels from pathological host write patterns. If the host generates much more write traffic to a channel than to the other channels, then both garbage-collection advancing and cycle filling would be little help in improving the overall channel utilization, but would produce unnecessary flash operations in the lesser-written channels. Table I summarizes the start and stop (suspend) conditions of garbage-collection advancing and cycle filling.

4. EXPERIMENTAL RESULTS

4.1. Experimental Setup and Performance Metrics

The experiments in this study consist of two parts. The first part (Sections 4.2 to 4.5) uses a trace-driven simulation to evaluate the proposed channel management methods. The second part (Section 4.6) is a case study based on a real solid-state disk.

The following experiments are based on three types of host disk-workloads. The first workload was collected from a Windows XP desktop under the daily use of an ordinary user. The user activities include Web surfing, word processing, program developing, and gaming. The second workload was generated by an industrial-standard disk benchmark tool Iometer [Open Source Development Lab 2003]. This benchmark tool is widely used to understand the random write performance of solid-state disks. The settings of

Table I. Conditions for a Channel to Start and Stop (Suspend) Early Garbage Collection

Method	Start conditions (must all hold)	Stop conditions (must all hold)
Garbage-collection advancing (Section 3.2)	(1) The write buffer is full (2) Dirty pages of this channel depleted (3) Any other channel is undergoing mandatory garbage collection (4) The total number of spare blocks in this channel is not larger than N_s	(1) Dirty page(s) of this channel arrive at the write buffer (2) The current time is a preemption point of early garbage collection
Cycle filling (Section 3.3)	(1) Any other channel initiates mandatory garbage collection (2) The total number of spare blocks in this channel is not larger than N_s	(1) The initiator channel finishes its mandatory garbage collection

Table II. Characteristics of the Three Experimental Workloads

Workload	Host OS	Volume size	File system	Average w. request size	Total written
Random	Win XP	16GB	NTFS	4KB	18.6GB
Multimedia	Win CE	20GB	FAT	60KB	19.8GB
Desktop	Win XP	40GB	NTFS	23KB	81.2GB

Iometer were 4KB requests with a 100% random write pattern on a 16GB disk. The third workload was generated by a Windows CE device that repeatedly copied and deleted music and video files to/from a 20GB SDHC card. Table II summarizes the characteristics of these workloads.

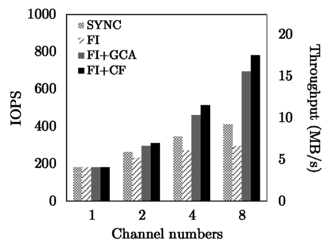
Each run of the experiments has several options, including the write-buffer size, the total number of channels, the flash-translation algorithm, and the channel architecture. The buffer size ranged from 32KB to 32MB, and the channel number ranged from one to eight. The flash-translation layer was either FAST [Lee et al. 2007] or DFTL [Gupta et al. 2009], which are based on hybrid mapping and page mapping, respectively. Channels were either synchronized (SYNC) or fully independent (FI). If channels are independent, there will be three more options: garbage-collection advancing (denoted as FI+GCA, or simply GCA), cycle filling (denoted as FI+CF, or simply CF), or no early garbage collection at all (i.e., FI). Entry-level flash storage devices, such as memory cards and USB thumb drives, commonly adopt synchronized channels. Independent channels are a popular design option of many commodity solid-state disks and have been studied in many prior researches [Agrawal et al. 2008; Nam et al. 2011; Seong et al. 2010; Park et al. 2012]. The experiments in this study ignore host read requests, because serving read requests does not trigger garbage collection and it is much faster than serving write requests.[1] To better understand how garbage-collection advancing helps improve the parallelism among channels and planes, our experiments mainly used the FIFO buffer replacement policy to rule out the interference from the replacement algorithms. For comparison, BPLRU [Kim and Ahn 2008] was also considered in the experiments.

The experiments adopted two major performance metrics. The first metric is write IOPS (or simply IOPS), which represents the average number of write requests completed per second. Any multichannel design should aim for a high IOPS. Our experiments also adopted two auxiliary submetrics: write throughput and write response.

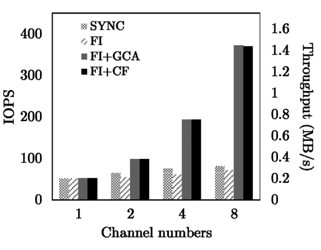
[1]Advanced multichannel designs employ separate queues for read requests and write requests and assign higher priorities to read requests than write requests [Seong et al. 2010]. Because read requests can interrupt ongoing write-related operations, early garbage collection will not delay the processing of read requests.

Table III. Experimental Parameters

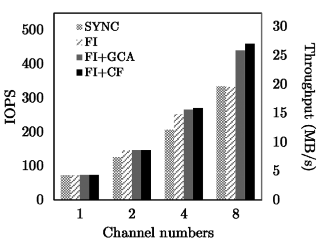
Parameter	Choices
Workload	Desktop, Random, Multimedia
Channel binding of pages	Static (RAID-0 striping)
Flash geometry	4KB (page), 512KB (block)
Flash over-provisioning ratio	10%
Mapping scheme	Page mapping (DFTL [Gupta et al. 2009]), Hybrid mapping (FAST [Lee et al. 2007])
Channel management method	SYNC, FI, GCA, CF
Channel number	1, 2, 4, 8
Planes per channel	1, 2
Buffer capacity	32KB, 2MB, 8MB, 32MB
Buffer management algorithm	FIFO, BPLRU [Kim and Ahn 2008]



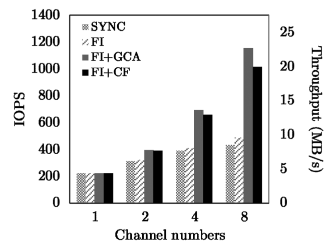
(a) Desktop, FAST.



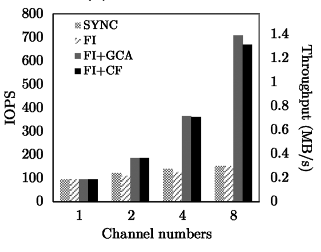
(b) Random, FAST.



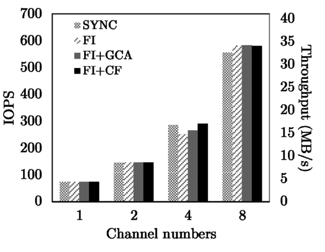
(c) Multimedia, FAST.



(d) Desktop, DFTL.



(e) Random, DFTL.



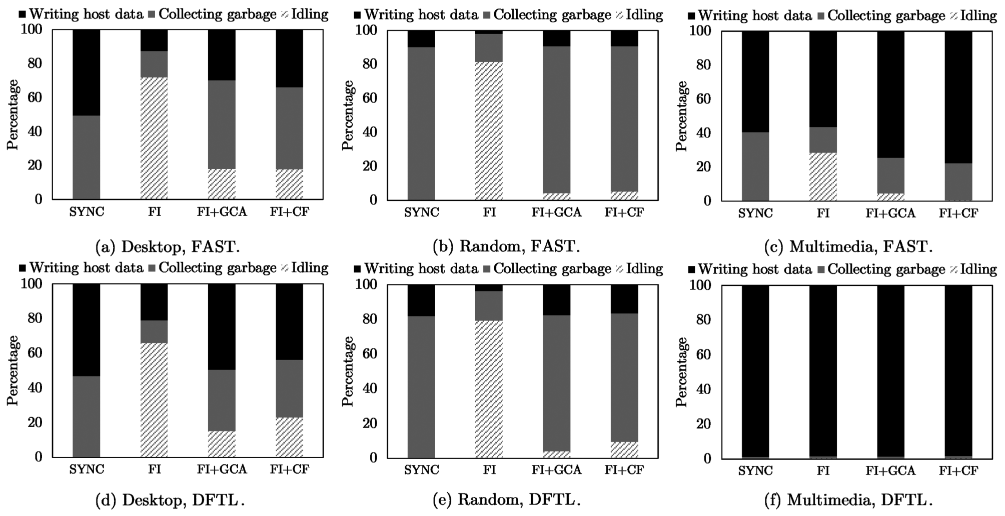
(f) Multimedia, DFTL.

Fig. 8. IOPS and bandwidth of using different number of channels under the three disk workloads. The write buffer size and the flash overprovisioning ratios were 32KB and 10%, respectively. The flash-translation layer was FAST in (a), (b), (c) and DFTL in (d), (e), (f).

The write throughput can be obtained by multiplying IOPS by the average request size (shown in Table II), while the response is the inverse of IOPS. The second metric is channel time, which represents how much channel time a channel system spends on certain activities. Channel time can be attributed to writing host data, collecting garbage, or idling. IOPS is susceptible to the cost of handling small write requests, while channel time is primarily affected by the parallelism among channels. In these experiments, the flash page size and block size were 4KB and 512KB, respectively. The time overheads for reading a page, writing a page, and erasing a block were 166μs, 906μs, and 1.5ms, respectively [Samsung 2008]. The threshold N_s of the total number of spare blocks for every channel was 200 in the experiments. Table III is a detailed listing of experimental parameters.

4.2. Channel Number

This section presents experimental results of using different numbers of channels. Figure 8 shows the IOPS and write throughput under the three disk workloads.



(a) Desktop, FAST. (b) Random, FAST. (c) Multimedia, FAST.

(d) Desktop, DFTL. (e) Random, DFTL. (f) Multimedia, DFTL.

Fig. 9. Percentages of channel time spent for writing host data, collecting garbage, and idling under the three disk workloads. The total number of channels was eight, the write buffer size was 32KB, and the flash overprovisioning ratio was 10%. The flash-translation layer was FAST in (a), (b), (c) and DFTL in (d), (e), (f).

Figures 8(a)–(c) show hybrid mapping, while Figures 8(d)–(f) show page-level mapping. The following explanation is also closely related to Figure 9, which shows the channel time utilization contributed by writing host data, collecting garbage, and idling.

Figures 8(a) and 8(d) show that under the Desktop workload, GCA (garbage-collection advancing) and CF (cycle filling) significantly improved their IOPS as the total number of channels increased. Hybrid mapping and page-level mapping exhibited similar trends in these improvements. In contrast, FI (i.e., fully independent channels without garbage-collection advancing) did not benefit significantly from using multiple channels under the Desktop workload, even though FI is also based on independent channels like GCA and CF. This is because FI suffers from severe contention for the buffer space, as previously described in Section 2.3. This contention created many idle cycles in channels. Figures 9(a) and 9(d) show that under the Desktop workload, GCA, CF, and even SYNC spent nearly half of their busy channel cycles for garbage collection. The contention for buffer space caused by these garbage-collection activities resulted in idle channel cycles, and this contention became worse when the channel number was large. When the channel number was eight, FI spent nearly 70% of the total channel time on idling. GCA and CF increased the overlap among channel activities of garbage collection to alleviate this contention, reducing this fraction to between 15% and 23%.

The results of the Random workload in Figures 8(b) and 8(e) are similar to those of the Desktop workload, but GCA and CF had a significant performance advantage over FI. The channel time utilization in Figures 9(b) and 9(e) shows that GCA, CF, and SYNC spent nearly 90% of channel time collecting garbage under this random write pattern, and FI wasted approximately 80% of its channel time in idling because of the severe contention for buffer space introduced by the intense garbage-collection activities. As a result, FI barely benefited from using multiple channels. As for SYNC, because all the write requests of the Random workload were small and the write pattern was purely random, most of the time, a buffered super page served only one small write request before the super page was evicted from the write buffer. Therefore, the IOPS of SYNC was subject to the read-modify-write overhead and hardly improved as the total number of channel increased.

Figures 8(c) and 8(f) show that under the Multimedia workload, all four methods benefited from using multiple channels, as the long and sequential write bursts in the workload are ready for parallel processing. However, these two sets of results reveal that the performance gain of using multiple channels was higher with page-level mapping than with hybrid mapping. This phenomenon is related to how the flash-translation later handles sequential write patterns. Recall that FAST predicts a write stream starting from the first logical page of a logical block to be sequential, and it creates an SW (sequential write) log block for this write stream. A write burst may write to the first logical pages of multiple logical blocks from different channels, creating SW log blocks in these accessed channels. However, when there are many channels, the write burst may not be long enough to write all the logical pages of the created SW log blocks. Because FAST cannot perform switch merge on partially-written SW log blocks, it reuses these SW log blocks as RW (random write) log blocks to better use free flash space. Figure 9(c) shows that the four methods spent 10% to 40% of the channel time performing full merge on the reused SW log blocks. Because of these full merge operations, FI wasted approximately 25% of channel time in idling, while GCA and CF significantly improved the idle channel time using garbage-collection advancing. In contrast, Figure 9(f) shows that page-level mapping produced very low garbage collection overhead with the four methods. This is because garbage collection for page-level mapping need not maintain page data sequentiality for logical blocks.

Results in Figure 8 show that GCA slightly outperformed CF in page mapping. This is because the density of preemption points for garbage collection in page mapping is high (see Figure 5(b)), and thus the latency of garbage-collection suspension for GCA in page mapping is low. Therefore, GCA can start early garbage collection as late as when a channel runs out of dirty pages and can suspend early collection almost as soon as new dirty pages arrive. In CF, early garbage collection and mandatory collection start at the same time, and early garbage collection can delay the processing of pending dirty pages. In summary, GCA is a good choice for page mapping, while CF is better for hybrid mapping.

4.3. Garbage-Collection Preemption

Section 3.2.2 introduced a preemption technique for early garbage collection. Using preemption points enhances GCA and CF in terms of write parallelism and synchrony of garbage collection in channels. Preemption points are optional to GCA, but mandatory to CF, because early garbage collection for CF must halt when mandatory garbage collection finishes. This experiment evaluated GCA with and without preemption points.

The IOPSs in Figure 10(a) show that using preemption points for hybrid mapping significantly benefited GCA. This is because the procedure of full merge is lengthy and can involve multiple data blocks, and thus the channel times in Figure 10(a) reveal that garbage collection preemption noticeably reduced the idle time in channels. The results also indicate that using preemption points significantly lowered the average response times (on the top of IOPS bars). In spite of the average response, real-time systems are also sensitive to the worst-case latency. Since GCA and CF are designed to be independent of flash-translation layers, we believe that the previously proposed algorithms for real-time garbage collection [Chang et al. 2004; Cho et al. 2009; Qin et al. 2012] can cooperate with GCA and CF. Figure 10(b) shows that using preemption points for page mapping did not affect the performance of GCA. This is because the procedure of erasing a victim is much shorter in page mapping than in hybrid mapping.

4.4. Write Buffer

4.4.1. Buffer Sharing vs. Buffer Partitioning. To avoid the contention for buffer space among channels, a naive approach is to equally divide the buffer space among channels into

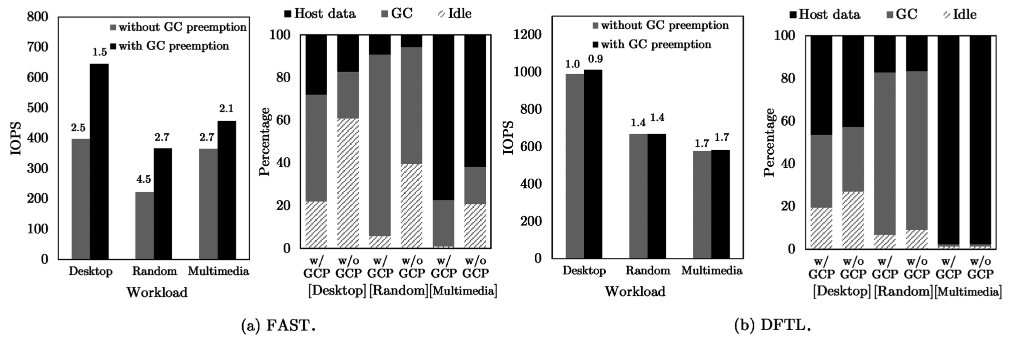


Fig. 10. IOPSs and channel time utilizations of GCA with and without preemption of garbage collection. The total number of channels was eight, the write buffer was 32KB, and the flash overprovisioning ratio was 10%. The average write response times (units: ms) are on the top of the IOPS bars.

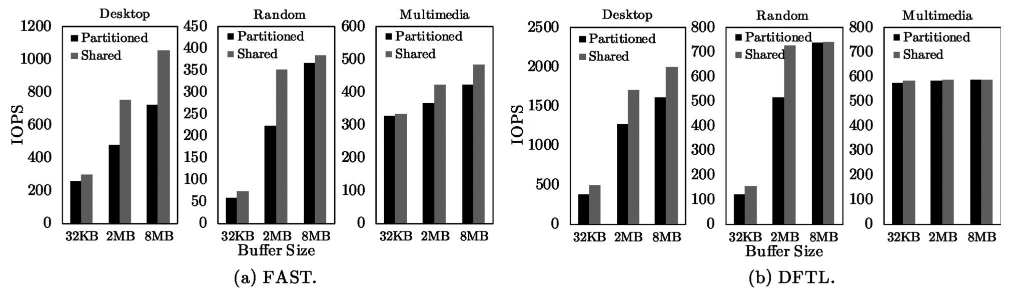


Fig. 11. Buffer partitioning versus buffer sharing. The total number of channels was eight, the flash over-provisioning ratio was 10%, and the write buffer size was 32KB. Channels were managed by FI.

small buffers. This experiment compared buffer partitioning against buffer sharing. In this experiment, channels were managed by FI to rule out the effects of early garbage collection. Results in Figure 11 show that buffer sharing significantly outperformed buffer partitioning under all workloads, especially when the write size was small (32 KB and 2 MB). This is because buffer partitioning degraded the overall utilization of buffer space. In buffer partitioning, when a channel is undergoing garbage collection, new write requests accumulated dirty pages in the buffer space of this channel. When this smaller buffer became full, the storage device would refuse to admit new write requests, even if the other channels still had free buffer space.

The results for the Desktop workload indicate that this problem was severe, and using large buffers barely resolved this problem. This is because the Desktop workload did not evenly write to all channels. We found that, by the end of the experiment, a channel received 21% more page writes from the host than another channel. A similar problem occurred in the case of the Random workload when the buffer size was 32KB or 2MB. Even though the Random workload uniformly writes to all channels, as garbage collection was costly under a random write pattern, a channel quickly ran out of its dedicated buffer space before garbage collection completed. For the case of the Multimedia workload, because this workload evenly writes to all channels and garbage collection is efficient for a sequential write pattern, the performance difference between buffer sharing and buffer partitioning was small.

4.4.2. Buffer Management Algorithm. This experiment considered BPLRU [Kim and Ahn 2008] as the buffer management policy. In this experiment, the block-padding threshold of BPLRU was assigned to the best value according to the workload and the buffer size.

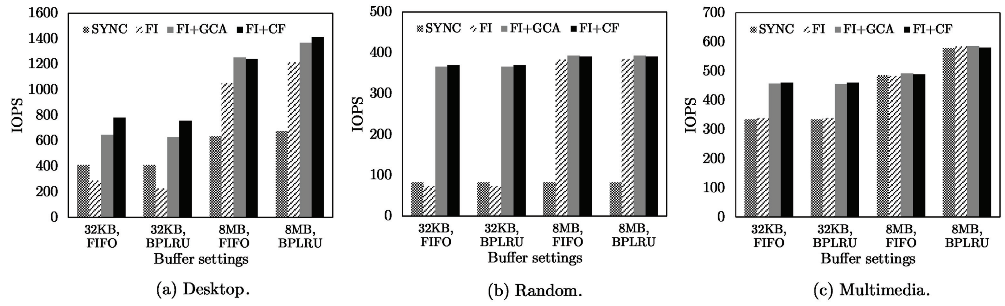
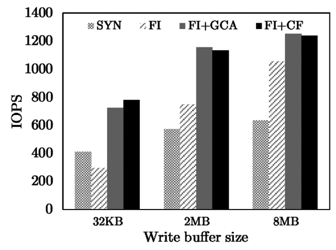


Fig. 12. IOPS of using FIFO and BPLRU as the buffer replacement algorithms. The total number of channels is eight, the flash overprovisioning ratio was 10%, and the FTL was FAST.

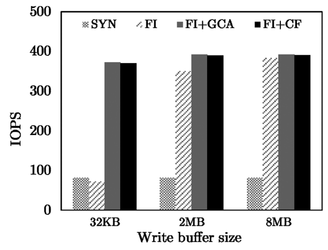
The FTL was FAST. Figure 12 compares the results of BPLRU and FIFO for buffer sizes 32KB and 8MB. When the buffer size was 32KB, all the four methods (SYNC, FI, GCA, and CF) had nearly the same performance with BPLRU and with FIFO. The main purpose of a write buffer of this size is to improve the page-level write parallelism (as mentioned in Section 2.2), and from this point of view, BPLRU and FIFO had the same efficacy.

Now focus on the results of 8MB buffers. Figure 12(a) showed that under the Desktop workload, all the four methods performed better with BPLRU. Compared to FIFO, BPLRU reduced the total page write by 12% in the case of CF. However, such an improvement did not affect the advantage of CF (and GCA) over FI and SYNC. We found that even though BPLRU reduced total overhead of garbage collection, the average number of data blocks involved by every full merge increased from 1.47 to 1.74. This is because buffer optimization techniques write only cold and nonsequential data to the flash log buffers [Kim and Ahn 2008; Kang et al. 2009; Chang and Su 2011]. As a result, the prolonged full-merge procedure aggravated the contention for buffer space among channels. Many replacement algorithms, including BPLRU, degrade into FIFO when the write pattern is random or sequential. This is true for the results of the Random workload in Figure 12(b). However, under the Multimedia workload, BLPRU benefited all four methods more than FIFO did, as Figure 12(c) shows. This is because BPLRU performed block padding: If the total number of dirty pages of a logical block is not smaller than the threshold of block padding, the write buffer will copy necessary page data from flash and compose a whole-block write request for efficient switch merge.

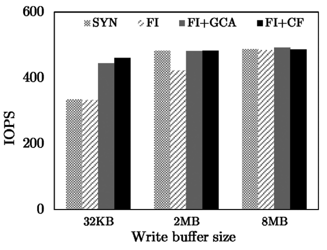
4.4.3. Buffer Capacity. This section reports results of evaluating GCA and CF with different buffer sizes. The buffer size varied from 32KB to 8MB. The results in Figure 13 show that under the Random and Multimedia workloads, GCA and CF with 32KB buffers obtained almost as high performance improvements upon SYNC and FI as they did with 8MB buffers. This is because, under these two workloads, the frequencies of garbage collection in channels were close, and GCA and CF effectively resolved the contention for buffer space using a 32KB buffer. For the case of the Desktop workload, because this workload did not evenly write to all channels (as mentioned in Section 4.4.1), we found that the most-written channel triggered 28% more times garbage collection than the least-written channel, and the asynchrony among channel activities of garbage collection was very high. Thus, the performance advantage of GCA and CF over FI and SYNC was significant for all buffer sizes, especially in hybrid mapping. Overall, using large buffers did not completely resolve the problem of buffer-space contention.



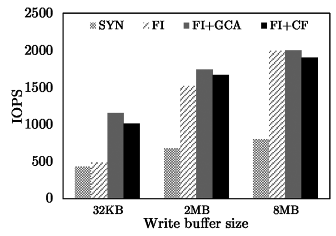
(a) Desktop, FAST.



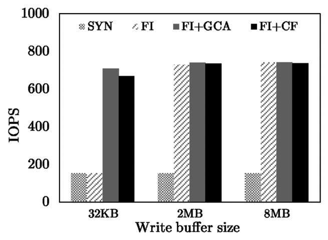
(b) Random, FAST.



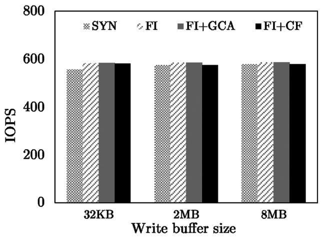
(c) Multimedia, FAST.



(d) Desktop, DFTL.



(e) Random, DFTL.



(f) Multimedia, DFTL.

Fig. 13. IOPS of using different sizes of the write buffer. The total number of channels is eight, the flash overprovisioning ratios was 10%.

4.5. Multiplane Structures

This section evaluates the results of using hybrid architectures of channels and planes. The multichannel system had eight channels, and each of these channels used two-plane commands. In other words, this architecture effectively had sixteen parallel memory units. The write buffer sizes varied from 64KB to 32MB. Among the four channel management methods, only CF can independently addresses flash pages within the two planes of a channel using two-plane commands while guaranteeing that the two planes will always perform the same type of flash operation. In contrast, GCA, FI, and SYNC must synchronize the block/page addresses of the two planes when using two-plane commands. This operation model of planes effectively doubles the block size and page size, introducing the read-modify-write overhead for synchronized planes.

Figure 14 shows the IOPS of the four channel-management methods with the hybrid channel-plane architecture under three disk workloads. Figures 14(a) and 14(d) show the results of the Desktop workload. With hybrid mapping, the performance advantage of GCA/CF over FI was very significant when the write buffer was not larger than 8MB. As previously mentioned, the channel system spent nearly half of the channel time in collecting garbage, and the garbage-collection activities resulted in severe competition for the buffer space and created idle periods in channels. Using a write buffer as large as 32MB reduced the performance gap between GCA/CF and FI, but CF still performed the best, because CF can use different flash addresses for two planes to serve two small write requests in parallel. SYNC performed the worst under the Desktop workload. This is because a super page in this architecture is sixteen times as large as a flash page, and the read-modify-write overhead of super pages severely limited its IOPS improvement.

The results of the Random workload in Figures 14(b) and 14(e) indicate that CF greatly outperformed GCA and FI, even when the write buffer was as large as 32MB. This is because the Random workload generated all small and random write requests, and CF efficiently handled these requests without the read-modify-write overhead for synchronized planes. In addition, GCA and FI noticeably outperformed SYNC when

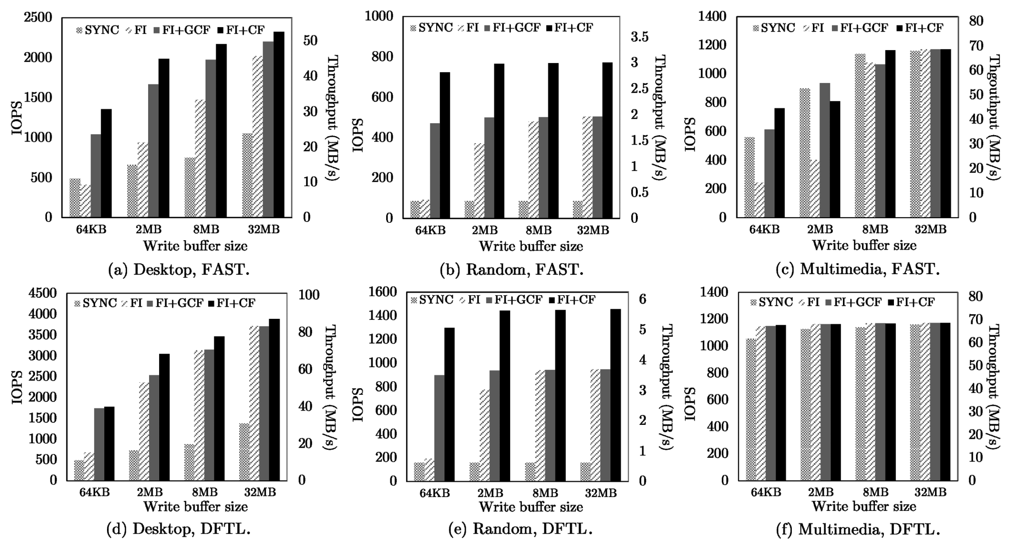


Fig. 14. IOPS of multichannel architectures with multiple flash planes. The total number of channels was eight, and each of the channels can issue two-plane commands. The flash overprovisioning ratios was 10%.

the buffer size was 2MB, and SYNC consistently performed the worst, regardless of the buffer size. These observations indicate that a key to IOPS improvement for this random write pattern is to reduce the read-modify-write overhead for synchronized planes and super pages.

Unlike the results of the Desktop and the Random workloads, the results of the Multimedia workload in Figures 14(c) and 14(f) show that the performance of all four methods was comparable when the write buffer was not smaller than 8MB. This is because the Multimedia workload consists of plenty of sequential write bursts, and serving these bursts with synchronized planes (GCA and FI) and even synchronized channels (SYNC) did not incur noticeable read-modify-write overheads. For the case of the Multimedia workload and hybrid mapping, as shown in Figure 14(c), FI performed much worse than SYNC when the write buffer was not larger than 2MB. This is because, with two-plane commands, FI erased in terms of super blocks whose size was twice as large as the flash block size. Using large super blocks deteriorated the efficiency of garbage collection and aggravated the contention for buffer space. We found that when the buffer size was 64KB, FI wasted about 57% of channel time on idling. This figure was much worse than the idle time of FI without planes under the same workload (i.e., 28%, as shown in Figure 9(c)).

A comparison of the results of hybrid mapping and page-level mapping in Figure 14 shows that increasing the buffer size is more beneficial to page-level mapping than to hybrid mapping. This is because the procedure length of erasing a victim block (including page copy and block erase) in page-level mapping is shorter than that in hybrid mapping (Figure 5). Therefore, the intensity of the contention for buffer space (caused by garbage collection) under page-level mapping is not as high as that under hybrid mapping. Therefore, adding a small buffer can effectively alleviate the contention for buffer space and remove idle cycles from channels. However, CF is the only method that is free from the overhead of read-modify-write, because it can issue different flash addresses to planes when using multiplane commands. This advantage is orthogonal to the efficacy of using large write buffers.

Table IV. IOPS of Using a GP-5086-Based Solid-State Disk and the Trace-Driven Simulator

	Desktop	Random	Multimedia
(0 KB, SYNC, real SSD)	292	71	112
(0 KB, SYNC, simulation)	307	76	121
(32 KB, CF, real SSD)	475	182	204
(32 KB, CF, simulation)	486	194	211

Note: "Real SSD" and "simulation" denote the solid-state disk and the simulation, respectively. "0KB" and "32KB" denote the write buffer size.

4.6. Case Study: A GP5086-Based SSD

This study reports an implementation of the proposed channel-management scheme using a real solid-state disk. The microcontroller of this solid-state disk (i.e., GP5086) was designed by Global UniChip Cooperation. This controller consists of a 150MHz ARM7 core, a BCH-based ECC engine, NAND flash interfaces, and a serial ATA host interface. GP5086 features a four-channel architecture, and each of the channels can independently accept and process flash commands. A solid-state disk was designed using GP5086 and four MLC NAND flash chips. Every channel had one flash chip. The flash chips used in this evaluation support two-plane commands, but these commands do not allow two planes to independently address their page and blocks. Therefore, this evaluation did not use the two-plane commands. The firmware of this solid-state disk implemented a FAST-like flash-translation layer. This version of FAST used multiple sequential-write log blocks and prevented write requests smaller than 4KB from allocating sequential-write log blocks. The solid-state disk was connected to a Windows-based host PC, and the host ran an application that replayed the disk traces of the three workloads via Win32 non-buffered-IO APIs. The overprovisioning ratio in this experiment was 10%, and the size of the write buffer was 32KB.

Table IV compares the results of using the solid-state disk and those of using the trace-driven simulation. Overall, the IOPSs of the solid-state disk are slightly lower than those of the simulation. This is because this simulation ignored the time overhead of transferring data over the serial ATA interface and to/from the write buffer. There are also minor differences between the standard FAST and the GP5086 firmware, as previously described. However, the performance differences are small, and they seem constant. This experiment not only validates the simulation results but also proves the feasibility of the proposed approach.

5. CONCLUSION

Solid-state disks employ multiple channels for parallel flash access to boost their data transfer rate. As realistic disk workloads produce plenty of small write requests, modern designs adopt independent channels to reduce the read-modify-write overhead and use write buffers to improve the parallelism among page writes. This study investigates the problem of low channel utilization caused by contention for the write-buffer space. This contention originated from the asynchrony among channels' garbage-collection activities. This study presents garbage-collection advancing, a technique for increasing the overlap among garbage-collection activities in different channels. This method uses idle channel cycles with garbage collection and restores the balance of buffer-space utilizations among channels. Because many commodity flash chips now support multiplane commands, this study also present cycling filling, an improved version of garbage-collection advancing for channel systems of multiplane flash chips. Cycle filling synchronizes channels' garbage collection operations but imposes no restrictions on flash addressing for planes. Thus, it retains the advantage of garbage-collection advancing while being compatible with the operation model of planes.

We conducted a series of experiments on a trace-driven simulator. Results show that garbage-collection advancing and cycle filling are useful under the disk workloads of a Windows desktop and the Iometer disk benchmark tool. Cycle filling is particularly beneficial to multichannel architectures that use multiplane flash chips. We also successfully implemented the proposed channel/plane management strategy in a real solid-state disk, and the evaluation results based on the real platform verified our simulation results.

REFERENCES

N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference on Annual Technical Conference (ATC'08)*. USENIX Association, 57–70.

L. Chang and Y. Su. 2011. Plugging versus logging: A new approach to write buffer management for solid-state disks. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 23–28.

L.-P. Chang. 2010. A hybrid approach to nand-flash-based solid-state disks. *IEEE Trans. Comput.* 59, 10, 1337–1349.

L.-P. Chang, and T.-W. Kuo. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. 187–196.

L.-P. Chang, T.-W. Kuo, and S.-W. Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4, 837–863.

H. Cho, D. Shin, and Y. I. Eom. 2009. Kast: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*. European Design and Automation Association, 507–512.

C. Dirik and B. Jacob. 2009. The performance of PC solid-state disks (SSDS) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 279–289.

A. Gupta, Y. Kim, and B. Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, 229–240.

J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee. 2007. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.* 53, 9, 644–658.

S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. 2009. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.* 58, 6, 744–758.

H. Kim and S. Ahn. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, 1–14.

S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. Embed. Comput. Syst.* 6, 3, 18.

Micron Technology. 2009. MT29F512G08 NAND Flash Memory Data Sheet. Micron Technology, Inc.

E. H. Nam, B. Kim, H. Eom, and S. L. Min. 2011. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Trans. Comput.* 60, 5, 653–666.

Open NAND Flash Interface. 2011. ONFi 3.0 Specification. Open NAND Flash Interface.

Open Source Development Lab. 2003. Iometer. http://http://www.iometer.org/.

S. Park, Y. Park, G. Shim, and K. Park. 2011. Cave: Channel-aware buffer management scheme for solid state disk. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, 346–353.

S.-H. Park, J.-W. Park, S.-D. Kim, and C. C. Weems. 2012. A pattern adaptive NAND flash memory storage structure. *IEEE Trans. Comput.* 61, 1, 134–138.

Z. Qin, Y. Wang, D. Liu, and Z. Shao 2012. Real-time flash translation layer for NAND flash memory storage systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. IEEE, 35–44.

Samsung. 2008. K9MDG08U5M 4G * 8 Bit MLC NAND Flash Memory Data Sheet. Samsung Electronics Company.

Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 905–921.

Received January 2012; revised July 2012; accepted October 2012